Tic-Tac-Toe Reloaded

VERSIONSGESCHICHTE			
NUMMER	DATUM	BESCHREIBUNG	NAME

Inhaltsverzeichnis

1 Teil I - Erste Begegnung

1.1 Compilieren eines AVR-Programms

Das Arbeitsverzeichnis für diesen Teil des Workshops ist `01_HelloWorld/`.

Ein einfaches C-Programm hello.c

```
#include <stdio.h>

int main()
{
int i;
    printf("Hello World\n");
    for (i = 0; i < 4; i++)
    {
        printf("i=%d\n", i);
    }
    return 0;
}
```

Es ist auf dem PC übersetzbar und ausführbar.

```
$ gcc hello.c
$ ./a.out
Hello World
i=0
i=1
i=2
i=3
```

Wenn man das Programm nicht mit `gcc` sondern mit `avr-gcc` übersetzt, entsteht ebenfalls ein File `a.out`, das aber nicht mehr auf einem PC ausführbar ist.

```
$ avr-gcc hello.c
$ ./a.out
bash: ./a.out: Kann die Datei nicht ausführen.
```

Was passiert im Detail? Mit der Option -v werden detailliertere Informationen zum Compile- und Linkvorgang angezeigt.

```
$ avr-gcc -v hello.c -o hello-avr.out
$ gcc -v hello.c -o hello-pc.out
$ file hello*out
hello_avr.out: ELF 32-bit LSB executable, Atmel AVR 8-bit, version 1 (SYSV),\
               statically linked, not stripped
hello_pc.out:  ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),\
               dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
```

Derzeit gibt es ca. 200 unterschiedliche AVR-Controller, die sich hinsichtlich des Speicherausbaus, der Pin-Anzahl und der integrierten Peripherie unterscheiden. Innerhalb der AVR-Familie kann man folgende Untergruppen unterscheiden:

- ATtiny*, die kleinsten und preiswertesten Controller für einfache Aufgaben.
- ATmega*, besser ausgestattet als tinyAVR, für komplexere Aufgaben, auch mit Spezial-Peripherie, z.B. zur LCD-Ansteuerung oder einem integrierten Funk-Transceiver (ATmega128RFA1).
- AT90USB* und AT90CAN* mit integriertem USB- oder CAN-Bus-Controller.
- ATxmega* ATmega-Nachfolger-Generation mit verbesserter und leistungsfähigerer Architektur.

Das Programm "hello.c" wird wie folgt für den ATmega128RFA1 compiliert.

```
$ avr-gcc -mmcu=atmega128rfa1 hello.c
```

Das Programm liegt nun als ELF-File `a.out` vor. Um es in den AVR zu laden, muss es in ein Fileformat umgewandelt werden, das vom Programm `avrdude` gelesen werden kann. Eine Möglichkeit ist z.B. das Intel-HEX-Format, das mit dem Programm `avr-objcopy` erzeugt wird.

```
$ avr-objcopy -O ihex a.out a.hex
```

1.2 Verbinden des JTAG-Programmers und Programmierung des AVR

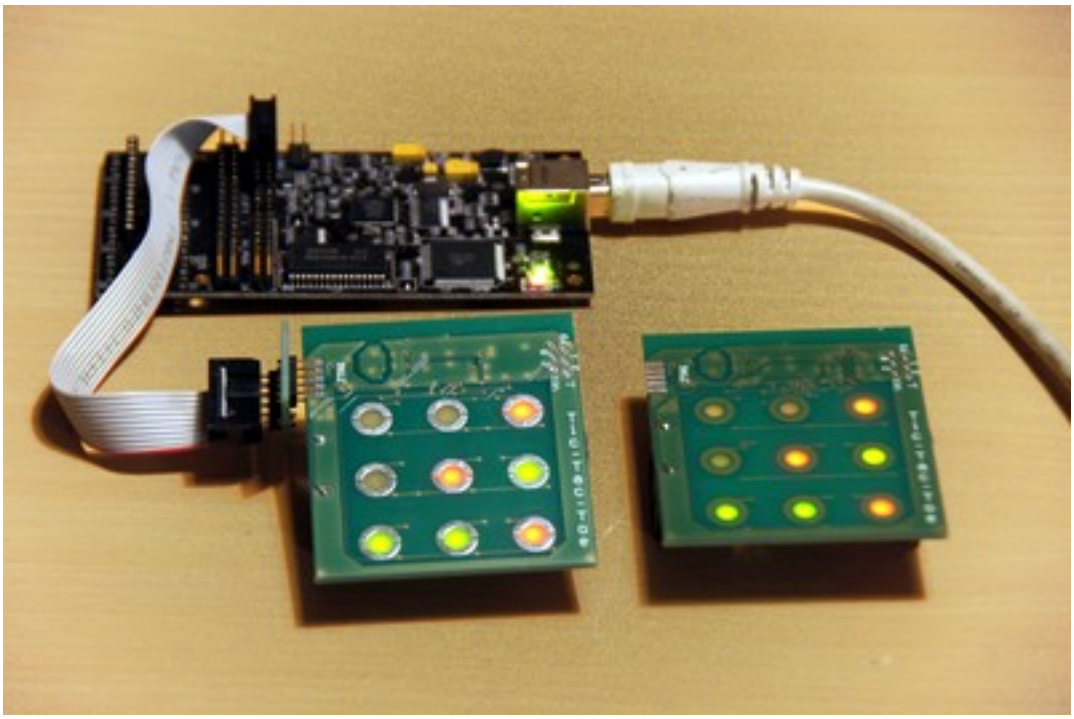


Abbildung 1: Setup mit AVR-Dragon und zwei Tic-Tac-Toe-Platinen

Mit dem Programm `avrdude` wird nun das File `a.hex` in den Mikrocontroller "geflasht".

```
$ avrdude -P usb -p atmega128rfa1 -c dragon_jtag -U fl:w:a.hex:i
```

Die Optionen des Programms bedeuten im einzelnen:

-P usb	Der Programmer ist am USB Port angeschlossen
-p atmega128rfa1	Der Mikrocontroller ist ein ATMega128RFA1
-c dragon_jtag	Der Programmer ist ein AVR Dragon im JTAG mode.
-U fl:w:a.hex:i	Das File `a.hex` wird ins Flash (fl) geschrieben (w) und liegt im Intel-HEX Fileformat (i) vor.

Nach dem Flashen des Programms sieht man im Gegensatz zum PC, das man nichts sieht :-(Das liegt daran, das kein Ausgabegerät für die `printf()` Anweisung definiert ist. Als Ursachen dafür, das nichts passiert kommen aber auch folgende Gründe in Frage:

- Programm falsch übersetzt oder umgewandelt,

- Hardware defekt,
- vagabundierende Neutronen, Aliens, ... ?

Mit einem Hardware-Debugger kann man nachweisen, ob der Mikrocontroller überhaupt den erzeugten Programmcode ausführt.

1.3 JTAG-Debugging

Beim Debugging von AVR-Programmen wird das Programm `avr-gdb` verwendet. Zusätzlich ist das Programm `avarice` erforderlich, es dient als Proxy zwischen dem AVR-Dragon und dem Programm `avr-gdb`.

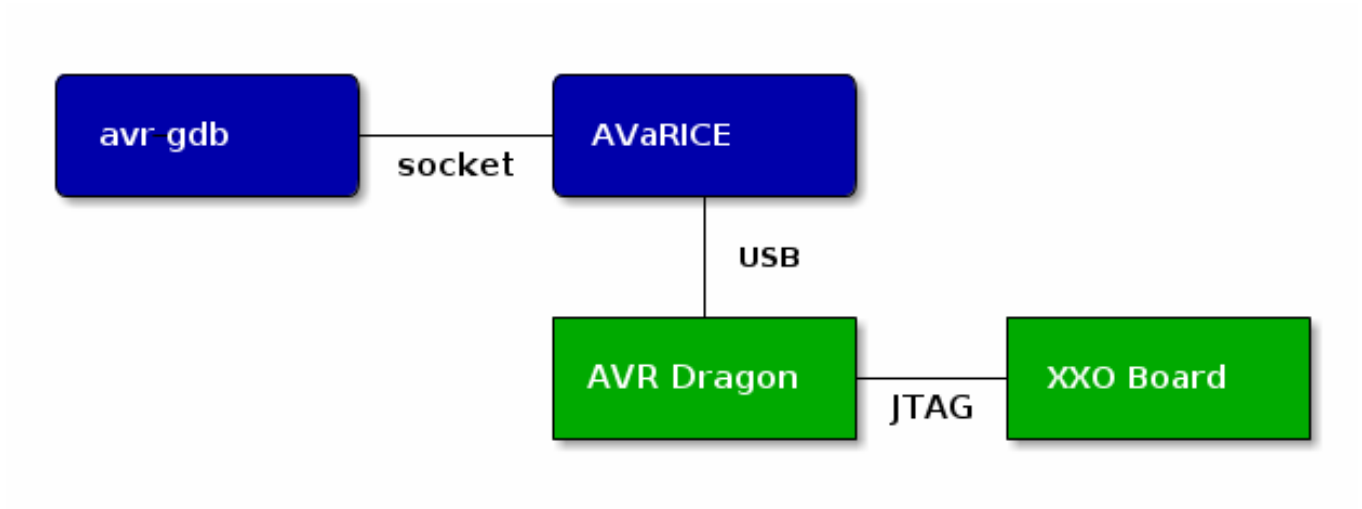


Abbildung 2: Debug-Komponenten

Zum Debuggen wird zunächst das Programm `avarice` gestartet. Es öffnet einen Socket, der vom Programm `avr-gdb` zur Kommunikation mit dem AVR-Dragon genutzt wird.

```
$ avarice -I -P atmega128rfa1 -2g --detach :4242
AVaRICE version 2.10, Jun 30 2010 20:31:30
...
Waiting for connection on port 4242.
```

Die Kommandozeilen-Optionen bedeuten im einzelnen:

-I	step over interrupts, ignore
-P atmega128rfa1	Der Mikrocontroller ist ein ATMega128RFA1
-2g	Dragon JTAG mkII Programmer der an USB Port angeschlossen ist.
--detach	Programm startet nach erfolgreichem Kontakt mit dem JTAG ICE im Hintergrund
:4242	Ein Socket auf Portnummer 4242 wird geöffnet, mit dem sich der `avr-gdb` verbindet.

Der Debugger `avr-gdb` wird mit dem ELF-File `a.out` als Argument gestartet. Es enthält neben dem eigentlichen Programmcode auch Zusatzinformationen, die zusammen mit dem Quellfile `hello.c` zum debuggen in Hochsprache benötigt werden.

```
$ avr-gdb a.out
GNU gdb 6.8
...
```

```
(gdb)
```

Jetzt muss der `avr-gdb` mit `avarice` Kontakt aufnehmen, das geschieht mit dem Befehl "target remote :4242".

```
(gdb) target remote :4242
Connection opened by host 127.0.0.1, port 59513.
0x00000000 in __vectors ()
```

Gehe zu zur Funktion `main()`.

```
(gdb) tb main
Breakpoint 1 at 0x162
(gdb) continue
Continuing.

Breakpoint 1, 0x00000162 in main ()
Current language:  auto; currently asm
     +-----------------------------^
 fehlende C-Debugsymbole!
```

Leider haben wir das Programm ohne C-Debugsymbole übersetzt, d.h. beim Compilieren wurde die `avr-gcc` Option "-g" vergessen. Um den Debugger zu beenden, benutzen wir die Kommandos "detach" und "quit".

```
(gdb) detach
Ending remote debugging.
(gdb) quit
$
```

Jetzt also nochmal das ganze, nun aber mit Debugsymbolen:

```
$ avr-gcc -g -mmcu=atmega128rfa1 hello.c
$ avr-objcopy -O ihex a.out a.hex
$ avrdude -P usb -p atmega128rfa1 -c dragon_jtag -U fl:w:a.hex:i
$ avarice -I -P atmega128rfa1 -2g  --detach :4242
$ avr-gdb -x ../debug.cfg a.out
```

Die GDB-Befehle um bis zur Funktion `main()` zu kommen kann man in einem Startup-Script zusammenfassen, siehe ../-debug.cfg. Da man beim Beenden des GDB meist den Befehl `detach` vergisst, der das Programm `avarice` beendet, ist im File `debug.cfg` eine Funktion `q` definiert, die die Befehle `detach` und `quit` kombiniert, d.h. zum Verlassen des `avr-gdb` können wir nun einfach "q" tippen.

Um nun nicht jedesmal sämtliche Befehle einzeln eingeben zu müssen, kann man sich die Arbeit durch ein Shell-Skript, z.B. ../01_HelloWorld/debug.sh erleichtern.

Zusammenfassung

> Im **Teil I** des Workshops haben wir gelernt, ein AVR Programm zu compilieren, es per `avrdude` in den Mikrocontroller zu laden und es anschließend mit `avarice` und `avr-gdb` zu debuggen. Die verwendeten Befehlszeilen sind teilweise schon recht lang, so dass man sich der Bequemlichkeit halber ein Shell-Script und eine GDB-Startup-Datei schreibt.

2 Teil II - LED Ansteuerung

2.1 IO-Ports

Aus dem Schaltplan der Tic-Tac-Toe-Platine ist ersichtlich, dass die zweifarbigen Leuchtdioden am PORTB des Mikrocontrollers angeschlossen sind. Der PORTB ist ein 8 Bit breiter digitaler IO-Port, der über 3 Register konfiguriert und programmiert werden kann.

Register	IO-Adresse	RAM-Adresse	Bezeichnung
PORTB	0x05, 0x25	0x800025	Port B Data Register
DDRB	0x04, 0x24	0x800024	Port B Data Direction Register
PINB	0x03, 0x23	0x800023	Port B Input Register

Die LED "DL1" ist an den Pins PB0 und PB3 angeschlossen. Das folgende Beispiel zeigt wie sie abwechselnd rot und grün blinkt.

LED1 Blinker led1.c

```
#include <avr/io.h>
#include <util/delay.h>


int main()
{
    DDRB = (_BV(PB0) + _BV(PB3));

    while (1)
    {
        PORTB = _BV(PB0);
        _delay_ms(250);
        PORTB = _BV(PB3);
        _delay_ms(250);
    }
    return 0;
}
```

Das Programm wird mit dem Befehl `make -f led1.mk flash debug` compiliert, in den AVR geflasht und anschließend wird der Debugger gestartet.

Um im Debugger die Registerwerte von PORTB und DDRB anzuzeigen, macht man sich die Tatsache zu nutze, dass die IO-Port-Register in den RAM-Bereich des Controllers gemappt sind.

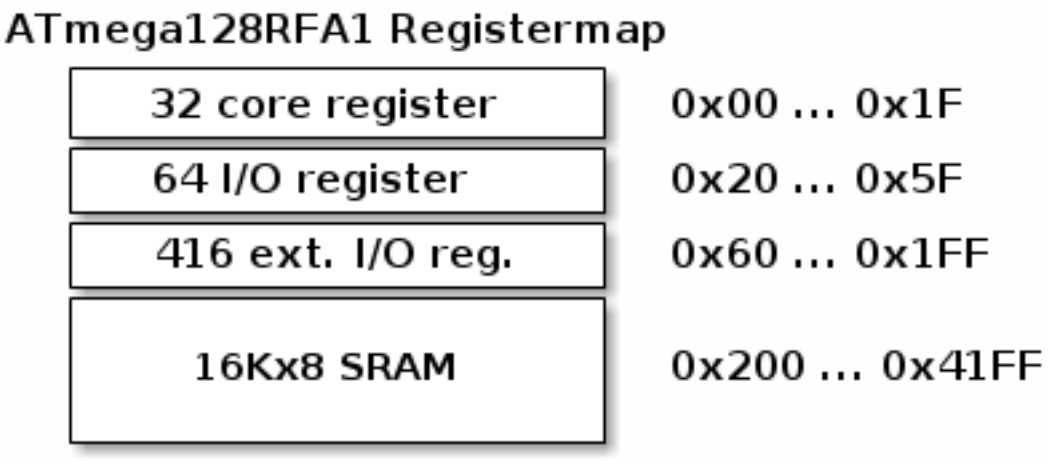


Abbildung 3: RAM-Memory-Map des ATmega128RFA1

Im `avr-gdb` kann man die Register nun wie folgt anzeigen und verändern:

```
#Anzeige DDRB
(gdb) x /b 0x800024
#Alle Port Pins fuer Zeile 1 als Ausgang freischalten
(gdb) x /b *0x800024 = 0x0f
# DL1 grün
(gdb) x /b *0x800025 = 0x1
# DL2 grün
(gdb) x /b *0x800025 = 0x2
# DL1 rot
(gdb) x /b *0x800025 = 0x8

... ?????
```

2.2 Zeilenweise Ansteuerung der LEDs

Um bei eingeschalteter Zeile 0 (ROW_0) nur eine einzelne LED "rot" zu schalten und die anderen im Zustand "aus" zu belassen, muss neben dem PORTB-Register auch noch das DDRB-Register mit dem richtig Wert programmiert werden. Wenn auf ROW_0 eine "1" getrieben wird und an der Spalte COL_0 eine "0" ausgegeben wird, dann ist die LED "rot". Wenn sie "aus" sein soll, dann muss die zugehörige Spalte abgeklemmt sein. Das erreicht man, wenn man das entsprechende Spalten-PIN hochohmig auf Eingang schaltet, dadurch fließt kein Strom und die LED ist aus.

2.3 Ansteuerung der gesamten LED-Matrix

Um alle LEDs unabhängig ansteuern zu können, muss die Aktualisierung der PORTB-Register zyklisch erfolgen.

Die Matrixansteuerung soll nun im Programm leds.c implementiert werden:

Hier ist ein Codefragment, das nun funktionsfähig gemacht werden soll leds.c.

```
#include <avr/io.h>
#include <util/delay.h>

#define COL_LEDS    (_BV(PB0) | _BV(PB1) | _BV(PB2))
#define ROW_IO_MASK (_BV(PB3) | _BV(PB4) | _BV(PB5))

#define OFF (0)
#define RED (1)
#define GREEN (2)

#define DISPLAY_RED (0)
#define DISPLAY_GREEN (3)


static uint8_t LedState[9] = { RED,   GREEN, RED,
                               RED,   OFF,   GREEN,
                               GREEN, GREEN, RED};

int main(void)
{
    uint8_t display_state = DISPLAY_GREEN;
    uint8_t i, portb, row = 0;

    while(1)
    {

        for(i = 0; i < 3; i++)
        {
            /* berechne die Variable portb für die aktuelle Zeile (row)*/
        }
```

```
        DDRB = 0;
        PORTB = portb;
        DDRB = (DDRB & ~ROW_IO_MASK) | _BV(row+3) | 7;

        /*
         * schalte die Variable display_state weiter
         *   DISPLAY_GREEN, DISPLAY_RED, DISPLAY_GREEN, ....
         */

        row ++;
        if (row > 2)
        {
            row = 0;
        }
        _delay_ms(10);
    }

    return 0;
}
```

Das Programm leds.c wird mit dem Befehl `make -f leds.mk flash dedug` geladen und debuggt.

Ein fertiges Beispiel gibt's hier: leds_ref.c.

Zusammenfassung

Im **Teil II** des Workshops haben wir die Tücken der LED-Matrix-Ansteuerung gemeistert. Dabei wurden die PORT-Register des AVR näher untersucht. Um eine statische unabhängige LED-Anzeige zu erhalten müssen die Port-Register zyklisch aktualisiert werden. Man nutzt so die Trägheit des Auges aus, um ein scheinbar statisches Bild zu erzeugen.

3 Teil III - Timer-Steuerung

3.1 Exakte Zeitbasis

Die Implementierung von Wartezeiten mittels Delay-Funktionen, wie im Beispiel leds_ref.c gezeigt wurde, ist nicht besonders flexibel. Einerseits hängt die exakte Durchlaufzeit der Schleife neben dem Delay-Wert auch noch von den anderen auszuführenden Instruktionen ab. Ändert sich das Programm, so ändert sich auch die Durchlaufzeit oder aber man korrigiert bei jeder Programmänderung den Delay-Wert manuell.

Mikrocontroller bieten mit den eingebauten Hardware-Timern eine sehr viel elegantere Lösungsvariante. Die in AVR-Controllern implementierten Timer-Blöcke bieten viele Möglichkeiten, u.a. das Messen von Zeiten anhand von Signalen an IO-Pins (input capture), die Ausgabe von PWM-Signalen und auch das Auslösen von Interrupts zu einer bestimmten programmierten Zeit.

Ein Hardware-Timer besteht im wesentlichen aus einem Zähler, der von einem Taktsignal inkrementiert wird. Durch die programmierbare Logikbeschaltung des Timers kann bei Überlauf des Zählers oder bei Erreichen eines bestimmten Zählerstandes ein Interrupt ausgelöst werden.

Das folgende Programm zeigt, wie der Timer konfiguriert wird. Hier ist die LED-Ausgabe aus dem vorherigen Workshop-Teil einzubauen. Zuvor soll mit dem Debugger getestet werden, ob die Funktion `display_leds()` aufgerufen wird.

Timergesteuertes LED-Programm timer.c

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define COL_LEDS    (_BV(PB0) | _BV(PB1) | _BV(PB2))
#define ROW_IO_MASK (_BV(PB3) | _BV(PB4) | _BV(PB5))
```

```
#define OFF (0)
#define GREEN (1)
#define RED (2)

#define DISPLAY_RED (0)
#define DISPLAY_GREEN (3)

static uint8_t LedState[9] = { RED,   GREEN, RED,
                               RED,   OFF,   GREEN,
                               GREEN, GREEN, RED};

static void io_init(void)
{
    /*
     * Timer 0 with prescaler 8, and overflow interrupts enabled.
     * This is the basic "housekeeping" interrupt which manages the
     * LED multiplexing as well as the input pad scanning.
     *
     * At F_CPU = 1 MHz, the timer rolls over at 488 Hz, or
     * approximately each 2 ms.
     *
     * In the resistive touchpad version, the pads are being scanned
     * some time later after the respective row output has been
     * activated; the actual amount of time is determined by the
     * OCR0A value.
     */
    TCCR0B = _BV(CS01);
    TIMSK0 = _BV(TOIE0);
    OCR0A = 220;
    sei();
}

static void display_leds(uint8_t row)
{
    /*
     * Aufgabe 1:
     *  Setze einen Breakpoint in dieser Funktion
     *  Wird sie aufgerufen ?
     *
     * Aufgabe 2:
     *  Implementiere diese Funktion, nach dem
     *  Beispiel ../02_Leds/leds_ref.c
     */
}

ISR(TIMER0_OVF_vect)
{
    static uint8_t row = 0;

    display_leds(row);

    row ++;
    if (row > 2)
    {
        row = 0;
    }
}

int main(void)
{
    io_init();
    while(1)
```

```
    {
    }

    return 0;
}
```

Ungeduldige finden die fertige Lösung in timer_ref.c.

3.2 Wie schnell tickt der Timer?

Die Taktfrequenz mit des Mikrocontrollers wird über sog. Fuse-Bits eingestellt. Diese Frequenz wird beim Compilieren üblicherweise mit dem Makro F_CPU angegeben. Die Delay-Funktionen der avr-libc verlangen es u.a. zwingend. Die Frequenz F_CPU kann beim ATmega128RFA1 im Bereich von wenigen Hertz bis hin zu 16 MHz eingestellt werden.

Die Tic-Tac-Toe-Platinen sind mit den Fuse-Werten LF=0x62, HF=0x11, EF=0xFE vorprogrammiert. Durch den Wert 0x62 für die Low Fuse (LF) läuft der Mikrocontroller mit 1MHz Taktfrequenz.

Timer 0 wird mit einem Teilerfaktor von 8 aufgerufen (CS02:00 = 2), d.h. der 8-Bit-Timer wird mit 125kHz getaktet und generiert so aller 2ms (256 Taktzyklen) einen Interrupt, in dem das Display aktualisiert wird.

3.3 Strom sparen

Um Strom zu sparen, kann in der Endlosschleife der Befehl `sleep_mode();` eingefügt werden. Die zugehörigen Definitionen werden durch die Include-Datei `avr/sleep.h` bereit gestellt.

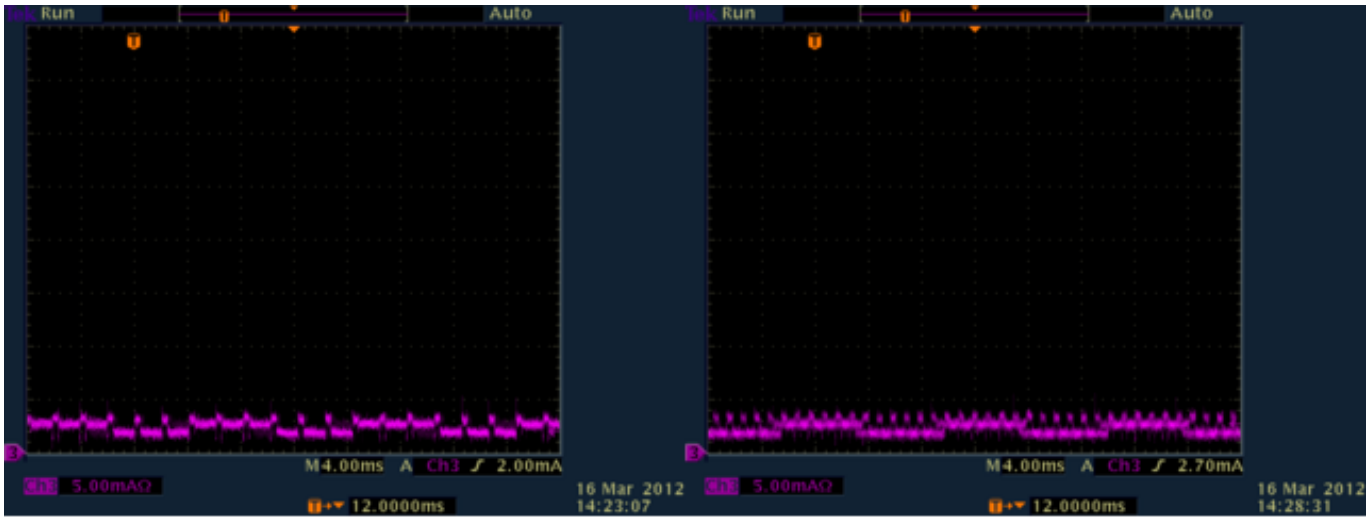
```
    while(1)
    {
        sleep_mode();
    }
```

Die AVR-Controller kennen mehrere verschiedene sogenannte *sleep modes*. Diese legen fest, welche Teile des Controllers noch getaktet werden (und damit Strom verbrauchen). Der einfachste Modus ist der sogenannte *idle*-Modus. Der komplette Controller bleibt dabei noch getaktet, nur die CPU wird angehalten. Diesen Modus stellt man durch Aufruf der Funktion `set_sleep_mode(SLEEP_MODE_IDLE);` ein.

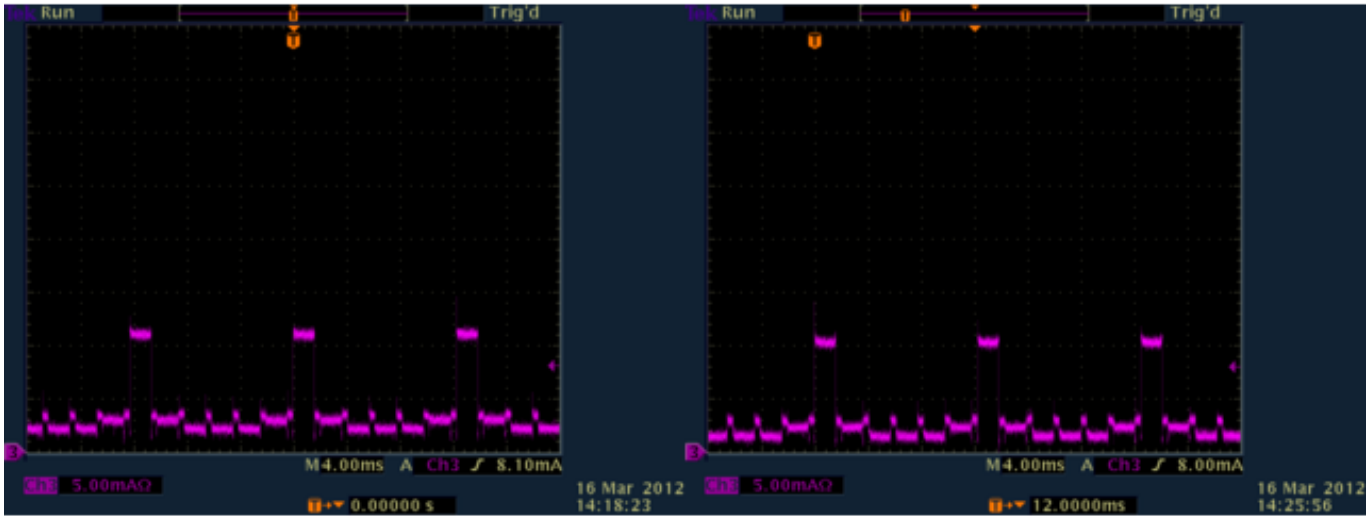
Der Controller kann aus einem Schlafzustand nur durch einen Interrupt (oder einen Reset) wieder befreit werden. Daher ist es wichtig, dass es mindestens eine freigeschaltete Interruptquelle gibt und dass die Interrupts global freigegeben sind, bevor ein Schlafzustand eingenommen wird.

3.4 Dynamischer Stromverbrauch

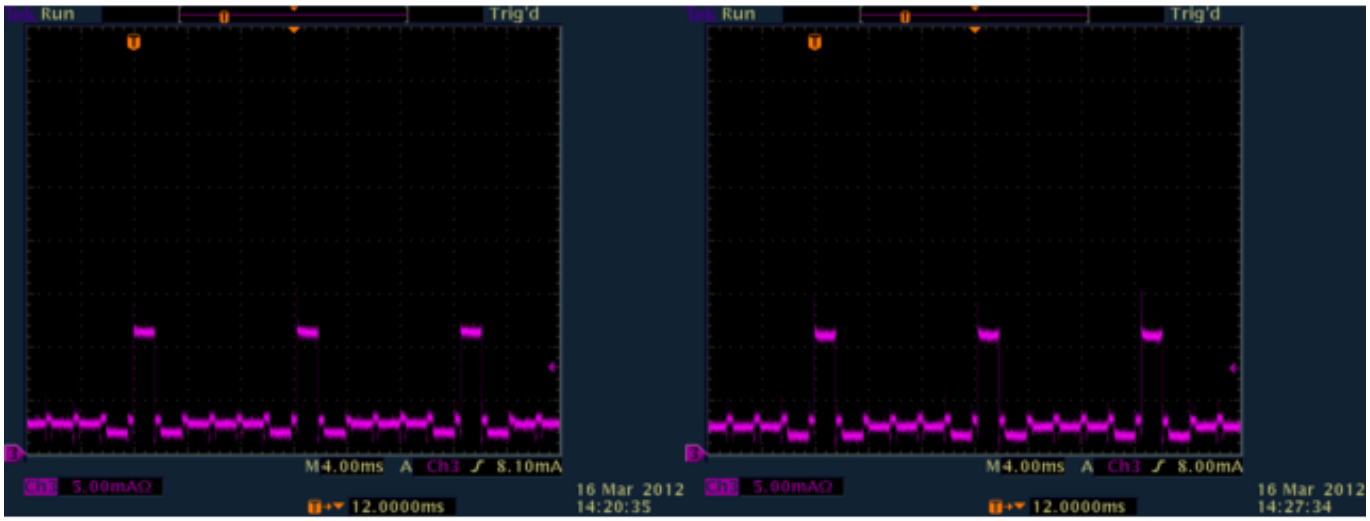
Das folgende Bild zeigt Oszillogramme, die mit einer Stromzange gemessen wurden. Die Bilder stellen den zeitlichen Stromverbrauch in verschiedenen Betriebszuständen dar.



Alle LEDs aus, MCU in spin lock | *Alle LEDs aus, MCU in sleep*



LED 4 grün, MCU in spin lock | *LED 4 grün, MCU in sleep*



LED 4 rot, MCU in spin lock | *LED 4 rot, MCU in sleep*

Zusammenfassung

Im **Teil III** des Workshops wurde ein 8-Bit-Hardware-Timer eingesetzt um eine zeitlich exakte Aktualisierung der LED-Matrix zu erreichen. Wir haben ferner gesehen, dass durch die Optimierung des Compilers leere Funktionen verschwinden bzw. normale

Funktionen durch Inline-Funktionen ersetzt werden können. Zum Schluss dieses Abschnittes wurde gezeigt, mit welcher Funktion der Mikrocontroller in den Schlafzustand versetzt werden kann. Das Aufwecken wird vom Timerinterrupt erledigt.

4 Teil IV - Platine mit Kapazitivem Touch-Sensor

4.1 Funktion des Sensors

In diesem Teil des Workshops kommt die Platine mit den goldfarbenen Touch-Pads zum Einsatz. Das Prinzip der kapazitiven Touch-Sensoren ist im nachfolgenden Bild gezeigt.

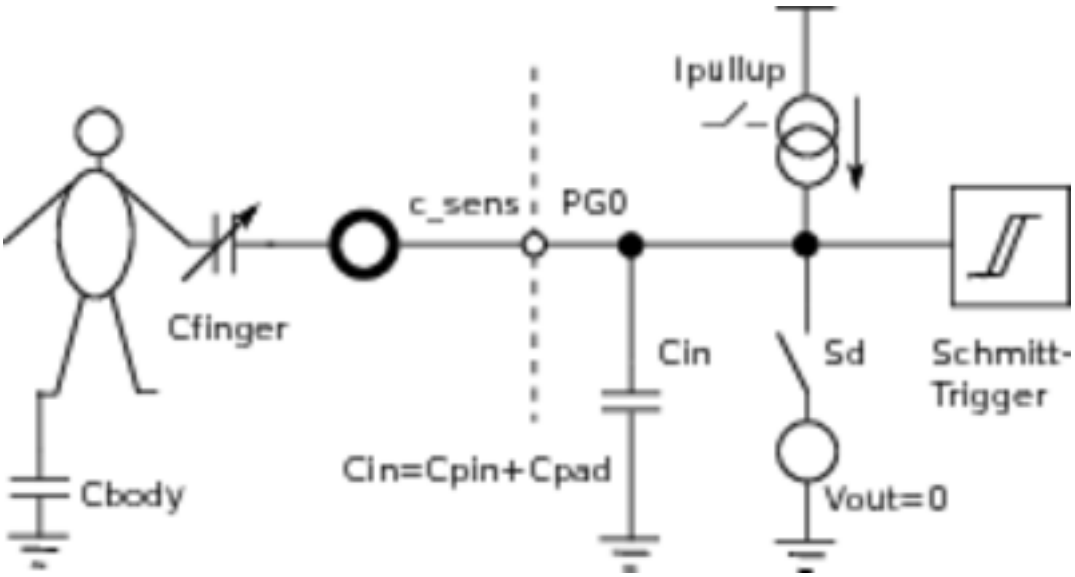


Abbildung 4: Funktion des kapazitiven Touch-Sensors

Zu Beginn der Messung wird zunächst die Kapazität C_{in} über den Schalter S_d entladen. Im Programm wird dazu der jeweilige Port auf Ausgang geschaltet und eine "0" ausgegen. Nach dieser Entladung wird der Port wieder auf "Eingang" geschaltet. Wenn ein Touch-Sensor berührt wird, dann vergrößert sich die Kapazität von C_{in} durch die Parallelschaltung von C_{body} und C_{finger} und der Ladevorgang dauert länger. Im Programm wird während des Ladevorgangs der Wert des jeweiligen PIN-Registers fortlaufend in ein Array geschrieben (`SamplesPortD[]`, `SamplesPortG[]`). Die Auswertung erfolgt dann so, dass ein Pin, das zum Zeitpunkt NSAMPLE noch den Wert 0 hat, als "Tastendruck" bewertet wird. NSAMPLE wurde dabei einmal exemplarisch ermittelt.

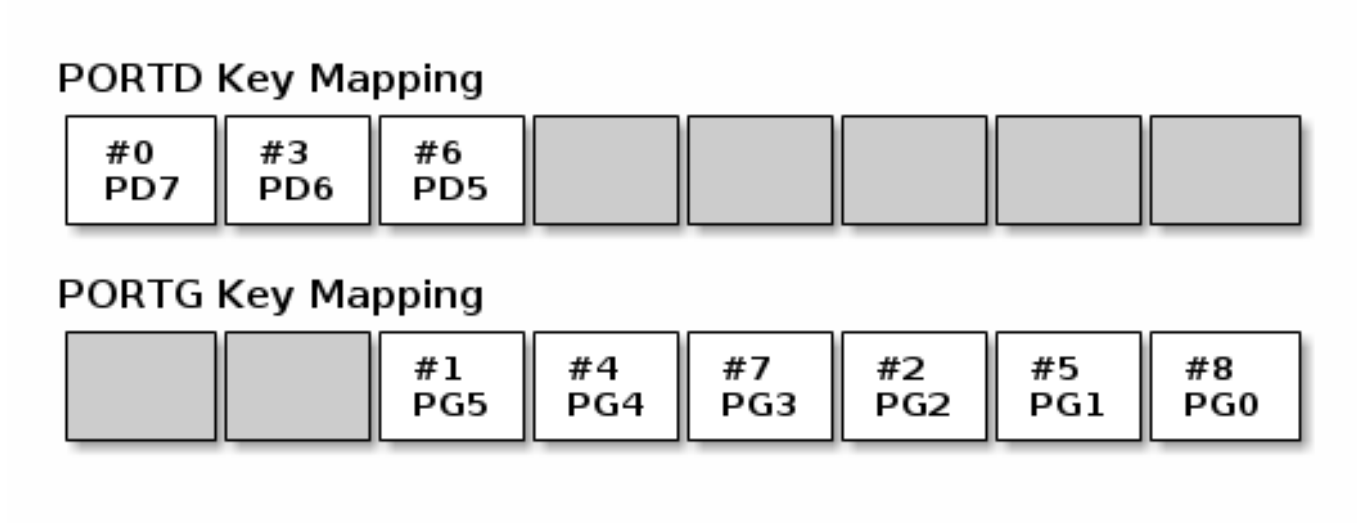
Hier ist ein Beispiel für eine Wertefolge die im Debugger beim berühren von Key #0 (PD7) gemessen wurde.

```
(gdb) print /x SamplesPortD
$1 = {0x1f, 0x1f, 0x7f, 0xff <repeats 13 times>}

PIND: 0x1f   0x1f   0x7f   0xff    0xff
PD7 :   0     0     *0*     1
PD6 :   0     0      1      1
PD5 :   0     0      1      1
----+------+------+------+------+---------->
       t0     t1     t2    t3
```

4.2 Tasten Sortieren und Auswerten

Die Tasten-Nummern scheinen auf den ersten Blick den Port-Pins recht willkürlich zugeordnet zu sein, was dem Platinen-Layout geschuldet ist. Die Software muss also die Tasten-Nummern und die Port-Pins einander zuordnen. Im folgenden Bild ist das

Mapping von Tasten und Port-Pins dargestellt.



Im Beispielprogramm captouch.c fehlt noch die Tastenzuordnung und die Tastendruckerkennung.

```
/* === includes ============================================================ */
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include "captouch.h"

/* === macros ============================================================== */
#define NSAMPLES (16) /* must match the inline asm below! */
#define NSAMPLE  (2)
#define KEY_NONE (255)

/* === globals ============================================================= */
uint8_t LedState[9] = { RED,   GREEN, RED,
                        RED,   OFF,   GREEN,
                        GREEN, GREEN, RED};

static uint8_t SamplesPortD[NSAMPLES], SamplesPortG[NSAMPLES];
static uint8_t PadState[9];

/* === functions =========================================================== */

/*
 * Sample port D's input pins for their attached capacitance.
 *
 * First, the port is set to output, at low level, to discharge the
 * capacitor attached to the pin.  Then, the port is turned into an
 * input, and the input pullups are applied.  This causes the input
 * capacitor to be slowly charged, while the digital input register is
 * continuously sampled 16 times.  As the timing of this part is
 * crucial, inline assembly is used to quickly (and in equal time
 * steps) sample the input data into registers, from where they can be
 * stored into their final destination later on (by the compiler).
 */
static void sample_port_d(void)
{
    PORTD = 0;
    DDRD = 0xe0;
    __asm ("nop");
```

```
    DDRD = 0;
    PORTD = 0xe0;
    __asm ("in %[s0], %[pind]" "\n\t"
           "in %[s1], %[pind]" "\n\t"
           "in %[s2], %[pind]" "\n\t"
           "in %[s3], %[pind]" "\n\t"
           "in %[s4], %[pind]" "\n\t"
           "in %[s5], %[pind]" "\n\t"
           "in %[s6], %[pind]" "\n\t"
           "in %[s7], %[pind]" "\n\t"
           "in %[s8], %[pind]" "\n\t"
           "in %[s9], %[pind]" "\n\t"
           "in %[s10], %[pind]" "\n\t"
           "in %[s11], %[pind]" "\n\t"
           "in %[s12], %[pind]" "\n\t"
           "in %[s13], %[pind]" "\n\t"
           "in %[s14], %[pind]" "\n\t"
           "in %[s15], %[pind]"
           :
           /* output operands */
           [s0] "=r" (SamplesPortD[0]),
           [s1] "=r" (SamplesPortD[1]),
           [s2] "=r" (SamplesPortD[2]),
           [s3] "=r" (SamplesPortD[3]),
           [s4] "=r" (SamplesPortD[4]),
           [s5] "=r" (SamplesPortD[5]),
           [s6] "=r" (SamplesPortD[6]),
           [s7] "=r" (SamplesPortD[7]),
           [s8] "=r" (SamplesPortD[8]),
           [s9] "=r" (SamplesPortD[9]),
           [s10] "=r" (SamplesPortD[10]),
           [s11] "=r" (SamplesPortD[11]),
           [s12] "=r" (SamplesPortD[12]),
           [s13] "=r" (SamplesPortD[13]),
           [s14] "=r" (SamplesPortD[14]),
           [s15] "=r" (SamplesPortD[15])
           :
           /* input operands */
           [pind] "I" (_SFR_IO_ADDR(PIND)));
}

/*
 * Same as for port D above, but only portpin G0 is used as an input,
 * all other pins are not used for that purpose (but are sampled
 * anyway, as sampling always applies to a full port).
 */
static void sample_port_g(void)
{
    PORTG = 0;
    DDRG = 0x3f;
    __asm ("nop");
    DDRG = 0;
    PORTG = 0x3f;
    __asm ("in %[s0], %[ping]" "\n\t"
           "in %[s1], %[ping]" "\n\t"
           "in %[s2], %[ping]" "\n\t"
           "in %[s3], %[ping]" "\n\t"
           "in %[s4], %[ping]" "\n\t"
           "in %[s5], %[ping]" "\n\t"
           "in %[s6], %[ping]" "\n\t"
           "in %[s7], %[ping]" "\n\t"
           "in %[s8], %[ping]" "\n\t"
```

```
            "in %[s9], %[ping]" "\n\t"
            "in %[s10], %[ping]" "\n\t"
            "in %[s11], %[ping]" "\n\t"
            "in %[s12], %[ping]" "\n\t"
            "in %[s13], %[ping]" "\n\t"
            "in %[s14], %[ping]" "\n\t"
            "in %[s15], %[ping]"
            :
            /* output operands */
            [s0] "=r" (SamplesPortG[0]),
            [s1] "=r" (SamplesPortG[1]),
            [s2] "=r" (SamplesPortG[2]),
            [s3] "=r" (SamplesPortG[3]),
            [s4] "=r" (SamplesPortG[4]),
            [s5] "=r" (SamplesPortG[5]),
            [s6] "=r" (SamplesPortG[6]),
            [s7] "=r" (SamplesPortG[7]),
            [s8] "=r" (SamplesPortG[8]),
            [s9] "=r" (SamplesPortG[9]),
            [s10] "=r" (SamplesPortG[10]),
            [s11] "=r" (SamplesPortG[11]),
            [s12] "=r" (SamplesPortG[12]),
            [s13] "=r" (SamplesPortG[13]),
            [s14] "=r" (SamplesPortG[14]),
            [s15] "=r" (SamplesPortG[15])
            :
            /* input operands */
            [ping] "I" (_SFR_IO_ADDR(PING)));
}

uint8_t update_pads(void)
{
    uint8_t scans[9], i;
    uint8_t *pscan, *pstate, ret;
    ret = KEY_NONE;
    sample_port_d();
    sample_port_g();

    /* Map the */
    for (i=0; i<9;i++)
    {
        /*
         * Implementiere die Pin-Zuordnung und die Tastenerkennung hier.
         */
    }

    return ret;
}

ISR(TIMER0_OVF_vect)
{
    static uint8_t row = 0;
    uint8_t key;

    key = update_pads();

    if (key != KEY_NONE)
    {
        LedState[key] ++;
        if(LedState[key] > 2)
        {
            LedState[key] = 0;
```

```
        }
    }

    display_leds(row);

    row ++;
    if (row > 2)
    {
        row = 0;
    }
}

int main(void)
{
    io_init();
    set_sleep_mode(SLEEP_MODE_IDLE);
    while(1)
    {
        sleep_mode();
    }

    return 0;
}
```

Ungeduldige finden die fertige Lösung in captouch_ref.c.

Zusammenfassung

Im **Teil IV** des Workshops wurde die Eingabe-Routine für die kapazitive Platine implementiert. Dabei wurde für das schnelle Abtasten des PIN-Registers Gebrauch von Inline-Assemblercode gemacht. Ferner wurde gezeigt, dass Einsparungen beim Hardware-Design zu einem erhöhten Programmieraufwand führen kann (Tastenzuordnung). Da aber die Ressourcen des Controllers physikalisch gegeben sind, bringt es keinen Vorteil, wenn man sie nicht nutzt, man bekommt kein Geld vom Schaltkreishersteller zurück, wenn man z.B. den RAM verwendet, oder wie Donald Knuth sagte: "Premature optimization is the root of all evil (or at least most of it) in programming."

5 Teil V - Resistive Touch Sensoren

5.1 Funktion des Sensors

Beim resistiven Touch-Sensor wird die Kapazität C_{in} über den Wiederstand R_{finger} aufgeladen. Das folgende Bild zeigt das Ersatzschaltbild.

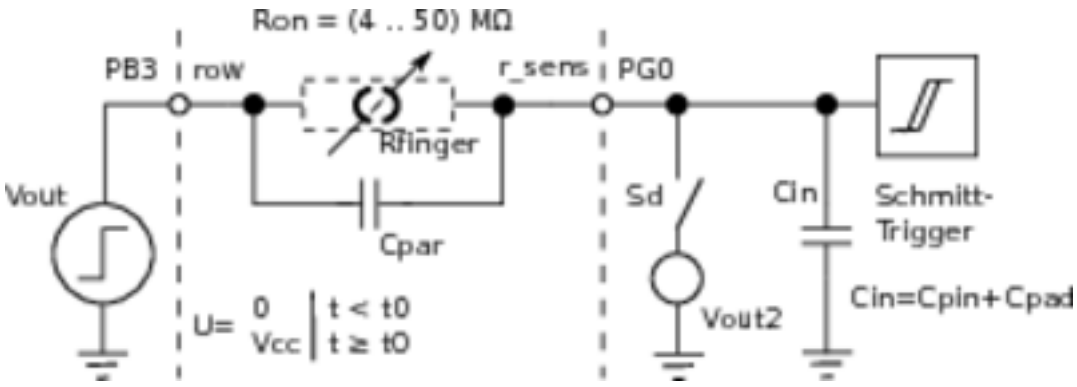


Abbildung 5: Funktion des resistiven Touch-Sensors

Zu Beginn der Messung wird PORTB und PORTD auf Augsgang geschaltet und eine "0" ausgegeben, wodurch C_{in} über den Schalter S_d entladen wird. Danach wird auf der Zeilen-Leitung (PORTB) eine "1" ausgegeben und PORTD (Spalten-Leitung) auf Eingang geschaltet. Wenn der Touch-Sensor durch einen Fingerdruck gebrückt ist, lädt sich über den Widerstand R_{finger} die Kapazität C_{in} auf und am Eingangspin der Zeilenleitung kann eine "1" detektiert werden. Um die Sicherheit der Eingabe zu verbessern, muss mehrmals hintereinander ein Tastendruck erkannt sein, bevor ein Tasten-Ereignis ausgegeben wird (`SCAN_THRS_PRESS`).

In der Timer-Interruptroutine wird vor der Display-Aktualisierung die Tastatur-Abfrage durchgeführt. Da die jeweilige Zeilenleitung für die Dauer der Messung auf 1 gesetzt wird, leuchten also ganz kurzzeitig die roten LEDs, d.h. wenn alle LEDs aus sind, sieht man bei Dunkelheit doch einen leichten Rotschimmer.

5.2 Individuelle Tastatur-Abfrage

Der Widerstand R_{finger} variiert von Person zu Person sehr stark. Menschen mit sehr trockener Haut haben einen hohen Widerstand. Je nach Widerstandswert verkürzt oder verlängert sich damit die Aufladezeit von C_{in}. Es kann auch vorkommen, dass man stärker drücken muss, um eine Reaktion zu erzielen. In der folgenden Tabelle sind die Hautwiderstandswerte, die bei einer informellen Messung von zehn Probanden ermittelt wurden, dargestellt. Die Werte variieren je nach Anpressdruck und Person zwischen 1,4 MΩ und mehr als 59 MΩ, d.h sie streuen um einen Faktor >40.

Proband	**R_{finger} (links) / MΩ**	**R_{finger} (rechts) / MΩ**
1	20,0 … 55,0	36,0 … > 59,0
2	1,8 … 14,0	8,0 … 12,0
3	5,0 … 7,0	13,0 … 14,0
4	4,5 … 7,0	1,9 … 3,0
5	1,5 … 6,7	4,3 … 6,5
6	7,6 … 20,5	6,8 … 20,0
7	2,0 … 14,8	2,0 … 26,0
8	2,0 … 4,9	2,0 … 5,0
9	1,4 … 6,0	2,0 … 12,0
10	2,5 … 10,0	1,6 … 2,3

Als Aufgabe bietet es sich an, im Programm restouch.c die Algorithmusparameter zu variieren und zu schauen wie sich das Reaktionsverhalten des Programms verändert.

```
/* === includes ============================================================ */
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <util/delay.h>
#include "restouch.h"

/* === macros ============================================================== */
#define NSAMPLES (16) /* must match the inline asm below! */
#define NSAMPLE  (2)
#define KEY_NONE (255)
#define SCAN_LOWER (2)
#define SCAN_UPPER (6)
#define SCAN_THRS_RELEASE (SCAN_LOWER + 1)
#define SCAN_THRS_PRESS   (SCAN_UPPER - 1)

#define DISCHARGE_TIME   (30) /* time in us, to uncharge C_in */
#define CHARGE_TIME      (60) /* time in us, to charge C_in */

/* === globals ============================================================= */
uint8_t LedState[9] = { RED,   GREEN, RED,
                        RED,   OFF,   GREEN,
                        GREEN, GREEN, RED};
```

```
static uint8_t PadState[9] = { '_', '_', '_',
                               '_', '_', '_',
                               '_', '_', '_'  };

/* === functions ============================================================ */


uint8_t update_pads(uint8_t row)
{
    static uint8_t scans[9] = { SCAN_LOWER, SCAN_LOWER, SCAN_LOWER,
                                SCAN_LOWER, SCAN_LOWER, SCAN_LOWER,
                                SCAN_LOWER, SCAN_LOWER, SCAN_LOWER };

    uint8_t portb, *pscan, *pstate, i, pind, ret, keyidx;

    ret = KEY_NONE;
    portb = PORTB;

    /* discharge C_in */
    PORTB = 0;
    DDRB  = ROW_IO_MASK;
    DDRD = COL_PADS;
    _delay_us(DISCHARGE_TIME);

    /* charge C_in */
    DDRD = 0x0;
    switch(row)
    {
        case 0:
            PORTB = _BV(PB3);
            break;
        case 1:
            PORTB = _BV(PB4);
            break;
        case 2:
            PORTB = _BV(PB5);
            break;
    }
    _delay_us(CHARGE_TIME);
    pind = PIND;
    PORTB = portb;
    keyidx = row*3;
    pscan = &scans[keyidx];

    /* accumulate charge values */
    pstate = &PadState[keyidx];
    for (i = 0; i < 3; i++)
    {
        *pscan += (pind & _BV(i+5)) ? +1 : -1;
        if (*pscan < SCAN_LOWER)
        {
            *pscan = SCAN_LOWER;
        }
        if (*pscan > SCAN_UPPER)
        {
            *pscan = SCAN_UPPER;
        }

        /* detect key events */
        if ((*pscan < SCAN_THRS_RELEASE) && (*pstate != '_'))
        {
            *pstate = '_';
```

```
        }


        if ((*pscan > SCAN_THRS_PRESS) && (*pstate != '#'))
        {
            *pstate = '#';
            ret = keyidx;
            __asm("nop");
        }

        pscan ++;
        pstate ++;
        keyidx ++;
    }

    return ret;
}

ISR(TIMER0_OVF_vect)
{
    static uint8_t row = 0;
    uint8_t key;

    key = update_pads(row);

    if (key != KEY_NONE)
    {
        LedState[key] ++;
        if(LedState[key] > 2)
        {
            LedState[key] = 0;
        }
    }

    display_leds(row);

    row ++;
    if (row > 2)
    {
        row = 0;
    }
}

int main(void)
{
    io_init();
    set_sleep_mode(SLEEP_MODE_IDLE);
    while(1)
    {
        sleep_mode();
    }

    return 0;
}
```

Zusammenfassung

Im **Teil V** des Workshops wurde die Ansteuerung der resistiven Touch-Sensoren implementiert. Durch die hohe Varianz, die die Werte des Hautwiderstandes annehmen können, kann der Algorithmus nicht optimal stabil für alle Benutzer eingestellt werden. Benutzer mit sehr trockener Haut und demzufolge hohem Hautwiderstand müssen stärker und tw. auch länger den Touchsensor berühren als andere.

6 Teil VI - Funkkommunikation

6.1 Die Radio Library

Nachdem die Ansteuerung der LEDs und der Touch-Sensoren fertig ist, fehlt noch die Funkkommunikation zwischen den beiden Tic-Tac-Toe-Platinen. Um den Radio-Transceiver des ATmega128RFA1 zu benutzen verwenden wir die Funktionen der Radio-Library des μracoli-Projektes. Im wesentlichen sind drei Module zu implementieren:

- die Initialisierung des Transceivers,
- das Senden von Daten,
- und das Empfangen von Daten.

Das Paket `uracoli-src-0.2.0.zip` wird wie folgt installiert und für die Tic-Tac-Toe Hardware vorbereitet.

```
$ unzip uracoli-src-0.2.0.zip
$ ln -sfv uracoli-src-0.2.0 uracoli
$ make -C uracoli/src xxo
```

Im Makefile ../06_Funk/funk.mk findet man die Kommandozeile für den Compiler-Aufruf:

```
avr-gcc -I../uracoli/inc/ -Dxxo -DF_CPU=1000000UL  -O2 -g -mmcu=atmega128rfa1 \
        -o funk_cap.out \
        funk.c leds.c captouch.c \
        -L../uracoli/lib -lradio_xxo
```

Die Kommandozeilenoptionen bedeuten im einzelnen:

-I../uracoli/inc/	Suchpfad für die μracoli-Include-Dateien
-Dxxo	Definition des Macros für das Board "xxo"
-O2 -g	Optimierung Stufe 2 und Debugsymbole
-mmcu=atmega128rfa1	für einen ATmega128RFA1-Mikrocontroller
-DF_CPU=1000000UL	Taktfrequenz des Mikrocontrollers
-o funk_cap.out	erzeuge ein ELF-File mit dem Namen `funk_cap.out` oder `funk_res.out`
funk.c ...	die Quelldateien der einzelnen Module
-L../uracoli/lib	Suchpfad für die μracoli-Radio-Library
-lradio_xxo	Linke die Library `libradio_xxo.a`

Das Makefile wird wie folgt benutzt:

```
# Alle Programme komplett neu bauen
make -f funk.mk clean all
# Flashen und Debuggen des resistiven Boards
make cflash debug BOARD=res
# Flashen und Debuggen des kapazitiven Boards
make -f funk.mk flash debug BOARD=cap
```

6.2 Initialisierung des Transceivers

Die Initialisierung des Transceivers und der Radio-Library erfolgt in der Funktion

```
static uint8_t RxFrame[TRX_FRAME_SIZE];

void xxo_radio_init(void)
{
```

```
  /* zuweisen des Empfangspuffers RxFrame */
  radio_init(RxFrame, sizeof(RxFrame));

  /* einstellen des Funkkanals */
  radio_set_param(RP_CHANNEL(CHANNEL));

  /* Als Defaultzustand soll der Transceiver im Zustand RX_ON sein */
  radio_set_param(RP_IDLESTATE(STATE_RX));

  /* Zuerst wird der Zustand TRX_OFF eingestellt */
  radio_set_state(STATE_RX);

  /* Initialisierung der globalen Variable */
  RadioRxKey = KEY_NONE;
  RadioTxKey = KEY_NONE;
}
```

6.3 Senden eines Rahmens

Das Senden eines Rahmens wird durch zwei Funktionen implementiert. Die Funktion `xxo_send()` füllt den Sendepuffer aus. Wir versenden hier bereits einen richtigen IEEE-802.15.4-Rahmen, bestehend aus dem Steuerfeld (16 Bit FCF), der Sequenznummer und den Empfänger- und Absender-Adressen. Nach der Payload des Rahmens (key) ist noch ein 16 Bit CRC-Feld enthalten, dass von Transceiver berechnet wird. Die Funktion `usr_radio_tx_done()` ist eine Callback-Funktion, die von der TX-Interruptroutine aufgerufen wird. Hier wird das Flag `TxInProgress` zurück auf 0 gesetzt und damit der nächste Rahmen gesendet werden kann.

```
void xxo_send(uint8_t key)
{
    static uint8_t seqno = 0;
    xxo_frame_t txbuf;

    /* fill frame information */
    txbuf.fcf = FRAME_CTRL_FIELD;
    txbuf.seq = seqno++;
    txbuf.panid = PANID;
    txbuf.dst = 0xffff;
    txbuf.src = SHORTADDR;
    radio_set_state(STATE_TXAUTO);

    /* fill payload */
    txbuf.key = key;

    /* send frame */
    TxInProgress = true;

    radio_send_frame(sizeof(xxo_frame_t), (uint8_t*)&txbuf, 0);

    set_sleep_mode(SLEEP_MODE_IDLE);
    while (TxInProgress)
    {
        sleep_mode();
    }
}

void usr_radio_tx_done(radio_tx_done_t status)
{
    TxInProgress = false;
}
```

6.4 Empfangen eines Rahmens

Die Verarbeitung von empfangenen Rahmen erfolgt hauptsächlich in der Funktion `usr_radio_receive_frame()`. Das ist ebenfalls eine Callback-Funktion, die von der Receive-Interrupt-Routine aufgerufen wird. Hier wird der gesendete Key aus der Payload in die Variable `RadioRxKey` kopiert.

```
uint8_t * usr_radio_receive_frame(uint8_t len, uint8_t *frm,
                                  uint8_t lqi, int8_t ed, uint8_t crc)
{
    xxo_frame_t *pframe;
    pframe = (xxo_frame_t *)frm;
    RadioRxKey = pframe->key;
    return frm;
}

uint8_t xxo_radio_get_event(void)
{
uint8_t ret;

    cli();
    ret = RadioRxKey;
    RadioRxKey = KEY_NONE;
    sei();
    return ret;
}
```

6.5 LED-Battle

Das folgende Programm implementiert das einfache Senden und Empfangen von Tastendrücken. Durch den Druck einer Taste wird die lokale LED grün und jedes Board, das den Rahmen empfängt, schaltet seine entsprechend LED auf rot.

```
/* === includes ============================================================ */
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <util/delay.h>

/* uracoli libraries */
#include <board.h>
#include <radio.h>

#include "funk.h"

/* === macros ============================================================== */
#define KEY_NONE (255)

/* === globals ============================================================= */
uint8_t LedState[9] = { RED,   GREEN, RED,
                        RED,   OFF,   GREEN,
                        GREEN, GREEN, RED};

static uint8_t RxFrame[TRX_FRAME_SIZE];
static volatile bool TxInProgress;
static volatile uint8_t RadioRxKey;
static volatile uint8_t RadioTxKey;

/* === functions =========================================================== */

/* === radio init ========== */
void xxo_radio_init(void)
```

```
{
  radio_init(RxFrame, sizeof(RxFrame));
  radio_set_param(RP_CHANNEL(CHANNEL));
  radio_set_param(RP_IDLESTATE(STATE_RX));
  radio_set_param(RP_TXPWR(-4));

  radio_set_state(STATE_RX);
  RadioRxKey = KEY_NONE;
  RadioTxKey = KEY_NONE;
}

/* === transmit functions === */
void xxo_send(uint8_t key)
{
    static uint8_t seqno = 0;
    xxo_frame_t txbuf;

    /* fill frame information */
    txbuf.fcf = FRAME_CTRL_FIELD;
    txbuf.seq = seqno++;
    txbuf.panid = PANID;
    txbuf.dst = 0xffff;
    txbuf.src = SHORTADDR;
    radio_set_state(STATE_TXAUTO);

    /* fill payload */
    txbuf.key = key;

    /* send frame */
    TxInProgress = true;

    radio_send_frame(sizeof(xxo_frame_t), (uint8_t*)&txbuf, 0);

    set_sleep_mode(SLEEP_MODE_IDLE);
    while (TxInProgress)
    {
        sleep_mode();
    }
}

void usr_radio_tx_done(radio_tx_done_t status)
{
    TxInProgress = false;
}

/* === receive functions === */
uint8_t * usr_radio_receive_frame(uint8_t len, uint8_t *frm, uint8_t lqi, int8_t ed,  ↩
    uint8_t crc)
{
    xxo_frame_t *pframe;
    pframe = (xxo_frame_t *)frm;
    RadioRxKey = pframe->key;
    return frm;
}

uint8_t xxo_radio_get_event(void)
{
uint8_t ret;

    cli();
    ret = RadioRxKey;
    RadioRxKey = KEY_NONE;
```

```
    sei();
    return ret;
}

ISR(TIMER0_OVF_vect)
{
    static uint8_t row = 0;
    uint8_t key;

    key = update_pads(row);

    if (key != KEY_NONE)
    {
        LedState[key] = GREEN;
        if (RadioTxKey == KEY_NONE)
        {
            RadioTxKey = key;
        }
    }

    display_leds(row);

    row ++;
    if (row > 2)
    {
        row = 0;
    }
}

int main(void)
{
    uint8_t radio_key;
    io_init();
    xxo_radio_init();

    set_sleep_mode(SLEEP_MODE_IDLE);

    while(1)
    {

        radio_key = xxo_radio_get_event();
        if (radio_key!= KEY_NONE)
        {
            LedState[radio_key] = RED;
        }
        if (RadioTxKey != KEY_NONE)
        {
            xxo_send(RadioTxKey);
            RadioTxKey = KEY_NONE;
        }

        sleep_mode();
    }

    return 0;
}
```

Das Programm hat nun das entscheidendes Manko, das jeder die LEDs aller anwesenden Tic-Tac-Toe-Platinen beeinflussen kann. In der Übung soll nun ein kleines Adressfilter programmiert werden, so dass man nur die eigenen Platinen beeinflusst.

Zusammenfassung

Im **Teil VI** des Workshops wurde eine Funkkommunikation der Platinen implementiert. Die lokalen Tastendrücke wurden jeweils zu einem/mehreren Empfänger(n) übertragen und dort angezeigt. Um ein funktionierendes Spiel zu implementieren muss der gesamte Funkverkehr noch ein wenig koordiniert und eine Spielauswertung implementiert werden.

7 Teil VII - Die erste Version des Spiels

7.1 Selbst- und Partnerfindung

Eine wichtige Aufgabe ist die Kontaktaufnahme mit dem Spielpartner. Um die Platinen eindeutig zu identifizieren, ist jedes der Tic-Tac-Toe-Boards mit einer 16-Bit-Seriennummer und einer weltweit eindeutigen 64-Bit-MAC-Adresse ausgestattet (die 64-Bit-MAC-Adresse wird im Workshop zunächst nicht verwendet). Diese Informationen sind auf dem Adressaufkleber und im EEPROM gespeichert. Die EEPROM-Informationen können mit dem Programm `avrdude` ausgelesen und überprüft werden. Zusätzlich liegt jedem Boardsatz ein Ausdruck der beiden EEPROM-Inhalte bei.

```
$ avrdude -P usb -p atmega128rfa1 -c dragon_jtag -tF

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1ea701
avrdude> d ee 0 16
>>> d ee 0 16
0000  01 00 42 42 c6 03 17 ff  ff 25 04 00 11 00 00 a4  |..BB.....%......|

avrdude>
```

Die Informationen im EEPROM sind mit der Struktur `node_config_t` formatiert. Diese Struktur ist im Headerfile board.h definiert.

```
typedef struct
{
    uint16_t short_addr;
    uint16_t pan_id;
    uint64_t ieee_addr;
    uint8_t  channel;
    uint8_t _reserved_[2];
    uint8_t crc;
} node_config_t;
```

Mit der Funktion `get_node_config_eeprom()` kann der EEPROM-Inhalt ausgelesen werden. Die 8-Bit-CRC am Ende der Struktur stellt die Integrität der Daten sicher und die Funktion gibt den Wert 0 zurück, wenn die Prüfsumme über die Daten richtig berechnet ist.

Bei der Vergabe der Seriennummern wurde darauf geachtet, dass die kapazitiven Platinen jeweils eine ungeradzahlige Seriennummer und die resistiven Platinen die darauffolgende geradzahlige Seriennummer haben. Ein Tic-Tac-Toe-Platinen-Paar hat somit die Seriennummern (2n-1) und (2n) mit, n = 1, 2, 3, 4, 5, ... Den jeweiligen Spielpartner erreicht man, in dem man das untere Bit der Short-Adresse auswertet und +1 oder -1 addiert.

```
      eigene SN.        Partner SN
CAP   0x0001     +   1    0x0002
RES   0x0002     -   1    0x0001
...
CAP   0x0007     +   1    0x0008
RES   0x0008     -   1    0x0007


PEER_ADDR = MY_ADDR + (MY_ADDR & 1) ? +1 : -1;
```

Neben den einfachen Sende- und Empfangsfunktionen, die im letzten Teil des Workshops verwendet wurden, hat der Transceiver noch einen automatischen Sende- und Empfangs-Modus, bei dem der Rahmenaustausch adressgefiltert und mit einem Antwort-Rahmen zur Empfangsbestätigung gesendet werden kann. Um diese Funktionen zu nutzen, muss der Transceiver nur etwas anders konfiguriert werden, wie es im Modul ../07_Spiel/funk.c implementiert ist.

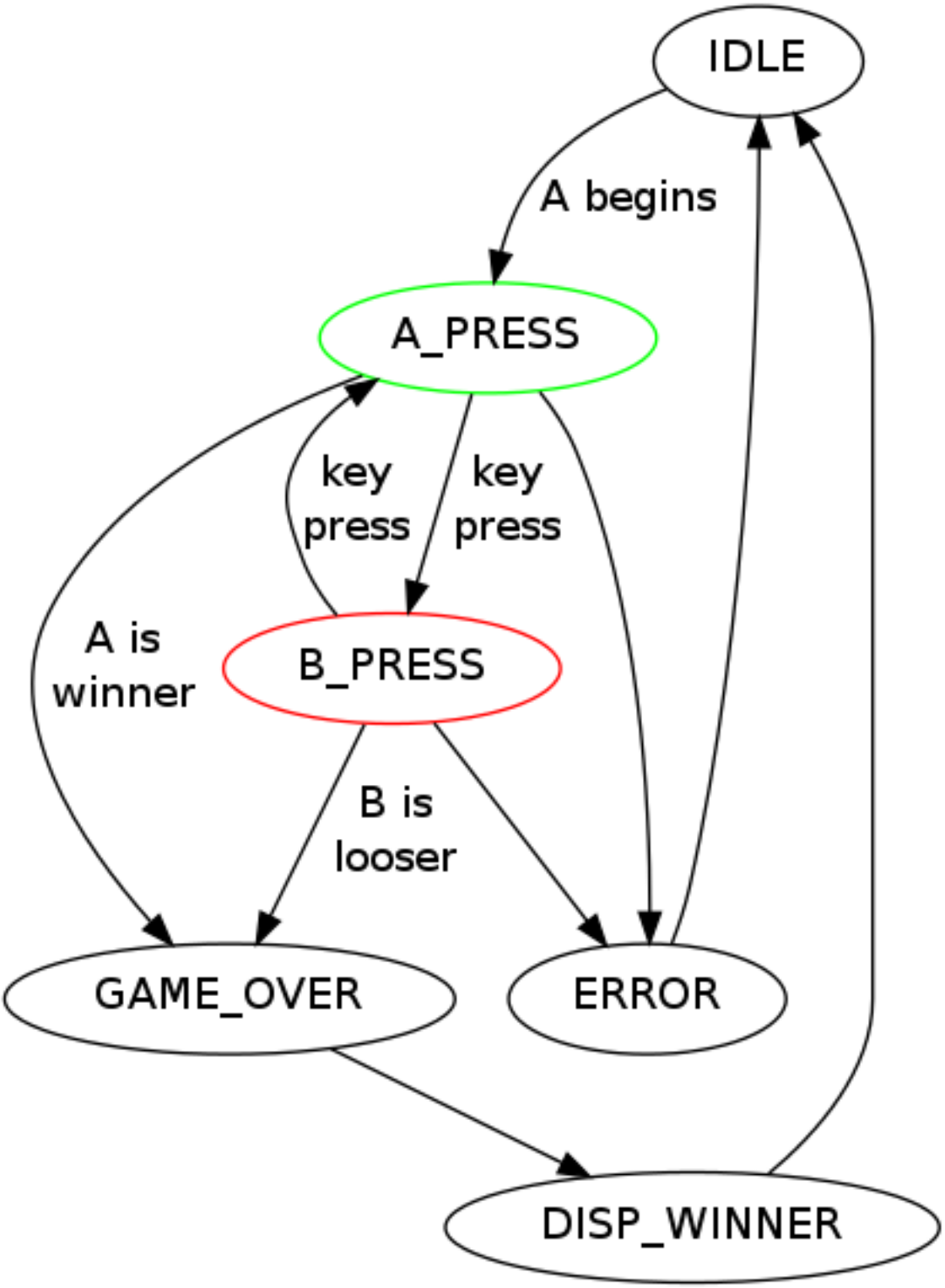
```
/* === includes ============================================================ */
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <util/delay.h>

/* uracoli libraries */
#include <board.h>
#include <transceiver.h>
#include <radio.h>

#include "spiel.h"

/* === macros ============================================================== */
#define KEY_NONE (255)

/* === globals ============================================================= */

static uint8_t RxFrame[MAX_FRAME_SIZE];
static volatile bool TxInProgress;
static volatile uint8_t RxEvent;
static uint16_t DestinationAddress;

/* === functions =========================================================== */

/* === radio init ========== */
void xxo_radio_init(node_config_t *ncfg)
{

    if (0 != get_node_config_eeprom(ncfg, 0))
    {
        /* Standardwerte setzen, fall ungültige Daten im EEPROM */
        ncfg->short_addr = SHORTADDR;
        ncfg->pan_id = PANID;
        ncfg->channel = CHANNEL;
        DestinationAddress = SHORTADDR;
    }
    else
    {
        DestinationAddress = ncfg->short_addr;
        DestinationAddress += (ncfg->short_addr & 1) ? +1 : -1;
    }

    radio_init(RxFrame, sizeof(RxFrame));
    radio_set_param(RP_CHANNEL(ncfg->channel));
    radio_set_param(RP_SHORTADDR(ncfg->short_addr));
    radio_set_param(RP_PANID(ncfg->pan_id));
    radio_set_param(RP_IDLESTATE(STATE_RXAUTO));
    radio_set_param(RP_TXPWR(-4));

    radio_set_state(STATE_RXAUTO);
    RxEvent = KEY_NONE;
}

/* === transmit functions === */
void xxo_send(uint8_t event, node_config_t *ncfg)
{
```

```
    static uint8_t seqno = 0;
    xxo_frame_t txbuf;
    /* fill frame information */
    txbuf.fcf = 0x8861;
    txbuf.seq = seqno++;
    txbuf.panid = ncfg->pan_id;
    txbuf.dst = DestinationAddress;
    txbuf.src = ncfg->short_addr;
    radio_set_state(STATE_TXAUTO);

    /* fill payload */
    txbuf.event = event;

    /* send frame */
    TxInProgress = true;

    radio_send_frame(sizeof(xxo_frame_t), (uint8_t*)&txbuf, 0);

    set_sleep_mode(SLEEP_MODE_IDLE);
    while (TxInProgress)
    {
        sleep_mode();
    }
}

void usr_radio_tx_done(radio_tx_done_t status)
{

    TxInProgress = false;
    if (status == TX_NO_ACK)
    {
        xxo_set_error(NO_PEER_RESPONSE);
    }
}

/* === receive functions === */
uint8_t * usr_radio_receive_frame(uint8_t len, uint8_t *frm, uint8_t lqi, int8_t ed,  ↩
    uint8_t crc)
{
    xxo_frame_t *pframe;
    pframe = (xxo_frame_t *)frm;
    if (RxEvent == KEY_NONE)
    {
        RxEvent = pframe->event;
    }
    return frm;
}

uint8_t xxo_radio_get_event(void)
{
uint8_t ret;

    cli();
    ret = RxEvent;
    RxEvent = KEY_NONE;
    sei();
    return ret;
}

void xxo_turn_off_radio(void)
{
    radio_set_state(STATE_SLEEP);
```

```
}
```

7.2 Zug um Zug

Im nächsten Schritt sorgt ein Zustandsautomat für einen geregelten Spielablauf.



Die Implementierung der eigentlichen Statemachine findet man in der Funktion main() des Moduls spiel.c.

7.3 Gewinnauswertung

Nach jedem Spielzug wird die aktuelle Gewinnsituation geprüft. Dazu wird geprüft ob es eine Zeile, Spalte oder Diagonale gibt, die mit der gleichen Farbe besetzt sind. Die möglichen Gewinnkombinationen sind in einer Tabelle im Modul leds.c gespeichert und können so einfach in einer Schleife abgeprüft werden. Die 9. Möglichkeit ist ein "unentschieden", das sich ergibt, wenn alle Felder besetzt sind aber keine der vorigen Gewinn Möglichkeiten aufgetreten ist.

```
static uint8_t WinningPattern[8][3] = {
        {0, 1, 2},
        {3, 4, 5},
        {6, 7, 8},
        {0, 3, 6},
        {1, 4, 7},
        {2, 5, 8},
        {0, 4, 8},
        {2, 4, 6}
};

uint8_t led_check_winner(void)
{
    uint8_t i;
    uint8_t *pwin;
    uint8_t winner_color = OFF;
    for (i = 0; i < 8; i++)
    {
        pwin = WinningPattern[i];
        if ((LedState[pwin[0]] == GREEN) &&
            (LedState[pwin[1]] == GREEN) &&
            (LedState[pwin[2]] == GREEN))
        {
            winner_color = GREEN;
            break;
        }
        if ((LedState[pwin[0]] == RED) &&
            (LedState[pwin[1]] == RED) &&
            (LedState[pwin[2]] == RED))
        {
            winner_color = RED;
            break;
        }
    }

    if (winner_color != OFF)
    {
        led_flash_pattern(5, pwin[0], pwin[1], pwin[2], winner_color);
    }
    else
    {
        for (i = 0; i< 9; i++)
        {
            if (LedState[i] == OFF)
            {
                break;
            }
        }
        if (i == 9)
        {
            winner_color = 4;
        }
    }
    return winner_color;
}
```

Die Gewinnprüfung erfolgt bei beiden Spielpartnern individuell und es erfolgt kein Datenaustausch am Spielende.

7.4 Noch mehr Strom sparen

Den Löwenanteil des Stromverbrauchs macht nicht der Controller, sondern der Transceiver aus. Oft unterschätzt wird dabei der Energieverbrauch im reinen Bereitschaftsmodus, wenn der Empfänger auf Daten wartet. Einerseits werkeln im UHF-Frontend eine Reihe von Schaltungsteilen mit Frequenzen im Gigahertz-Bereich, andererseits ist in diesem Zeitraum der digitale Demodulator die ganze Zeit damit beschäftigt, aus dem Rauschen des UHF-Teils per Korrelation seine Präambel-Sequenz zu erkennen. Der sich dabei ergebende Stromverbrauch liegt im Bereich von 18 mA.

Daher lohnt es sich, während der Zeit, in der keine Empfangsbereitschaft notwendig ist (Spielstatus *IDLE*), den Transceiver selbst schlafen zu legen. Dies erledigt der Aufruf von `radio_set_state(STATE_SLEEP);`, der als `xxo_turn_off_radio()` in die oberen Ebenen der Spiel-Logik abstrahiert wird. Der Schlafzustand des Transceivers wird dabei mit dem nächsten gesendeten Datenpaket wieder verlassen.

Diese Maßnahme reduziert den Stromverbrauch von 14 mA auf ca. 1,2 mA.

Für ein Gerät ohne Ausschalter ist dieser Wert natürlich immer noch recht hoch. Für übliche LR03-Zellen kann von einer Kapazität im Bereich um die 1000 mAh ausgegangen werden. Bei einem Ruhestrom von 1,2 mA wären die Batterien folglich nach ca. 1 Monat leer.

Zumindest während der inaktiven Phase (kein Spiel läuft) muss der Energieverbrauch daher noch weiter reduziert werden. Dies erreicht man, indem man einen aggressiveren *sleep mode* benutzt, bei dem alle Taktleitungen, die nicht benötigt werden, abgeschaltet werden. CMOS-Schaltkreise benötigen im Wesentlichen nur dann Strom, wenn sie getaktet werden, weil dann intern immer wieder Gate- und Leitungskapazitäten umgeladen werden müssen. Der rein statische Stromverbrauch (kein einziger Takt läuft mehr) eines ATmega128RFA1 bei Zimmertemperatur liegt bei wenigen 100 nA.

Der Modus mit der geringsten Stromaufnahme ist der *power-down*-Modus. Dabei ist der Hauptoszillator gestoppt, sodass weder CPU noch Peripheriegeräte getaktet werden. Aus diesem Modus führt (neben einem Reset) nur ein Interrupt, der selbst keinen der normalen Peripherietakte benötigt. Die Möglichkeit eines sogenannten Extern- oder Pin-Change-Interrupts ist im Tic-Tac-Toe-Board jedoch aufgrund der benutzten Sensor-Technologie für die Tasten nicht gegeben.

Daher nutzen wir den sogenannten *watchdog*. Dieser ist eigentlich zur Funktionsüberwachung der Firmware gedacht, indem die aktive Firmware ständig den Wachhund wieder zurücksetzt. Fährt die Software sich aus irgendeinem Grund fest, dann ist irgendwann die dem Wachhund einprogrammierte Zeit abgelaufen, und er löst einen Reset des Controllers aus. Alternativ kann man ihn aber auch in einem Interruptmodus betreiben. Da der Wachhund einen eigenen Oszillator besitzt, ist dieser auch im *power-down*-Schlafzustand noch benutzbar. Bei Ablauf der programmierten Zeit erfolgt dann kein Reset, sondern ein normaler Interrupt. In der zugehörigen ISR wird kurz die Tastaturabfrage für die mittlere Taste aktiviert. Wenn keine solche Taste gedrückt erkannt worden ist, so legt der Controller sich sofort wieder schlafen. Im Ergebnis entsteht mit dieser Maßnahme ein mittlerer Stromverbrauch von ca. 50 μA, was einer Batterielebensdauer von etwa 2 Jahren entspricht. Dieser Wert ist schon eher annehmbar und genügt zur Demonstration dieses einfachen Spiels.

Wurde die mittlere Taste als gedrückt erkannt, so wird der Wachhund anschließend in den Reset-Modus umgeschaltet, was nach weiteren etwa 15 ms zu einem Reboot führt, sodass das Spiel von vorn beginnen kann.

Nachfolgend noch einige Oszillogramme während der Tiefschlafphase. Man erkennt das zyklische Aufwachen aller ca. 15 ms, während eines Zeitraums von ca. 500 μs wird die mittlere Taste abgefragt. Beim Wiedereinschalten per mittlerer Taste läuft dann alles wieder an; besonders markant ist dabei der Aufladestromimpuls für den analogen Spannungsregler des Transceivers.

current2.png

Wichtig für das Stromsparen sind noch folgende Dinge:

- keine Pins offen lassen (*floating*); entweder als Ausgang (vorzugsweise auf *low*) schalten, oder bei einem Eingang die *pullup*-Widerstände aktivieren; offene Eingänge fangen schnell Störungen ein, die dann zu Umschaltströmen in den Eingangsstufen führen
- das *on-chip-debugging* muss deaktiviert werden; die entsprechende Fuse (im *high fuse byte*) wurde für den Debugger aktiviert, sie muss im endgültigen Gerät deaktiviert werden (HF = 0x91 statt 0x11)
- ein angesteckter Programmierer/Debugger (AVR Dragon) entnimmt dem Gerät Strom (für die Pegelwandler im Debugger); bei Strommessungen sollte er also abgezogen werden

7.5 Fehlerbehandlung

Um das Spiel stabiler und robuster zu machen, sollten einige Fehlerfälle abgefangen werden. Dabei gibt es folgende Möglichkeiten für einen Spielabbruch, der infolge eines Fehlers auftreten kann:

- Es erfolgte keine Tasteneingabe innerhalb eines Timeout-Intervalls.
- Es wurde kein Rahmen vom Spielgegner innerhalb eines Timeout-Intervalls empfangen.
- Ein Daten-Rahmen konnte nicht gesendet werden.
- Ein Spiel-Abbruch-Rahmen wurde von der Gegenstelle empfangen.

In all diesen Fehlerfällen soll das Spiel abgebrochen und die Software in den Ausgangszustand versetzt werden.

Die Aufgabe die Fehlerbehandlung zu implementieren, wird an dieser Stelle an den Leser übertragen.

Zusammenfassung

> Im **Teil VII** des Workshops wurde die originale Spiel-Idee inklusive Zustandsautomat und Gewinnauswertung fertig implementiert. Eine robuste Methode zur Behandlung ist noch zu implementieren.

8 Ausblick

8.1 Erweiterungen

- Freies Peering, doppelte Tastendrücke
- verbessertes Stromsparen
- 3 Spieler Modus (orange als LED-Farbe darstellbar)

8.2 Aktualisierung des Config-Records

Auf den Boards sind Aufkleber mit den Seriennummern und den MAC-Adressen angebracht.

Wenn man einen neuen Config-Record in den EEPROM schreiben möchte, geht man wie folgt vor:

```
# Disable EESAVE-Fuse
$ avrdude -P usb -p atmega128rfa1 -c dragon_jtag -U hf:w:0x19:m

# Erase Chip (inkl. EEPROM)
$ avrdude -P usb -p atmega128rfa1 -c dragon_jtag -e

# Enable EESAVE-Fuse
$ avrdude -P usb -p atmega128rfa1 -c dragon_jtag -U hf:w:0x11:m

# Erzeuge Config-Record
$ python uracoli/wibo/nodeaddr.py \
        -Bxxo -O 0 \
        -a <SN> -A <MAC> -p <PAN>
        -c <CHANNEL> -o cfg.hex

# Schreibe den Config Record ins EEPROM
$ avrdude -P usb -p atmega128rfa1 -c dragon_jtag -U ee:w:cfg.hex:i

# Anzeigen des EEPROM-Inhaltes
$ avrdude -P usb -p atmega128rfa1 -c dragon_jtag -tF
avrdude> d ee 0 16
>>> d ee 0 16
0000  16 00 42 42 b4 21 17 ff  ff 25 04 00 11 00 00 e0  |..BB.!...%......|
```

Diese Prozedur ist für jedes Board einzeln, mit den jeweils aktuellen Parametern durchzuführen. Die Parameter des Scripts `nodeaddr.py` bedeuten dabei im einzelnen:

-B xxo	Tic-Tac-Toe Hardware
-O 0	ist der Adress-Offset 0 im EEPROM
-a <SN>	<SN> is die Seriennummer vom Aufkleber, die dezimal ohne führende 0 eingegeben wird. SN.0042 ⇒ -a 42
-A <MAC>	Bei der MAC-Adresse werden die ":" entfernt und ein "0x" vorangestellt. "00:04:25:ff:ff:17:21:B4" ⇒ -A 0x000425ffff1721B4
-p <PAN>	Dieser Parameter wird direkt als Hexadezimal-Zahl übernommen, -p 0x4242
-c <CHANNEL>	Der IEEE-802.15.4-Kanal ist eine Dezimal-Zahl zwischen 11 und 26.
-o cfg.hex	Name des Intel-HEX-Files mit dem Config-Record.