Hochschule Aalen

Aufgabe 7.1

Schreiben Sie ein einfaches Programm, welches über die serielle Schnittstelle RS232 Daten an einen angeschlossenen PC sendet.

- Nach dem Start des Mikrocontrollerprogrammes soll die Nachtricht "Hello World\n" verschickt werden.
- Im Sekundenrythmus soll, als eine Art von Heartbeat, die aktuelle Betriebszeit in Sekunden verschickt werden.
- Übertragungsparameter: 9600 Baud, 8 Datenbits, no parity, 1 Stopbit

- Implementieren Sie den Datenversand mittels Interrupt Service Routinen (ISR).
- Achten Sie bei der Implementierung z.B. beim Aufruf obiger Funktionen auf kritsche Resourcen und deren Zugriffsschutz.

Aufgabe 7.2:

- Verwenden Sie das Programm von Aufgabe 4 zur Prozessorauslastungbestimmung und versenden Sie diese Zahl in % nach jeder Messung (im 10 s Raster).
- Versehen Sie dafür auch die diversen neuen Tasks mit dem Statussignal auf dem Expansionsport
    - Beobachten Sie dieses Signal auch mit dem Oszilloskop

Aufgabe 7.3:

- Die 4 LEDs welche bei der Aufgabe 4 blinken, sollen mit Befehlen welche vom PC Terminal aus geschickt werden, zusätzlich gesteuert werden.
- Implementieren Sie analog zur Aufgabe 1 das Empfangen von seriellen Daten.
- Legen Sie dazu sinvolle Datenstrukturen (speicher) und Zugriffs- und Auswertefunktionen an.
  Siehe Aufgabe 1. Hier z.B. zusätzlich eine **int16_t getcUART(char**

***c)** Funktion im Zusammenspiel mit RX FIFO Funktionen oder auch höherintegriert z.B. int16_t **getNextCommand(char *command_string)**

- Kommandos für die LED Steuerung:
    - Jeder Befehl besteht aus einem Kommando (1 Buchstabe), der LED Nr, ev. zusätzliche Daten und einem abschließenden New Line "\n"
    - B0\n -> LED 0 blinkt mit der aktuell festgelegten Zeit (vom Tastertask oder Kommando)
    - R1\n -> LED 1 ist aus (reset)
    - S2\n -> LED 2 ist an (set)
- Kommando zur Blinkperiodenveränderung:
    - Erweitern Sie das Protokoll um einen Befehl um die Blinkperiode in ms für eine angegebene LED einstellen zu können.
    - Die Daten werden dezimal nach einem Unterstrich angegeben. Der Wertebereich ist 2-9999, dies entspreicht 2-9999 ms.
        - Ungültige Daten bzw. wenn die Zahl außerhalb dem Werteberich ist, wird der Befehl ignoriert.
    - T0_100\n -> LED 0, Blinkperiode 100 ms
    - T2_1000\n -> LED 2, Blinkperiode 1000 ms
- Quittierung:
    - Quittieren Sie jeden Befehl mit "OK\n" oder "Error\n"

Aufgabe 7.1

```
1.  void initUART(){
2.      U1MODEbits.STSEL = 0; // 1-Stop bit
3.      U1MODEbits.PDSEL = 0; // No Parity, 8-Data bits
4.      U1MODEbits.ABAUD = 0; // Auto-Baud disabled
5.      U1MODEbits.UEN = 0;
6.      U1MODEbits.LPBACK = 0;
7.      U1MODEbits.RXINV = 0;
8.      //U1MODEbits.ALTIO = 0;
9.
10.     U1MODEbits.URXINV = 0;
11.     U1MODEbits.RTSMD = 0;
12.
13.     U1MODEbits.BRGH = 0; // Standard-Speed mode
14.     U1BRG = BRGVAL; // Baud Rate setting for 9600
15.
16.     U1STAbits.UTXISEL0 = 0; // Interrupt after one TX character is transmitted
17.     U1STAbits.UTXISEL1 = 0;
18.     U1STAbits.UTXBRK = 0;
19.     U1STAbits.ADDEN = 0;
20.     U1STAbits.UTXINV = 0;
21.     U1STAbits.URXISEL = 0;
22.     U1STA = U1STA | 0b0001000000000000;
23.     //_URXEN = 1;
24.
25.     _U1RXIE = 1; // Enable UART RX interrupt
26.
27.     U1MODEbits.UARTEN = 1; // Enable UART
28.     //delay_ms(2);
29.     U1STAbits.UTXEN = 1; // Enable UART TX
30.
31.     /* Wait at least 105 microseconds (1/9600) before sending first char */
32.     delay_ms(2);
33.     _U1TXIE = 1; // Enable UART TX interrupt
34.
35. }
```

Um die Kommunikation über die RS232 (UART)-Schnittstelle zu ermöglichen ist es im ersten Schritt notwendig die benötigten Register mithilfe des Datenblatts zu initialisieren. Als Basis wurden gewählt: *non-parity, 1 Stopbit, 8 Datenbits und 9600 Baud.*

Um nun im weiteren Verlauf Daten über den RS232 D-Sub Connector zu senden, wird beim Beschreiben des FIFO-Buffer nach jedem einzelnen gesendeten ASCII-Charakter eine ISR aufgerufen, welche wie folgt lautet:

```
1.  void __attribute__((__interrupt__)) _U1TXInterrupt(void)
2.  {
3.      _U1TXIF = 0; // Clear TX Interrupt flag
4.
5.      getcFIFO_TX(&U1TXREG);
6.  }
```

Wie gewohnt wird zuerst das entsprechende Interrupt-Flag zurückgesetzt und dann die getcFIFO_TX() Funktion aufgerufen. Da diese bisher noch nicht erläutert wurde, folgt hier die Implementierung:

```
typedef struct {
  uint8_t data[BUFFER_SIZE];
  uint8_t read; // zeigt auf das Feld mit dem ältesten Inhalt
  uint8_t write; // zeigt immer auf leeres Feld
}Buffer;

Buffer FIFO = {{}, 0, 0};
Buffer FIFO_RX = {{}, 0, 0};
```

In obigem Codeausschnitt befinden sich die Deklarationen des FIFO-Speichers. Dabei stellt der „data“-String den eigentlichen Datenspeicher dar und besitzt eine Größe von (in unserem Fall) 128 Stellen.

```
int16_t getcFIFO_TX(char *c)
{
  _LATF0 = 1;
  if (FIFO.read == FIFO.write)
    return BUFFER_FAIL;

  *c = FIFO.data[FIFO.read];

  FIFO.read++;
  if (FIFO.read >= BUFFER_SIZE)
    FIFO.read = 0;

  return BUFFER_SUCCESS;
}
```

Hier findet sich nun auch die getcFIFO-Funktion, durch welche sich die Werte, die im data-string gespeichert sind Stelle für Stelle auslesen lassen. Sobald das FIFO-Buffer leer ist (gekennzeichnet durch read- und write-zeiger auf gleiche Stelle), wird im Rückgabewert ein BUFFER_FAIL zurückgegeben.

```
int16_t putcFIFO_TX(char c)
{
  //if (buffer.write >= BUFFER_SIZE)
  //  buffer.write = 0; // erhöht sicherheit
  _LATF0 = 1;
  if ( ( FIFO.write + 1 == FIFO.read ) ||
       ( FIFO.read == 0 && FIFO.write + 1 == BUFFER_SIZE ) )
    return BUFFER_FAIL; // voll

  FIFO.data[FIFO.write] = c;

  FIFO.write++;
  if (FIFO.write >= BUFFER_SIZE)
    FIFO.write = 0;

  return BUFFER_SUCCESS;
}
```

Die putcFIFO-Funktion ist zu verwenden, um Daten in den FIFO-Speicher zu schreiben. Hierbei wird ebenfalls Stelle für Stelle vorgegangen und jeder Buchstabe (jede Zahl) einzeln in ein Element geschrieben. Hier gibt das Buffer den Rückgabewert BUFFER_FAIL aus, sobald alle Stellen beschrieben sind und beide Zeiger, wie oben, auf der gleichen Stelle stehen.

Um im weiteren Verlauf einen einzelnen Char über das UART zu senden, wurde die putc_UART()-Funktion geschrieben:

```
1.  int16_t putcUART(char c){
2.      _LATF0 = 1;
3.      _GIE = 0; // Interrupts ausschalten
4.      int16_t erfolg = putcFIFO_TX(c);
5.      _GIE = 1;
6.      return erfolg;
7.
8.
9.  }
```

LATF0 wird hier auf HIGH gesetzt, damit die Prozessorauslastung später auch innerhalb dieser Funktionen mitzählt. Danach wird im Zuge des Ressourcenschutzes ein Interrupt-Disable ausgeführt, um den entsprechenden char sicher in das FIFO-Register zu speichern. Die Variable „erfolg“ bekommt dann den Rückgabewert der putc_FIFO Funktion.

Um nicht nur einen einzelnen Char senden zu können, wurde eine putsUART()-Funktion geschrieben, welche einen ganzen String über die RS232-Schnittstelle sendet.

```
1.  int16_t putsUART(const char *str) {
2.      _LATF0 = 1;
3.      uint16_t i;
4.      uint16_t length = strlen(str);
5.
6.      _GIE = 0;   //Global Interrupt disable
7.      for(i = 0; i < length; i++) {
8.          //uint16_t ret = putcFIFO_TX(str[i]);
9.          if(! putcFIFO_TX(str[i]))
10.             break;
11.     }
12.     _GIE = 1;
13.     int16_t erfolg = -i;
14.     if(erfolg == -length)
15.         erfolg *= -1;
16.     _U1TXIF = 1; //Interuppt Routine Starten um FIFO-Inhalt zu senden
17.     return erfolg;
18. }
```

Hier werden die Bestandteile des Strings Elemente für Element in den FIFO-Speicher geschrieben, um im späteren Verlauf vom UART versendet zu werden. Die Abfrage in Zeile 9 dient zum einen dazu, die Elemente einzeln in das FIFO zu schreiben, aber gleichzeitig auch dazu, die Rückgabe dieser Funktion zu prüfen. Wenn der FIFO-Speicher voll sein sollte, bevor der komplette String gespeichert ist, wird die For-Schleife verlassen und ab Zeile 13 die Anzahl der gespeicherten Zeichen mit einem negativen Vorzeichen als Rückgabeparameter verwendet.

Um die Betriebszeit auszugeben wurde die Timer1 ISR folgendermaßen erweitert:

```
1.  void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void)
2.      {
3.          _T1IF = 0; //Clear Timer1 interrupt flag
4.          u32_uptime_seconds++;
5.          char str[16];
6.          uint32_t test = getBetriebszeit();
7.          sprintf(str,"%lu",test);
8.          putsUART("Betriebszeit: ");
9.          putsUART(str);
10.         putsUART("\n");
11.         //u32_inactive_seconds++;
12.
13.     }
```

Aufgabe 7.2

```
1.  void measureProcesstime(){
2.      static uint16_t time = 0;
3.      time++;
4.      if (time >= MESSZEIT_MS){
5.          float Auslastung;
6.          uint16_t lsw = TMR2;
7.          uint16_t msw = TMR3HLD;
8.
9.          float Auslastung1 = (float)lsw * 10;
10.         float Auslastung2 = (float)msw *65535*10;
11.         Auslastung = (Auslastung1 + Auslastung2) / (FCY);
12.         time = 0;
13.         TMR3HLD=0;
14.         TMR3 = 0; // Clear 32-bit Timer (msw)
15.         TMR2 = 0;
16.         char str[16];
17.         putsUART("Auslastung in %: ");
18.         sprintf(str,"%f",Auslastung);
19.         putsUART(str);
20.         putsUART("\n");
21.     }
22. }
```

Diese Funktion wurde von vorherigen Aufgaben übernommen und um die Ausgabe per UART erweitert. Dafür wurde die Auslastung mit Hilfe der Funktion sprintf in ein String umgewandelt.

Zusätzlich wurde in jeder Funktion, welche nun zusätzlich Rechenzeit in Anspruch nimmt _LATF0=1 gesetzt, um den Gated-Timer zu starten.

Die Auslastung liegt hier bei einer Frequenz von 50MHz bei ca. 1,2% und steigt leicht an, wenn viele Befehle empfangen oder gesendet werden.

Aufgabe 7.3

In der initUART-Funktion wurden die jeweiligen Register so beschrieben, dass ein Interrupt ausgeführt wird, sobald ein Zeichen empfangen worden ist. Die Service Routine ist hier zu sehen:

```
1.  void __attribute__((__interrupt__)) _U1RXInterrupt(void)
2.  {
3.      _U1RXIF = 0;
4.      static int aufrufe = 0;
5.      if (aufrufe != 0){
6.          putcFIFO_RX(U1RXREG);
7.          getNextCommand();
8.      }
9.      else{
10.         char dummy= U1RXREG;
11.     }
12.     aufrufe = 1;
13. }
```

Da der Interrupt auch schon bei der Initialisierung aufgerufen wird und zu dieser Zeit noch kein interessierendes Zeichen im U1RXREG-Register liegt wird dieses bei erstmaligen Aufruf geleert (Z. 10).

Für die Speicherung der Daten wurde ein zweites FIFO (FIFO_RX) angelegt, welches analog zum ersten arbeitet.

Zusätzlich zur Speicherung der Daten wird die Funktion getNextCommand() aufgerufen:

```
1.  void getNextCommand(){
2.      //Bsp: B0\n
3.      static int i = 0;
4.      static char input[32];
5.      char zeichen;
6.         _GIE = 0;
7.      getcFIFO_RX(&zeichen);
8.         _GIE = 1;
9.      if (zeichen == '\n'){
10.         int s;
11.         for(s=0; s<32; s++){
12.             command[s] = input[s];
13.         }
14.         memset(input,0,32);
15.         i=0;
16.
17.     }
18.     else{
19.         input[i] = zeichen;
20.         i++;
21.     }
22.     return;
23.
24. }
```

Hier wird das aktuelle Zeichen aus dem FIFO gelesen. Falls das Zeichen noch nicht einem „\n“ entspricht, wird es im input-String angehängt. Sobald das ausgelesene Zeichen einem „\n“ entspricht, wird der Inhalt des input-Strings in den globalen String

command kopiert, und der input-String geleert. Auf diese Weise befindet sich im command String immer der letzte gesendete Befehl.

Um die empfangenen Befehle auszuwerten wird die Funktion „Auswertung“ zyklisch in der Superloop ausgeführt:

```
1.  void Auswertung(){
2.  
3.      int k = 0;
4.      char zahl[4];
5.      int delay;
6.      switch(command[0]){
7.          case 'R':
8.              switch(command[1]){
9.                  case '0':
10.                     b0 = 0;
11.                     LED0 = 0;
12.                     putsUART("OK\n");
13.                     memset(command,0,32);
14.                     break;
15.                 case '1':
16.                     b1 = 0;
17.                     LED1 = 0;
18.                     putsUART("OK\n");
19.                     memset(command,0,32);
20.                     break;
21.                 case '2':
22.                     b2 = 0;
23.                     LED2 = 0;
24.                     putsUART("OK\n");
25.                     memset(command,0,32);
26.                     break;
27.                 case '3':
28.                     b3 = 0;
29.                     LED3 = 0;
30.                     putsUART("OK\n");
31.                     memset(command,0,32);
32.                     break;
33.                 default:
34.                 putsUART("ERROR\n");
35.                 memset(command,0,32);
36.                 break;
37.             }
38.             break;
39.         case 'B':
40.             switch(command[1]){
41.                 case '0':
42.                     b0 = 1;
43.                     putsUART("OK\n");
44.                     memset(command,0,32);
45.                     break;
46.                 case '1':
47.                     b1 = 1;
48.                     putsUART("OK\n");
49.                     memset(command,0,32);
50.                     break;
51.                 case '2':
52.                     b2 = 1;
53.                     putsUART("OK\n");
54.                     memset(command,0,32);
55.                     break;
56.                 case '3':
57.                     b3 = 1;
```

```
                    putsUART("OK\n");
                    memset(command,0,32);
                    break;
                default:
                    putsUART("ERROR\n");
                    memset(command,0,32);
                    break;
            }
            break;
        case 'S':
            switch(command[1]){
                case '0':
                    b0 = 0;
                    LED0 = 1;
                    putsUART("OK\n");
                    memset(command,0,32);
                    break;
                case '1':
                    b1 = 0;
                    LED1 = 1;
                    putsUART("OK\n");
                    memset(command,0,32);
                    break;
                case '2':
                    b2 = 0;
                    LED2 = 1;
                    putsUART("OK\n");
                    memset(command,0,32);
                    break;
                case '3':
                    b3 = 0;
                    LED3 = 1;
                    putsUART("OK\n");
                    memset(command,0,32);
                    break;
                default:
                    putsUART("ERROR\n");
                    memset(command,0,32);
                    break;
            }
            break;
        case 'T':
                for(k=3;command[k] != 0x0; k++ ){
                    if (k>6){return;}
                    zahl[k-3] = command[k];

                }
                delay = atoi(zahl);
                switch(command[1]){
                    case '0':
                        if (command[2] == '_'){
                            T_blink0 = delay;
                            putsUART("OK\n");
                            memset(command,0,32);
                        }
                        else{
                            putsUART("ERROR\n");
                            memset(command,0,32);
                        }
                        break;
                    case '1':
                        if (command[2] == '_'){
                            T_blink1 = delay;
                            putsUART("OK\n");
                            memset(command,0,32);
```

```
123.                            }
124.                            else{
125.                                putsUART("ERROR\n");
126.                                memset(command,0,32);
127.                            }
128.                            break;
129.                        case '2':
130.                            if (command[2] == '_'){
131.                                T_blink2 = delay;
132.                                putsUART("OK\n");
133.                                memset(command,0,32);
134.                            }
135.                            else{
136.                                putsUART("ERROR\n");
137.                                memset(command,0,32);
138.                            }
139.                            break;
140.                        case '3':
141.                            if (command[2] == '_'){
142.                                T_blink3 = delay;
143.                                putsUART("OK\n");
144.                                memset(command,0,32);
145.                            }
146.                            else{
147.                                putsUART("ERROR\n");
148.                                memset(command,0,32);
149.                            }
150.                            break;
151.                        default:
152.                            putsUART("ERROR\n");
153.                            memset(command,0,32);
154.                        break;
155.                    }
156.                    break;
157.                default:
158.                    if (command[0] != 0x0 ){
159.                        putsUART("ERROR\n");
160.                        memset(command,0x0,32);
161.                        break;
162.                    }
163.
164.
165.
166.            }
167.        }
```

Generell wird hier der command-String mit Hilfe einer Switch-Case Anweisung ausgewertet. Dabei wird für das erste Zeichen die Fälle R,B,S und T unterschieden. Falls das erste Zeichen davon abweicht und der command-String schon beschreiben wurde (Z. 158) wird „ERROR“ ausgegeben und der command-String zurückgesetzt. Analog werden dann die restlichen Zeichen über Switch-Case Anweisungen abgefragt. Um das Blinken der LEDs zu steuern wird hier ein globales Flag gesetzt (b0,b1,b2,b3) welches dann in der eigentlichen Blink-Funktion abgefragt wird. Um die Periodendauer der LEDs auszulesen werden maximal 4 Zeichen ausgelesen und in ein integer-Wert konvertiert. (Z.100-105).

Nach jedem erfolgreichen Befehl wird der command-String geleert und ein „OK“ ausgegeben.

Im nachfolgenden die main-Funktion:

```
int16_t main(void)
{
    //Sleep();
    DELAY_ANPASSUNG = ((SYS_FREQ/96)*2180ull)/1000000ull; //Berechnung der Delay Anpassung

    /* Configure the oscillator for the device */
    ConfigureOscillator();

    /* Initialize IO ports and peripherals */
    InitApp();
    //init_ms_t1();
    init_ms_t4(); //Timer 2
    init_t2_t3();
    initBetriebszeit(0); //Timer 1 für Betriebszeit zählen
    initUART();

    //Alle LEDs als Ausgang
    _TRISB8 = 0;
    _TRISB9 = 0;
    _TRISB10 = 0;
    _TRISB11 = 0;
    _TRISF0 = 0;
    //_TRISD2 = 0;
    //Alle LEDs als digital
    _ANSB8 = 0;
    _ANSB9 = 0;
    _ANSB10 = 0;
    _ANSB11 = 0;
    _RP66R = _RPOUT_U1TX;
    RPINR18bits.U1RXR = 0b1011000;

    //_ANSD2 = 0;
    //Taster INC G9 als digital
    _ANSG9 = 0;
    _ANSE8 = 0;
    //Taster als Eingang
    _TRISG14 = 1;
    _TRISG15 = 1;
    _TRISG12 = 1;
    _TRISG13 = 1;
    _TRISG9 = 1;
    _TRISE8 = 1;
    //Pull-up Widerstände einschalten
    _CNPUG14 = 1;
    _CNPUG15 = 1;
    _CNPUG12 = 1;
    _CNPUG13 = 1;
    _CNPUG9 = 1;

    uint16_t Count = 0;
    //Timer Input auf RP96 mappen
    //_T2CKR = 96;

    //_CNIE = 1; //Change Notification Interrupt Enable
    _CNIP = 1; // Interupt priority heruntersetzten
    _CNIEG14 = 1; //CN für Taster aktivieren
    _CNIEG15 = 1;
    _CNIEG12 = 1;
    _CNIEG13 = 1;
    _CNIEG9 = 1;
    //Timer Input auf RP96 mappen
    _T2CKR = 96;
```

```
63.
64.     putsUART("Hello World\n");  //Willkommensnachricht beim Start des µC
65.     while(1)
66.     {
67.      _LATF0 = 0;
68.
69.         if (_T4IF) {
70.             _T4IF = 0; //Flag clearen
71.             Count++;
72.             if (Count >= HEARTBEAT_MS){
73.                 _LATF0 = 1;
74.                 Count = 0;
75.                 //SendBetriebszeit();
76.                     Blink0();
77.                     Blink1();
78.                     Blink2();
79.                     Blink3();
80.
81.                     Taster0();
82.                     Taster1();
83.                     Taster2();
84.                     Taster3();
85.                 measureProcesstime();
86.                 Auswertung();
87.
88.
89.
90.
91.
92.                 //ShowBetreibszeitLED();
93.                 //U1TXREG = 'a';
94.                 //doIdle();
95.             }
96.         }
97.     }
98. }
```

Zu sehen sind hier die Pin-Mapping für das Empfangen und Senden von Daten (Z.29&30), die Willkommensnachricht (Z.64) und die Funktionen in der Superloop.