Mikroprozessorpraktikum WS 12/13

Carlos Martín Nieto, Simon Hohberg, Tu Tran

December 8, 2012

1 I/O Port

1.1 Output

1.1.1 Erläutern Sie unter Nutzung des User's Guide die Funktionalität der vier Register:

Die folgenden P4* Register konfigurieren den Port 4.

- P4SEL: Bitmaske zum wählen zwischen I/O Funktion und peripherellem Modul für die einzelnen Portleitungen
- P4DIR: Bitmaske, diet die Richtung für die Portleitungen wählt (0⇔input oder 1 ⇔ output)
- P4OUT: wenn eine Portleitung als Ausgang konfiguriert wurde, wird über dieses Register der Ausgangswert für die Portleitung gesetzt (0 ⇔low oder 1 ⇔high)
- P4IN: wenn eine Portleitung als Eingang konfiguriert wurde, enthält diese Register den Eingangswert (0 ⇔low oder 1 ⇔high)

1.1.2 Stellen Sie eine Liste der Operatoren zur Bitmanipulation auf. Erklären Sie die Möglichkeiten zum Setzen, Rücksetzen und Toggeln einzelner bzw. mehrerer Portleitungen eines Ports am Beispiel von P4OUT.

```
1 #define BIT_SET(a,b) ((a) |= (b))
2 #define BIT_CLR(a,b) ((a) &= ~(b))
3 #define BIT_TOGGLE(a,b) ((a) ^= (b))
```

`a` ist das Bitfeld und `b` welches bit (oder welche Bits) wir ändern wollen. Für P4OUT wäre a=P4OUT und b welcher Pin. Wen man P4.X ändern will, gibt man den Wert 2^X was das Bit an der X-ten Stelle setzt. Zum setzen, verwenden wir ein bitwise-OR der Werte, so dass das Bit, das b repräsentiert, gesetzt wird (alle andere Bits bleiben gleich). Zum Rücksetzen negieren wir den Wert, so dass alle außer das X-te Bit Null gesetzt werde. Das bitwise-AND sorgt dafür, dass alle Bits außer das X-te, die vorher gesetzt waren, so bleiben. Zum Toggeln benutzen wir eine XOR-Operation. An der Stelle an der eine 0 steht, bleibt also der Wert konstant, wo eine 1 ist, wird das Bit geändert.

1.1.3 Erläutern Sie anhand der Abbildung der inneren Struktur einer Portleitung für die folgenden Registerbelegungen den Signalpfad und den Logikpegel der Portleitung P4.0.

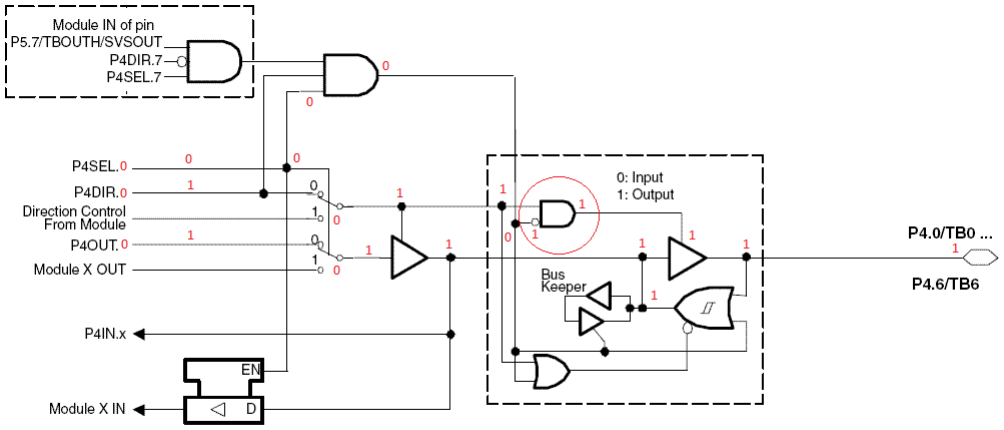

Figure 1: Signalpfad für P4.0

P4SEL(= $0x00$) setzt alle Portleitungen P4.x des Ports auf I/O. P4DIR(= $0x0F = (0000\,1111)_2$) setzt die Portleitugnen P4.0 bis P4.3 auf Output und die Portleitungen P4.4 bis P4.7 auf Eingang. Die Veroderung des Registers P4OUT mit $0x01$ (LED_ROT) setzt den Logikpegel der Portleitung P4.0, der als Ausgang konfiguriert ist, auf 1.

1.1.4 Warum leuchtet eine LED am Port P4.0, wenn der Logikpegel "0" ist und nicht bei dem Logikpegel "1"?

Die Portleitungen sind am negativen Kontakt der LEDs angeschlossen. Wenn der Pegel auf high ist, gibt es kein Potenzialdifferenz, so dass kein Strom fließt und dadurch die LED auch nicht leuchtet. Ist der Pegel low, dann entsteht eine Potenzialdifferenz, die einen Stromfluss ermöglicht, der die LED leuchten lässt.

1.1.5 Erläutern Sie inhaltlich die Bedeutung und Funktionalität der folgenden Codezeilen:

```
1  #define LEDRT (0x01) // LEDRT wird definiert, als 0te-Bit an (hier für P4.0 benutzt)
2  unsigned char a;
3  ...
4  P4DIR = 0x00; // alle pins werden als input gesetzt
5  a = 10; // a binär 1 0 1 0
6  P4OUT = a; // P4.0 und P4.2 werden auf low gesetzt, P4.1 und P4.3 auf high
7  P4OUT = 0x01; // P4.0 wird auf high gesetzt
8  P4DIR = 0x07; // P4.0 bis P4.6 werden als Ausgänge gewählt
9  P4OUT = 0x00; // alle pins werden auf low gesetzt => alle drei LEDs leuchten
10 P4OUT |= 0x01; // P4.0 wird auf high gesetzt => die rote LED wird ausgeschaltet
11 P4OUT |= LEDRT; // P4.0 wird wieder high gesetzt => die rote LED bleibt
       ausgeschaltet
12 P4OUT ^= ~LEDRT; // P4.0 wird auf low gesetzt, P4.1 bis P4.3 auf high => die rote
       LED wird wieder eingeschaltet, die anderen LED ausgeschaltet
13 P4OUT ^= LEDRT; // P4.0 auf high, P4.0 bis P4.3 sind high => die rote LED wird
       ausgeschaltet
14 ...
```

1.1.6 Schreiben Sie ein kleines Programm, welches den Durchlauf einer Sequenz eines Ampelsignals mit den Phasen (Rot, Rot-Gelb, Grün, Gelb, Rot) simuliert. Nutzen Sie dazu die bereitgestellte Funktion wait() für eine Zeitschleife. Benutzen Sie eigene Macros und Operatoren zur Bitmanipulation. Beachten Sie dabei bitte, daß nur auf Basis von Bitmanipulationen die Leitungen P4.0 bis P4.2 verändert werden. Alle anderen Leitung müssen, was die Richtung und den Zustand anbelangt unverändert bleiben.

```
1  #define BIT_SET(a,b) ((a) |= (b))
2  #define BIT_CLR(a,b) ((a) &= ~(b))
3  #define BIT_TOGGLE(a,b) ((a) ^= (b))
4
5  #define LED_ROT (0x01)    // 0 0 1
6  #define LED_GELB (0x02)   // 0 1 0
7  #define LED_GRUEN (0x04) // 1 0 0
8
9  // LEDs werden an, wenn das Bit nicht gesetzt ist, dies verbessert die Lesbarkeit
10 #define LED_ON(led) (BIT_CLR(P4OUT, led))
11 #define LED_OFF(led) (BIT_SET(P4OUT, led))
12 #define LED_TOGGLE(led) (BIT_TOGGLE(P4OUT, led))
13 #define SLEEP() (wait(65000))
14 //===Hier die Endlosschleife quasi das Betriebssystem====================
15 while(1){
16         // Fange von bekanntem Zustand an
17         LED_OFF(LED_GELB | LED_GRUEN | LED_ROT);
18         LED_ON(LED_ROT);
19         SLEEP();
20         LED_ON(LED_GELB);
21         SLEEP();
22         LED_OFF(LED_GELB | LED_ROT);
23         LED_ON(LED_GRUEN);
24         SLEEP();
25         LED_OFF(LED_GRUEN);
26         LED_ON(LED_GELB);
27         SLEEP();
28 }   // Ende der Endlosschleife
```

1.2 Input

1.2.1 Erläutern Sie unter Nutzung des User's Guide die Funktionalität der sieben Register:

Die folgenden P1* Register konfigurieren den Port 1.

- P1DIR: Bitmaske, die die Richtung für die Portleitungen wählt (0⇔input oder 1 ⇔ output)
- P1IN: enthält die Eingabewerte der einzelnen Portleitungen (0 low, 1 high)
- P1OUT: wenn eine Portleitung als Ausgang konfiguriert wurde, wird über dieses Register der Ausgangswert für die Portleitung gesetzt (0 ⇔low oder 1 ⇔high)
- P1SEL: Bitmaske zum Wählen zwischen I/O Funktion und peripherellem Modul für die einzlenen Portleitungen.
- P1IE: Bitmaske zur Aktivierung/Deaktivierung von Interrupts für die entsprechende Portleitung
- P1IES: Bitmaske, die die Flanke (Änderung des Logikpegels an einer Portleitung) definiert, welche einen Interrupt für die entsprechende Portleitung auslöst
- P1IFG: Jedes P1IFGx Bit ist das Interrupt-Flag der jeweiligen Portleitung. Dieses wird gesetzt, wenn die über P1IES ausgewählte Flanke beim Eingangssignal der Portleitung passiert.

1.2.2 Erläutern Sie die Funktion des Operators AND zur Bitmanipulation. Diskutieren Sie die Einsatzmöglichkeit folgender Schreibweisen:

Das bitwise-AND ermöglicht die logische Verknüpfung der einzelnen Bits. Dies erlabut auch das Überprüfen, ob einzelne Bits gesetzt sind.

```
...
#define T_links (0x01)
#define Taster (T_links)
#define Taste_links (P1IN & T_links)
...
if (P1IN & Taster) {...}
...
if (Taste_links) {...}
...
```

Mit `Taste_links` ist die Bedeutung fest (liefert 1, wenn der linke Taster gedrückt ist), aber mit `P1IN & Taster` kann auch ein anderer Taster geprüft werden. Dies ist beim Testen der Zustände der Taster sehr nützlich.

1.2.3 Nutzen Sie die obere Schaltung zur Erklärung der nachfolgend dargestellten Befehls- zeilen und geben Sie an, welchen Wert die Variable a (unsigned char) in den einzelnen Zeilen annimmt. Beachten sie dabei den Tastenzustand im Kommentar.

```
...
#define Taster_rechts (0x02)
#define Taster_links (0x01)
P1DIR = 0x00; // alle Pins in P1 für Input gesetzt
P4DIR = 0x07; // P4.0, P4.1 und P4.2 für Output gesetzt
P4OUT = 0; // Alle drei LEDs an
a = 7; // Bitfolge 0000111
P4OUT = a; // Alle drei LEDs aus
a = P1IN & 0x00;     // beide Tasten gedrückt
// a = 0
a = P1IN & 0x03;     // beide Tasten gedrückt
// a = 3 = b00000011
a = P1IN & 0x01;     // Taste rechts gedrückt
// a = 0
a = P1IN & 0x01;     // Taste links  gedrückt
// a = 1
a = P1IN & 0x02;     // Taste rechts gedrückt
// a = 2 = b00000010
a = P1IN & 0x02;     // Taste links  gedrückt
// a = 0
P4OUT = P1IN & Taster_rechts;    // Taster rechts nicht gedrückt
// P4OUT = 0, alle LEDs an
P4OUT = P1IN & Taster_rechts;    // Taster rechts gedrückt
// P4OUT = 0x02 = b010, gelbe LED aus, andere an
P4OUT = P1IN & Taster_links;     // Taster links nicht gedrückt
// P4OUT = 0, alle LEDs an
P4OUT = P1IN & Taster_links;     // Taster links gedrückt
// P4OUT = 0x1 = b01, rote LED aus, andere an
```

1.2.4 Schreiben Sie ein Programm mit folgender Funktionalität:

- gelbe LED (P4.1) an, wenn keine Taste gedrückt ist
- grüne LED (P4.2) an, wenn rechte Taste gedrückt und linke Taste nicht gedrückt ist
- rote LED (P4.0) an, wenn linke Taste gedrückt und rechte Taste nicht gedrückt ist
- gelbe LED (P4.1) an, wenn beide Tasten gedrückt sind

```
#define TASTE_RECHTS (0x02) // 1 0
#define TASTE_LINKS (0x01)  // 0 1

/* Diese Werte haben den Index des Wertes von P1IN & (TASTE_LINKS | TASTE_RECHTS) */
static unsigned char led_settings[] = {
    5, // Wert 0 (keine Taste gedrueckt): 1 0 1 => nur gelbe LED an
    6, // Wert 1 (linke Taste gedrueckt): 1 1 0 => nur rote LED an
    3, // Wert 2 (rechte Taste gedrueckt): 0 1 1 => nur gruen LED an
    5, // Wert 3 (linke & rechte Taste gedrueckt): 1 0 1 => nur gelbe LED an
};

// alle Portleitungen auf Eingang
P1DIR = 0x00;

//===Hier die Endlosschleife quasi das Betriebssystem====================
    while(1){
        P4OUT = led_settings[P1IN & (TASTE_LINKS | TASTE_RECHTS)];
    }   // Ende der Endlosschleife
```

1.3 Ampelsteuerung

1.3.1 Nutzen Sie alle drei LED und den linken Taster (P1.0), um eine Fußgängerampel (aus der Sichtweise des Autofahrers) zu programmieren. Folgender Ablauf soll dabei realisiert werden:

- Grundzustand alle LED aus
- Wenn Taste gedrückt wird, sofort gelbe LED (P4.1) einschalten
- Danach zeitverzögert gelbe LED aus und rote LED (P4.0) an
- Nach einer Pause gelbe LED (P4.1) zur roten LED (P4.0) dazuschalten
- Dann zeitverzögert nur die grüne LED (P4.2) einschalten
- Nach einer weiteren Pause alle LEDs ausschalten
- Erst danach mit einer größeren Wartezeit die Taste wieder abfragen

```
#define SLEEP_QUANTUM 10000
#define SLEEP(n) do { /* SLEEP n Sekunden */ \
    long time = n * 100000; /* wait() sleeps 10*n microseconds */ \
    while(time > SLEEP_QUANTUM) { \
        wait(SLEEP_QUANTUM); \
        time -= SLEEP_QUANTUM; \
    } \
    wait(time); \
} while(0)

LED_OFF(LED_ROT | LED_GELB | LED_GRUEN);
//===Hier die Endlosschleife quasi das Betriebssystem====================
    while(1){
        while (!(P1IN & TASTE_LINKS)) {
            ; // Cycles verschwenden FTW
        }
        // Wenn die Taste gedrueckt wird
        LED_ON(LED_GELB);
        SLEEP(1);
        LED_OFF(LED_GELB);
        LED_ON(LED_ROT);
        SLEEP(2);
        LED_ON(LED_GELB);
        SLEEP(3);
        LED_OFF(LED_ROT | LED_GELB);
        LED_ON(LED_GRUEN);
        SLEEP(4);
        LED_OFF(LED_GRUEN);
```

```
29          SLEEP(5);
30      }   // Ende der Endlosschleife
```

1.4 Tastenstatus Abfrage

1.4.1 Entwickeln Sie ein Programm das die Tastenbetätigungen auswertet und folgender Funktionsbeschreibung entspricht:

- Eine unsigned char Variable soll als Basis für diese Aufgabe genutzt werden
- Jede Tastenbetätigung der rechten Taste (P1.1), soll diese Variabel um eins inkrementieren
- Jede Betätigung der linken Taste (P1.0), soll diese Variable um eins dekrementieren
- Der Zahlenwert der Variablen soll als Dezimal- und Hexadezimalzahl auf dem LCD-Display dargestellt werden.
- Werden beide Tasten gleichzeitig gedrückt, wird die Variable auf Null gesetz

```
1  unsigned char wert = 0;
2  char string[256];
3  int update = 1;
4
5  //===Hier die Endlosschleife quasi das Betriebssystem====================
6      while(1){
7
8          // merke welche Taste(n) gedrueckt wurden
9          int waitfor = 0;
10
11         switch(P1IN) {
12
13             // beide Tasten gedrueckt
14             case TASTE_LINKS | TASTE_RECHTS:
15                 waitfor = TASTE_LINKS | TASTE_RECHTS;
16                 wert = 0;
17                 update = 1;
18                 break;
19
20             // linke Taste gedrueckt
21             case TASTE_LINKS:
22                 waitfor = TASTE_LINKS;
23                 wert--;
24                 update = 1;
25                 break;
26
27             // rechte Taste gedrueckt
28             case TASTE_RECHTS:
29                 waitfor = TASTE_RECHTS;
30                 wert++;
31                 update = 1;
32                 break;
33         }
34
35         if (!update) // Zeichnen ist teuer
36             continue;
37
38         // Ausgabe auf dem Display
39         sprintf(string, "DEC␣%03d\nHEX␣%03x", wert, wert);
40         lcd_clear(WHITE);    // Grafikspeicher auf dem MSP430 löschen
41         lcd_string(BLACK, 15, 25, string);  // Textausgabe
42         lcd_paint(); // Grafikspeicher auf das LCD Display ausgeben
43         update = 0;
44
45         // Warten, bis die Taste losgelassen wurde
46         if (waitfor) {
47             while (P1IN & waitfor) {
48                 ;
49             }
```

```
50          }
51      }   // Ende der Endlosschleife
```

1.5 Touchscreen als Taste

1.5.1 Stellen Sie die Anzahl der Touchscreen (TS) Berührungen auf dem LCD-Display dar. Aktualisieren Sie diese bei jeder Touchscreen Berührung. Realisieren Sie dazu folgende Funktionalität:

- solange linke Taste gedrückt → LED (P4.2) an,
- wenn dabei dabei TS getippt → Anzahl um eins erniedrigen solange rechte Taste gedrückt → LED (P4.0) an,
  wenn dabei TS getippt → Anzahl um eins erhöhen
- wenn keine Taste gedrückt → LED (P4.1) an, wenn dabei TS getippt → LED (P4.1) toggeln

```
1   print_value();
2
3   //===Hier die Endlosschleife quasi das Betriebssystem====================
4       while(1){
5
6           LED_ON(LED_GELB);
7
8           while(P1IN & TASTE_LINKS) {
9               LED_OFF(LED_GELB);
10              LED_ON(LED_GRUEN);
11              if (!(P1IN & BIT_YM)) {
12                  wert--;
13                  print_value();
14                  while(!(P1IN & BIT_YM)) {
15                  }
16              }
17          }
18
19          LED_OFF(LED_GRUEN);
20          LED_ON(LED_GELB);
21
22          while(P1IN & TASTE_RECHTS) {
23              LED_OFF(LED_GELB);
24              LED_ON(LED_ROT);
25              if (!(P1IN & BIT_YM)) {
26                  wert++;
27                  print_value();
28                  while(!(P1IN & BIT_YM)) {
29                  }
30              }
31          }
32
33          LED_OFF(LED_ROT);
34
35          if (!(P1IN & BIT_YM)) {
36              LED_TOGGLE(LED_GELB);
37              while(!(P1IN & BIT_YM)) {
38              }
39          }
40      }   // Ende der Endlosschleife
41  }       // Ende Main
42  //====Ende des Hauptprogramms ============================================
43
44  void print_value(void) {
45          char string[256];
46          sprintf(string, "DEC␣%03d\nHEX␣%03x", wert, wert);
47          lcd_clear(WHITE);    // Grafikspeicher auf dem MSP430 löschen
48          lcd_string(BLACK, 15, 25, string);  // Textausgabe
49          lcd_paint();         // Grafikspeicher auf das LCD Display ausgeben
50  }
```