Mobi-Robot
Hardware Abstraction Layer

Daniel Binggeli, FHNW 1.04.2015

Neu mit Version 01_04_15

- Komfortfunktionen für I2C. → Folie 6
- Diese Funktionen ermöglichen die interruptgesteuerte Kommunikation mit allen gängigen I2C Geräten.
- Die Funktionen **TWI_WriteCmd** sowie **TWI_ReadCombined** lösen die meisten Aufgaben.
- Als Beispiel ist die Ansteuerung des Colour/Gesture-Sensors SparkFun_APDS-9960 angefügt

LCD-Display

void lcd_init (uint8_t dispAttr)

Initialize display and select type of cursor.

void lcd_clrscr (void)

Clear display and set cursor to home position.

void lcd_gotoxy (uint8_t x, uint8_t y)

Set cursor to specified position.

void lcd_putc (char c)

Display character at current cursor position.

void lcd_puts (const char *s)

Display string.

void lcd_puts_p (const char *progmem_s)

Display string from program memory.

void lcd_command (uint8_t cmd)

Send LCD controller instruction command.

void lcd_data (uint8_t data)

Send data byte to LCD controller.

Tasten

void key_init (void)

Initialize keys.

uint8_t key_get (void)

Return code of a pressed key from buffer.

(0 if none).

void key_clear (void)

Clear key buffer.

uint8_t key_check (void)

Return true if a key was pressed.

I2C - Master

void TWI_Master_Initialise(void);

Initialisiert den I2C

unsigned char TWI_Transceiver_Busy(void);

Checkt ob die I2C-Logik beschäftigt ist

unsigned char TWI_Get_State_Info(void);

Liest Status-Informationen von der I2C-Logik

void TWI_Start_Transceiver_With_Data(unsigned char * , unsigned char);

Startet eine Übermittlung und übergibt zu sendende Daten

void TWI_Start_Transceiver(void);

Startet eine Übermittlung mit vorher schon gelieferten Daten

unsigned char TWI_Get_Data_From_Transceiver(unsigned char *, unsigned char);

Liest die empfangenen Daten aus dem Empfangspuffer

I2C – Beispiel mit PCA9555 Porterweiterung

```
TWI_Master_Initialise();
if(!TWI_Transceiver_Busy()){
// when ready, send direction register write command
     twibuf.numo = 4; //compose message
     twibuf.numi = 0;
     twibuf.out[0] = PCA9555+I2C_WRITE;
     wibuf.out[1] = PCA_C0;
     twibuf.out[2] = pca.dir.b[0];
     twibuf.out[3] = pca.dir.b[1];
     TWI_Start_Transceiver_With_Data(twibuf.out, twibuf.numo);
}
if(!TWI_Transceiver_Busy()){
// when ready, send output register write command
     twibuf.numo = 4; // compose message
     twibuf.numi = 0;
     twibuf.out[0] = PCA9555+I2C_WRITE;
     twibuf.out[1] = PCA_O0;
     twibuf.out[2] = pca.out.b[0];
     twibuf.out[3] = pca.out.b[1];
     TWI_Start_Transceiver_With_Data(twibuf.out, twibuf.numo);
}
```

```
if(!TWI_Transceiver_Busy()){
// when ready, send input register read command
     // compose message for the input register read procedure
     twibuf.numo = 2;  // two bytes to be sent
     twibuf.numi = 2;  // this is the number of requested bytes from
the slave
     twibuf.out[0] = PCA9555+I2C_WRITE;
     twibuf.out[1] = PCA_I0;
     TWI_Start_Transceiver_With_Data(twibuf.out, twibuf.numo);
}
if(!TWI_Transceiver_Busy()){
// when ready, start reception of input register data
     TWI_Get_Data_From_Transceiver(twibuf.in, twibuf.numi);
     // read the received bytes
     pca.inp.b[0]=twibuf.in[0];
     pca.inp.b[1]=twibuf.in[1];
}
```

I2C – Master II, Komfort-Funktionen

void TWI_Master_Initialise(void);

Initialisiert den I2C.

unsigned char TWI_Transceiver_Busy(void);

Checkt ob die I2C-Logik beschäftigt ist.

uint8_t TWI_WriteCmd(uint8_t addr, uint8_t cmd, uint8_t len, uint8_t *pmsg);

Schreibt den Befehl **cmd** mit der Mitteilung **pmsg** von **len** Bytes an das I2C Gerät **addr.**

uint8_t TWI_Write(uint8_t addr, uint8_t len, uint8_t *pmsg);

Schreibt die Mitteilung **pmsg** von **len** Bytes an das I2C Gerät **addr.**

getestet

uint8_t TWI_Read(uint8_t addr, uint8_t size);

Liest **size** Byte Daten vom I2C Gerät **addr.**

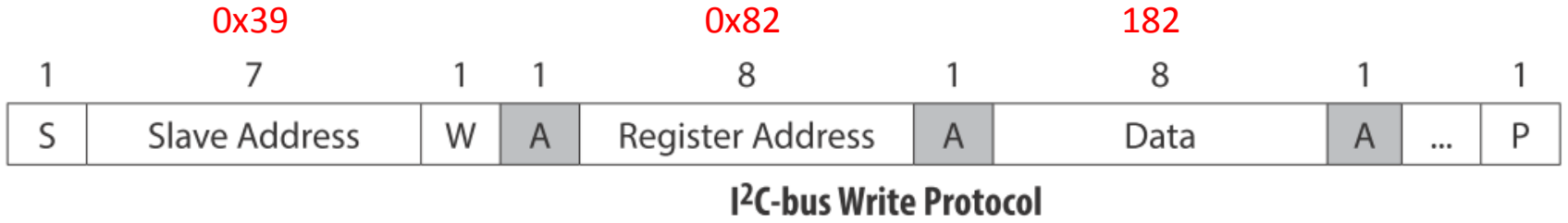
uint8_t TWI_ReadCombined(uint8_t addr, uint8_t len, uint8_t *pmsg, uint8_t size);

Schreibt die Mitteilung **pmsg** von **len** Bytes und liest anschliessend **size** Byte Daten vom I2C Gerät **addr.**

unsigned char TWI_Get_Data_From_Transceiver(uint8_t *val, uint8_t size);

Liest die empfangenen Daten aus dem Empfangspuffer.

I2C – Beispiel mit SparkFun_APDS-9960



I²C-bus Write Protocol

A	Acknowledge (0)
N	Not Acknowledged (1)
P	Stop Condition
R	Read (1)
S	Start Condition
Sr	Repeated Start Condition
W	Write (0)
...	Continuation of protocol
	Master-to-Slave
	Slave-to-Master

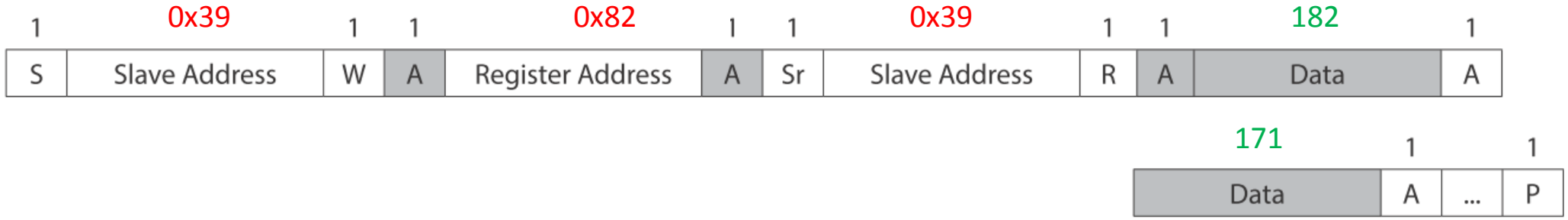
Dieser Befehl schreibt den Wert 182 in das Register ATIME (0x82), was eine ADC-Integrationszeit von 200ms für die ALS/Color-Engine einstellt.
Der APDS9960 hat immer die Adresse 0x39.

```
uint8_t msg[1] = {182};
if( TWI_WriteCmd( 0x39, 0x82, 1, msg ) > 0 )
{
        // senden war erfolgreich ...
}
```

Slave Address *Register Address* *Number of Bytes* *Message-Buffer*

TWI_WriteCmd kann auch für kontinuierliches Schreiben ganzer Blöcke verwendet werden. Diese werden im APDS9960 in auf-einanderfolgenden Register-adressen gespeichert.

I2C – Beispiel mit SparkFun_APDS-9960



I²C-bus Read Protocol - Combined Format

Dieser Befehl liest den Wert 182 vom das Register ATIME (0x82) sowie 171 aus dem Register WTIME (0x83).
ATIME ist 182, also 200ms
WTIME ist 171, also 236ms (siehe Datenblatt)

TWI_ReadCombined kann auch für kontinuierliches Lesen ganzer Blöcke verwendet werden. Diese belegen im APDS9960 aufeinanderfolgende Registeradressen.

```
uint8_t msg[1] = {0x82};
uint8_t val[2] = {0,0};
if( TWI_ReadCombined( 0x39, 1, msg, 2 ) > 0 )
{
  TWI_Get_Data_From_Transceiver(val, 2);

// val ist jetzt [182;171]
```

Slave Address *Number of Bytes sent* *Register Address* *Number of Bytes received*

Data-Buffer *Number of Bytes received*

Servos / Motors

uint8-t **servo_init(void)**

initialise servo resources

uint8_t **servo_set(uint8_t servo, uint16_t pos)**

set servo (0..9) to pos (0..1000)

uint8-t **motor_init(void)**

initialise motor resources

uint8_t **motor_set(uint8_t motor, int16_t power)**

set motor (1..4) to power (-1000..1000)

Encoders

uint8-t **enc_init(void)**

initialise encoder resources

uint8_t **enc_set(uint8_t enc, int32_t pos)**

set encoder (0..1) to

pos (-2^31..2^31)

int16_t **enc_get(uint8_t enc)**

get encoder (0..1) to

returns pos (-32768..32767)

AD-Kanäle

uint16_t adc_read(uint6_t adc_input)

konvertiere einen AD_Kanal

Gültig sind 0..3 und 8..15

uint8-t adc_init(uint8_t adc_reference)

initialisiere ADC auf die gegebene Referenzspannung

Schalte bei allen mit ANALOG_xy

belegten Kanälen die Digitalen Signale ab.

Nicht getestet

Sensors

uint8-t sensor_init(void)

initialise sensor resources

uint8_t digi_sensor_get(uint8_t sensor)

get sensor (0..3;8..15)

returns val (0..1)

uint16_t ana_sensor_get(uint8_t sensor)

get sensor (0..3;8..15)

returns val (0..1023), after <80µs

uint16_t usonic_sensor_read(uint8_t sensor)

trigger and read sensor (0..1)

returns delay (0..32767),

after 0.5 to 3ms

Nicht implementiert

Digital I/O Ports

Makro ddr_set erlaubt sorgloses Schreiben auf DDRx . Alle vom HAL benötigten Bits bleiben unverändert

void ddr_set(P, pattern);

Makro port_set erlaubt sorgloses Schreiben auf PORTx. Alle vom HAL benötigten Bits bleiben unverändert

void port_set(P, pattern);

Makro port_get erlaubt sorgloses Lesen von PINx. Alle vom HAL benötigt en Bits lesen 0.

uint8_t port_get(P);

PIN-change call-back routines:

void pinchange_Pxy(void)

x: B, E, J, K

y: 0..7

(nicht alle Kombinationen sind gültig)

Mit port_init() können einzelne Pins auf Pegelwechsel «scharf» gemacht werden. Ein Pegelwechsel am Pin ruft dann die entsprechende Routine auf.

An der vordefinierten globalen Variable slope kann die letzte Flanke abgefragt werden.

Timing

uint8-t system_timer_init(void)

initialise system timer, tick and time

uint32_t time_get(void)

get system time (0..2^{32}) [ms]

uint8_t tick_get(void)

get system tick (0..1)

TRUE if tick interval elapsed.

Every call of tick_get() resets the tick flag

System-tick call-back routine:

void system_tick(uint32-_t time)

Der System-Timer ruft in einem konfigurierbaren Intervall (4..1024 ms) diese Routine auf.

➔SYSTEM_INTERVAL

Hauptprogramm (à la ARDUINO)

```
void main (void)
{
      sei();

      setup();
      for(;;)
            loop();
}
```

```
void setup(void)
{
      //alle init-Routinen hier rein
}

void loop(void)
{
      // hier alle «action»
}
```

Call-back Routinen I

```
struct CBRoutine {
        uint8_t valid;
        void (*ptr_cbr)(uint8_t p);
} cbr[NUM_CBR] =
{
        false; NULL; //PB0,PCINT0
        true; pinchange_PB1;
        true; pinchange_PB2;
        false; NULL; //PB3,PCINT3
        …                       …
        false; NULL; //PK7,PCINT23
}
```

- wenn cbr[x].valid `true` ist, so muss bei cbr[x].ptr_cbr ein Funktionsname eingetragen sein.
- wenn cbr[x].valid `true` ist, heisst das, dass der HAL die eingetragene Funktion aufruft.
- wenn cbr[x]valid `false` ist, so kann statt ein Name `NULL` eingetragen werden.
- In der Tabelle cbr hat es Platz für 24 Einträge, nämlich alle 24 Pin-Change-Interrupts (s. rechts)

PCINT0	PB0
PCINT1	PB1
PCINT2	PB2
PCINT3	PB3
PCINT4	PB4
PCINT5	PB5
PCINT6	PB6
PCINT7	PB7
PCINT8	PE0
PCINT9	PJ0
PCINT10	PJ1
PCINT11	PJ2
PCINT12	PJ3
PCINT13	PJ4
PCINT14	PJ5
PCINT15	PJ6
PCINT16	PK0
PCINT17	PK1
PCINT18	PK2
PCINT19	PK3
PCINT20	PK4
PCINT21	PK5
PCINT22	PK6
PCINT23	PK7

Call-back Routinen II, Beispiel

```
volatile uint16_t timer = 0;


void system_tick(uint32_t time)
{
      timer++;
}
```

- alle CB-Routinen haben Argument und Returnwert void
- alle in den CB-Routinen verwendeten Variablen müssen `volatile` sein
- die CB-Routinen sollen kurz sein, keine Loops! Während der Ausführung sind die Interrupts gesperrt .
- die CB-Routinen dürfen nur vom HAL aufgerufen werden.
- Im Anwenderprogramm darf nirgendwo eine CB-Routine aufgerufen werden.
- Die Funktionsnamen der CB-Routinen sind in HAL_MobiRob.h vordefiniert
- Nur der Funktionsname ist eine Referenz und kann an einen Funktionszeiger übergeben werden.

Mehrere Aufgaben ausführen

```
while(linie_folgen){
      if(lage_rechts){
        motors(slow,fast);
      }
      else if(lage_links){
        motors(fast,slow);
      }
      else{
        motors(fast,fast);
      }
}
```

```
while(suchen){
      if(arm_rechts){
        step = -8;
      }
      else if(arm_links){
        step = 8;
      }
      pos = pos + step;
      servo_set(pos);
}
```

Mehrere Aufgaben ausführen

```
while(1){
  switch(linie_SM){
    case LINIE_RECHTS:
         motors(slow,fast);
         break;
    case LINIE_LINKS:
         motors(fast,slow);
         break;
    case LINIE_MITTE:
         motors(fast,fast);
         break;
    default: // OFF
         break;
  }
```

```
  switch (arm_SM){
    case ARM_RECHTS:
         step = -8;
         break;
    case ARM_LINKS:
         step = 8;
         break;
    case ARM_AKTIV:
         pos = pos + step;
         servo_set(pos);
         break;
    default: // OFF
         break;
  }
}
```

Mehrere Aufgaben ausführen Vorgehen:

- While loops auftrennen
- Pro Aufgabe eine SM
- Bei mehrschrittigen Aufgaben (Sequenzen) pro Schritt ein case-statement
- Alle SM in einem grossen loop nacheinander anordnen

- Die aktiven Zustände bestimmen sich meistens aus
- A) Sensorsignalen
- B) dem vorher aktiven Zustand

(SM = State-Machine)

Was ist eine State-Machine

- Software-Konstrukt zur Behandlung von seqentiellen Vorgängen

Komponenten:

- Zustandsvariable
- Switch-case Statement
- Berechnung der Aktor-Signale in den Schritten (= case Stetements)
- Auswertung der Sensoren (=Weiterschaltbedingungen) in den einzelnen Schritten oder nachher, separat