Skript und Aufgaben zum OOSE-Labor

Einleitung

Im OOSE-Labor sollen die in PDA, RUM und OOSE erlernten Konzepte integriert angewendet und vertieft werden.

Die Inhalte dieser Veranstaltungen werden als bekannt vorausgesetzt! Es wird von Ihnen erwartet, dass Sie bei Bedarf in den Vorlesungen/Übungen nachschlagen. Vor allem die Inhalte von Kapitel 3 und 4 von RUM sind unabdingbar. Lösungen zu Übungen oder zur Testataufgabe, welche die in den genannten Kapiteln besprochenen µP-spezifischen Aspekte ignorieren, werden nicht akzeptiert!

Es wird von Ihnen erwartet, dass Sie zur Bearbeitung der jeweiligen Aufgaben vorbereitet im Labor erscheinen. Wer ungenügend unvorbereitet erscheint, muss mit Verweis aus dem Labor rechnen! Gleiches gilt bei Verletzung der Hygieneregeln!

Das in diesem Modul betrachtete Anwendungsgebiet sind **eingebettete Systeme**, d.h., die Aufgaben werden die Verwendung von µ-Controllern zur Ansteuerung von Peripherie zum Gegenstand haben und **Performanz (Platz- und Zeitbedarf) als notwendige Qualitätseigenschaft der Lösungen** (Ihrer Lösungen !) betrachten.

Es wird auf dem aus RUM bekannten Prozessor ATMEL Mega 2560 und dem STK600 Board aufgesetzt und es werden in geringem Umfang Assembler (ASM), hauptsächlich C und schließlich C++ Anwendungen entwickelt.

Das Ziel wird sein, möglichst einfach verwendbare **Treiber**, idealerweise als C++-Klassen zu entwickeln, deren Instanzen die Komponenten des Prozessors (Ports, Timer, usw.) in leicht verwendbare Software-Objekte abbilden. Themen des Labors sind also **Hardware-Abstraktionen**, die in der Literatur oft als HAL (**hardware abstraction layer**) bezeichnet werden. Konkrete Anwendungen, wie bspw. komplexe Steuer-und Regelalgorithmen, werden im Rahmen des Labors nicht untersucht, würden aber auf der HAL aufgesetzt werden können.

Lernziele:

- Sie haben verstanden, wie man Assembler-Routinen aus C/C++ aufruft und wie man Programmteile aus C und C++ zusammenbringt (mit einem C++ Hauptprogramm).
- Sie haben verstanden, wie Hardware-Ressourcen durch **Treiber** (in ASM, C oder C++) abstrahiert (z.B. unter Verwendung des Konzepts „handle“) sowie einheitlich und komfortabel benutzt werden können.
- Sie können Hardware-Ressourcen mit der (objektorientierten) Software-Welt verbinden und wissen, welche Mechanismen zur Abstraktion der Hardware in Klassen/Objekten verwendet werden (z.B. statische Objekte, Registrierung und Aufrufe von Callback-Prozeduren, um Methoden als ISRs aufrufen zu können).
- Sie können, aufbauend auf primitiven Klassen zur Ansteuerung von Peripherie (z.B. allgemeine digitale Ports), speziellere Klassen ableiten (Vererbung), die spezifische Peripherie (z.B. LCDs, 7-Segment Anzeigen, Drehgeber, AD-Wandler) ansteuern können.

Aufgabe 1: Aus C aufrufbare Assembler Routinen zum Lesen und Schreiben digitaler Ports

Obwohl in C/C++ vieles sehr hardware-nah programmiert werden kann, sind Assembler-Unterprogramme bei der Controller-Programmierung für die Ansteuerung spezieller Hardware-Funktionen, Nutzung spezieller Assembler-Befehle und zwecks maximaler Optimierung gelegentlich unverzichtbar. Bei dieser Aufgabe soll daher eingeübt werden, wie Unterprogramme, die in Assembler implementiert sind, geschrieben sein müssen, damit sie aus C aufrufbar werden. Während in reinem Assembler der Programmierer frei entscheiden kann, wie er Parameter übergibt (in Registern, über den Stack, oder in globalen Variablen), muss er bei Anbindung an eine compilierte Sprache die Konventionen des jeweils verwendeten Compilers auf der jeweiligen Ziel-Hardware insbesondere bzgl. Registernutzung und Parameterübergabe beachten.

Wir verwenden im Labor ATMEL Studio 7 mit der GNU toolchain: C++ Compiler, Assembler und Binder und müssen uns daher an deren Konventionen zur Parameterübergabe bei Code-Generierung für Prozessoren der AVR Familie halten. Lesen Sie jetzt(!) das Dokument **Linking_ASM_and_C.pdf**[1] auf OLAT!
Zum Einlesen in die Materie „C Programme auf AVR Prozessoren“ können Sie hier einsteigen:
http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial
und für spezielle Details hier weiterlesen / suchen:
http://nongnu.org/avr-libc/user-manual/FAQ.html
Weitere Infos auf der OLAT Seite zum OOSE-Lab unter „Weitere Unterlagen“.

Zur Anbindung ASM an C ist im Wesentlichen folgendes zu tun:

1. In der Assembler-Datei: die Labels (Bezeichner) aller ASM-Unterprogramme, die aus C aufrufbar sein sollen, müssen vom Assembler an den Binder gemeldet werden. Dazu werden die Labels mit der Assembler-Direktive **.global name** (mit name = Label des Unterprogramms) als global, also außerhalb der Assembler-Datei sichtbar deklariert. In gleicher Weise können auch Bezeichner von globalen Variablen und Konstanten von Assembler nach C (und C++) exportiert werden. Bedenken Sie: Labels haben keinen Typ; sie stehen für Adressen. Was diese Adressen bezeichnen und die entsprechende sinnvolle Nutzung derselben, ist alleine Sache des Programmierers!
2. In C muss ein zu diesen global-Deklarationen von Unterprogramm-Labels passendes Header-File geschrieben werden, in dem für jedes Assembler-Unterprogramm dessen Prototyp in C-Nomenklatur beschrieben ist. Im Header-File werden also der Rückgabetyp des Unterprogramms und die Parameterliste deklariert und damit C-seitig eine Typisierung festgelegt. Beachten Sie: es gibt keine korrespondierende „Deklaration“ im Assembler-File und folglich keine Möglichkeit, dass der Assembler die Passung von Implementierung

[1] Im Dokument wird Studio 6 als Entwicklungsumgebung genannt, es gilt aber ebenso für das aktuelle Studio 7.

und Deklaration prüfen könnte. Sie sind selbst für die passende Implementierung verantwortlich!
In analoger Weise werden globale Assembler-Variablen (die im SRAM liegen müssen!) in C deklariert (mit einem Typ versehen und ggf. mit volatile und/oder const deklariert). Mit diesem Header-File kann nun in C gearbeitet werden, als ob die Unterprogramme und Variablen oder Konstanten in einem C Modul definiert wären. Sollen die ASM-Unterprogramme auch in C++ eingebunden werden, geht das genauso, allerdings muss dann das Headerfile als C-style markiert werden (**`extern "C" { … }`**), wie in der Textbox gezeigt. Am besten schreibt man jeden Header immer so wie in der Box gezeigt, dann sind die Header in C und C++ gleichermaßen anwendbar.

```
#ifdef __cplusplus
extern "C" {
#endif
…
C-Prototypen für die
ASM Routinen
…
#ifdef __cplusplus
}
#endif
```

3. Bei der Implementierung der Assembler-Routinen ist nun genau darauf zu achten, dass die Parameter und Rückgabewerte gemäß Deklaration in der C-Header auch tatsächlich implementiert werden. Details siehe Kapitel 5 und 6 in **Linking_ASM_and_C.pdf**. Wichtig ist zu verstehen, dass (bei dieser Kombination aus Prozessor und C-Compiler) Parameter grundsätzlich als 16 Bit Werte (also in Registerpaaren) übergeben werden und einige der 32 Register (beginnend von R25 abwärts bis max. R18) dazu verwendet werden. Einfache 8-Bit Werte (bspw. char, int8_t oder uint8_t in C) verwenden dabei immer nur das Register des Paars mit der jeweils niedrigeren Nummer (siehe Beispiel unten). Sie sollten alle anderen Register in der Assembler-Routine entweder nicht verwenden, oder mittels push und pop sichern und restaurieren. Wird die Parameterliste sehr lang, weicht der Compiler ab dem 5. Parameter auf den Stack aus – das kommt aber in unserer Aufgabe nicht vor. Funktionsrückgabewerte legen Sie im Registerpaar R25:R24 ab – einen 8-Bit Wert bspw. also in R24; und das ungenutzte R25 wird dann einfach auf null gesetzt.

Beispiel (schematisch):

```
// das C header file myproc.h, passend zur Assembler-Datei
extern uint8_t myproc (uint8_t param);
```

```
// Ausschnitt aus C-Datei, in der die Assembler-Routine
// gerufen wird:
#include "myproc.h"
uint8_t xyz;
…
xyz = myproc (28);  // xyz erhält den Funktionswert von myproc
```

```
// Assembler Datei myproc.s mit der Implementierung der Routine:
.global myproc
myproc:
      push …   ;sichere alle Register, die in dieser Routine
```

```
            ;  verwendet werden
            ;r24 enthält den 8-Bit Parameterwert(hier: 28)
            ;r25 ist null
…
clr r25         ;setze den 8-Bit Funktionswert 99 in das
ldi r24, 99     ;  Registerpaar r25:r24
pop …       ;Restauriere zerstörte Registerwerte
ret
```

Aufgabenstellung:

Die Aufgabe 1 besteht nun darin, die folgenden drei Unterprogramme in Assembler zu schreiben, dazu ein C-Header File mit zu den drei Unterprogrammen passenden Deklarationen anzulegen und ein C-Hauptprogramm zu schreiben, mit dem die Unterprogramme aufgerufen und getestet werden können.

- **void initialize (portreg, mode)** – setzt den Port mit Adresse portreg in den Input oder Output Modus. (Schauen Sie ggf. in der Vorlesung RUM die ersten Übungen nochmal an.)
- **void write (portreg, bits)** – schreibe den 8-Bit Wert bits auf den Port mit Adresse portreg.
- **uint8_t read (portreg) –** liefere ohne zu warten und ohne Entprellung den Port-Wert zurück.

Achtung: Welches der 3 Port-Register müssen Sie jeweils mittels portreg übergeben?

Hinweise:

Die Parameter **`bits`** und **`mode`** sollten **`uint8_t`** sein.

Der Parameter **`portreg`** ist jeweils die Adresse eines der Port-Register, die ihrerseits 8-Bit sind und unbedingt als volatile deklariert werden müssen. Der formale Parameter sollte also als **`volatile uint8_t* portreg`** definiert werden. Dann können in C die üblichen Port-Bezeichnungen PORTA, PORTB, …, PINA, PINB, …, DDRA, DDRB, … als aktuelle Parameter an die Prozeduren übergeben werden. Damit diese Namen (es sind Präprozessor-Makros) bekannt sind, muss deren Deklaration in den C-Dateien inkludiert werden: **#include <avr/io.h>**.

Zur Verwendung der Ports beachten Sie bitte folgende feste Zuordnung:

PORT A – LEDs
PORT B – nicht belegt
PORT C – LCD
PORT D – 7-Segment Anzeige (100/1000-er Stellen)
PORT E – 7-Segment Anzeige (1/10-er Stellen)
PORT F – analoge Temperatur und Feuchte Sensoren
PORT G – nicht belegt
PORT H – nicht belegt
PORT J – Drehgeber
PORT K – Tasten

Überlegen Sie nun, wie Sie auf der C-Seite davon abstrahieren können 3 verschiedene Port Register jeweils korrekt (also je nach gerufener Prozedur) angeben zu müssen! Hinweis: Für jeden Port gilt, dass die Register (bei unserem Prozessor) immer in gleicher Weise nebeneinander im Speicher angeordnet sind. (Vergleiche den Ausschnitt der Tabelle von Seite 403 des Datenblatts zum Prozessor). Man kann also eines der Register als universelle Portadresse festlegen, d.h., als aktuellen Wert für alle Parameter **`portreg`** verwenden. Die ASM Routinen berechnen daraus die Adressen der jeweils tatsächlich benötigten Register mit dem Wissen, dass sie nebeneinander im Speicher liegen. Solche Routinen sind offenbar hardwarespezifisch, also nicht auf andere Hardware direkt portierbar, aber – und das ist der vorrangige Zweck einer solchen Schicht von Prozeduren – leicht und einheitlich zu benutzen. → Hardware Abstraction Layer

Address	Name
0x14 (0x34)	PORTG
0x13 (0x33)	DDRG
0x12 (0x32)	PING
0x11 (0x31)	PORTF
0x10 (0x30)	DDRF
0x0F (0x2F)	PINF
0x0E (0x2E)	PORTE
0x0D (0x2D)	DDRE
0x0C (0x2C)	PINE
0x0B (0x2B)	PORTD
0x0A (0x2A)	DDRD
0x09 (0x29)	PIND
0x08 (0x28)	PORTC
0x07 (0x27)	DDRC
0x06 (0x26)	PINC
0x05 (0x25)	PORTB
0x04 (0x24)	DDRB
0x03 (0x23)	PINB
0x02 (0x22)	PORTA
0x01 (0x21)	DDRA
0x00 (0x20)	PINA

Als Testprogramm können Sie ein einfaches C Hauptprogramm mit Tastenlesen und Echo auf die LEDs nutzen.

Diese Aufgabe kann vollständig zuhause mit Studio 7 übersetzt und im Simulator getestet werden.

Bedenken Sie, dass die Hardware des STK600 negative Logik implementiert.

Lernziele (versuchen Sie für sich zu reflektieren, ob Sie die erreicht haben, sonst: nachfragen!):

- Sie können erklären, wie Symbole (Labels, Bezeichner) zwischen verschiedenen Programmiersprachen ausgetauscht werden und was dazu in der jeweiligen Sprache zu tun ist.
- Sie können die 3 Arten erklären auf die Parameter und Funktionsrückgabewerte transportiert werden und was Sie dazu in den jeweils beteiligten Sprachen tun / beachten müssen. Sie kennen Vor- und Nachteile der 3 Verfahren.
- Sie können die Problematik der Abstraktion von Hardware-Ressourcen in Software am Beispiel der digitalen Ports des ATMega2560 erläutern und kennen Möglichkeiten der Implementierung.

Aufgabe 2: Ansteuerung digitaler Ports mit „Komfort“ in C

In Aufgabe 1 haben Sie primitive Prozeduren in Assembler implementiert – Schwerpunkt lag auf der Sprachanbindung ASM an C (Parameterübergabe) jedoch nur rudimentäre Abstraktion von den drei IO Registern je Port (PINx, DDRx, PORTx). In Aufgabe 2 soll nun ein Satz Unterprogramme in C geschrieben werden, die den Umgang mit digitalen Ports noch komfortabler gestalten. Wir nennen solch einen Satz zusammengehörender Routinen einen **Treiber**.

Eine komfortablere und sicherere Programmierung der Ports mittels solcher Treiberroutinen könnte in C etwa wie folgt aussehen (Variante 1):

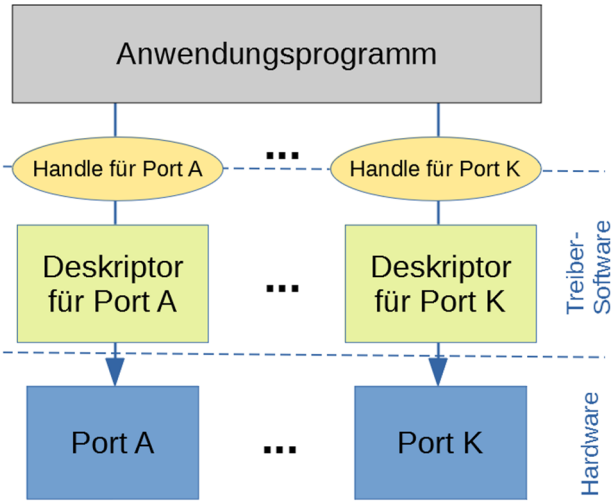
```
typedef struct { … } DigiPort_t;// Descriptor-Typ für einen Port

//erzeuge zwei Port Descriptor-Datenstrukturen:
DigiPort_t p_leds, p_keys;

// konfiguriere die Ports A und K für Ausgabe bzw. Eingabe und
// initialisiere deren DigiPort_t Strukturen. PA, PK, INPUT und
// OUTPUT seien passend definierte Makros:
initialize (&p_leds, PA, OUTPUT);
initialize (&p_keys, PK, INPUT);

// lies die Tasten ohne zu blockieren und gib deren Wert sofort
// auf dem LED Port aus:
do {
   write (&p_leds, read_raw (&p_keys));
} while(1);
```

Ein wesentlicher Teil des Komforts (und der Abstraktion von der Hardware) besteht in einer geeigneten Implementierung des Typs **DigiPort_t**, einer C-Struktur in der alle notwendigen Daten des Ports abgelegt werden. Welche Sind das? Die Adressen solcher **Deskriptoren** genannter Strukturen (im Beispiel &p_leds oder &p_keys) werden dann als Parameter für die Routinen verwendet, die sich aus den Deskriptoren die notwendigen Informationen entnehmen, bzw. Informationen dort ablegen, bspw. Konfigurations- und Statusinformationen.

Noch eleganter wird es, wenn man sich einen **„Handle“-Typ** definiert und dadurch die Adressoperation auf dem Deskriptor unnötig machen: **`typedef DigiPort_t* DigiPort_h`**. Nun verwenden Sie nur noch Variablen vom Typ DigiPort_h, die ohne den Adress-Operator zu schreiben sind. Die Implementierung sieht dann wie folgt aus (Variante 2):

```
typedef struct { … } DigiPort_t; // Descriptor-Typ für einen Port

typedef DigiPort_t* DigiPort_h;  // Handle-Typ für DigiPort_t

DigiPort_h leds_h, keys_h; //erzeuge zwei Port Handles

// konfiguriere die Ports A und K für Ausgabe bzw. Eingabe, erzeuge
```

```
// deren Deskriptoren, initialisiere diese und übergib die handles
// zu den Port-Deskriptoren an den Aufrufer:
initialize (leds_h, PA, OUTPUT);
initialize (keys_h, PK, INPUT);

// lies die Tasten ohne zu blockieren und gib deren Wert sofort
// auf dem LED Port aus:
do {
   write (leds_h, read_raw (keys_h));
} while(1);
```



Der Vorteil der Variante 2 (die Sie oft in professionellen Implementierungen vorfinden) ist, dass der Benutzer der Treiberroutinen nicht sieht, was hinter dem Handle steckt; der Deskriptor-Typ kann verborgen bleiben. Ein Handle-Typ kann ein Zeiger, aber auch ein Integer, der von den Treiber-Routinen als Index in eine Tabelle von Deskriptoren verwendet wird sein (in C sind die handle **`stdin`** und **`stdout`** bspw. solche Index-Nummern). Dies ist aber für den Nutzer der Treiberroutinen unerheblich. Insofern ergibt sich eine „gute“, weil abstrahierende Schnittstelle zw. Treibersoftware und Applikationsprogramm (siehe Bild).

Sie sollten sich für Variante zwei entscheiden (Aufwand ist der Gleiche, aber Variante 2 ist eleganter und professioneller). Implementieren und testen sie die folgenden Funktionen (der Parameter port ist dann gemäß Variante 2 jeweils als Handle zu realisieren):

- **void initialize (port, mode)** – setzt den Port in den Input oder Output Modus. Für mode sollten Sie #define Konstanten oder noch besser einen enum Typ definieren.
- **void write (port, bits)** – schreibe den 8-Bit Wert bits auf den Port.
- **void toggle (port, bits)** – invertiere diejenigen Bits des Ports, die in bits auf 1 gesetzt sind
- **void off (port, bits)** – schalte die Bits des Ports aus, zu denen in bits eine 1 steht.
- **void on (port, bits)** – schalte die Bits des Ports ein, zu denen ein bits eine 1 steht.
- **uint8_t read_raw (port, maske) –** liefere ohne zu warten und ohne Entprellung den momentanen Port-Wert zurück.
- **uint8_t read_busy_wait (port, maske)** – warte (endlos) bis eines oder mehrere der Bits, die in maske gesetzt sind, am Port auf null (active low!) gehen. Liefere dann diese Bits als Funktionswert in active high Logik zurück.

Die Funktionen write, toggle, off und on sind offenbar für die komfortable LED Ansteuerung gedacht. Die Funktionen read_raw und read_busy_wait für das Lesen von Tasten.

Hinweise:

- Sie müssen sich mit den in C verfügbaren Bitoperationen auseinandersetzen – schlagen Sie diese in den Folien zu PDA, RUM oder OOSE nach!
- Die Ports des ATmega 2560 auf dem STK600 sind **active low**. Das heißt, wenn eine Taste an einem Eingabe-Port anliegt und gedrückt wird, geht das zugehörige Bit auf null. Sonst sind alle Bits auf eins. Bei Ausgabeports mit angeschlossenen LEDs heißt das, dass die LED angeht, wenn eine null geschrieben wird. Alle Parameter (bits und maske) und alle Funktionsrückgabewerte sollen jedoch in **active high Logik** funktionieren. Sie müssen jetzt in den Treiberroutinen für die notwendigen Invertierungen sorgen, haben dafür später den Vorteil der wesentlich einfacheren Nutzung im Anwendungsprogramm.
- Der Grunddatentyp für einen 8-Bit Prozessor ist in C / C++ der Typ **uint8_t** (im Grunde ein Alias für unsigned char). Diesen sollten Sie grundsätzlich für alles verwenden, was 8 Bit hat. Sollten in späteren Aufgaben 16 Bit Werte vorkommen, so sind diese dann vom Typ uint16_t. Typen mit Vorzeichen verwenden Sie wirklich nur, wenn Sie es unbedingt brauchen (int8_t, int16_t)! (Auf größeren Prozessoren, bspw. 32 oder 64 Bit, wäre **int** mit Vorzeichen der Grunddatentyp – Sie müssen der Hardware entsprechend programmieren!)
- Die Adressen der 8-Bit Register eines Ports x sind (wie schon aus Assembler bekannt) durch je drei Konstanten der Art PORTx, PINx, DDRx spezifiziert. In C sind diese Konstanten ebenfalls bekannt (Sie müssen dazu **#include <avr/io.h>** angeben). Diese sind in C/C++ als Zeiger vom Typ `volatile uint8_t*` zu behandeln, denn sie zeigen ja auf die 8-Bit Port Register im ~~SRAM, deren Werte~~ sich ohne Zutun der Software ändern können (volatile!) .
- Als Warte-Routinen dürfen Sie vordefinierte Routinen aus **util/delay.h** verwenden. Dazu müssen Sie vor das include schreiben: **#define F_CPU 16000000UL** (siehe auch Beispiel nächste Seite). Diese delay-Routinen funktionieren zwar abhängig von der eingestellten Taktfrequenz (hier 16MHz), sie liefern aber trotzdem nur recht ungenaue Delay-Zeiten. Es stehen zur Verfügung **_delay_ms(ms)** und **_delay_us(us)**, wobei ms und us 8 oder 16 Bit Werte sein können. Beachten Sie, dass das STK600-Board die zur Konstanten F_CPU passende Frequenzeinstellung auch tatsächlich hat und außerdem die CKDIV8 fuse nicht eingeschaltet ist. (Lassen Sie sich im Labor zeigen, wie das geht.)

Es ist wichtig, dass Sie Ihre Prozeduren ausgiebig testen, denn deren Code wird in den folgenden Aufgaben wiederholt als Grundlage weiterer Lösungen (dann auch in C++) dienen. Beim Testen im Simulator alle delays ausklammern – das dauert sonst zu lange.

Auf der nächsten Seite finden Sie ein Testprogramm (wählen Sie die den Abschnitt entsprechend Ihrer Variante aus), mit dem alle Routinen mindestens einmal verwendet werden. Dies sollte, wenn Sie Ihren Treiber korrekt implementiert haben, problemlos laufen. Im Simulator die delay Aufrufe wieder ausklammern!

Lernziele (versuchen Sie für sich zu reflektieren, ob Sie die erreicht haben, sonst: nachfragen!):

- Sie können erklären, wozu ein Deskriptor gut ist und wie Sie ihn implementieren können.
- Sie können erklären, wie man die konkrete Implementierung eines Deskriptors im Anwendungsprogramm mittels handle-Konzept verbergen kann. Warum ist das wünschenswert?

```
#include <avr/io.h>
#ifndef F_CPU
   #define F_CPU  16000000UL
#endif
#include <util/delay.h>
#include "DigiPort.h"
```

Variante 1

```
DigiPort_t ledport;
DigiPort_t swport;

int main(void)
{
      initialize (&ledport, PA, SET_OUT_PORT);
      initialize (&swport, PK, SET_IN_PORT);
      write (&ledport, 0xAA);
      _delay_ms(500);
      off (&ledport, 0x0F);
      _delay_ms(500);
      on (&ledport, 0xF0);
      _delay_ms(500);
      toggle (&ledport, 0x3C);
      _delay_ms(500);
      toggle (&ledport, 0x3C);

      read_busy_wait(&swport, 0xFF);
      off(&ledport, 0xFF);

    do {
            write (&ledport, read_raw(&swport,0xFF));
    } while (1);
}
```

Variante 2

```
DigiPort_t ledport;
DigiPort_t swport;
DigiPort_h led_h = &ledport;
DigiPort_h sw_h  = &swport;

int main(void)
{
      initialize (led_h, PA, SET_OUT_PORT);
      initialize (sw_h, PK, SET_IN_PORT);
      write (led_h, 0xAA);
      _delay_ms(500);
      off (led_h, 0x0F);
      _delay_ms(500);
      on (led_h, 0xF0);
      _delay_ms(500);
      toggle (led_h, 0x3C);
      _delay_ms(500);
      toggle (led_h, 0x3C);

      read_busy_wait(sw_h, 0xFF);
      off(led_h, 0xFF);

    do {
            write (led_h, read_raw(sw_h,0xFF));
    } while (1);
}
```

Aufgabe 3: Abbildung der digitalen Ports in eine C++ Klasse

Die Tatsache, dass (bei unserem Prozessor) jeder Port durch 3 Register repräsentiert ist, haben wir in Aufgabe 2 durch ein Handle auf einen Deskriptor (eine C struct) in den zugehörigen Treiber-Routinen gut versteckt. Der Schritt zur „echten" Objektorientierung ist daher klein:

Aufgabenstellung (gehen Sie schrittweise dem Text folgend vor)**:**

1. Legen Sie ein neues Projekt (ein **GCC C++ Executable Project**) für Aufgabe 3 an und selektieren Sie dieses als neues Start-up Project im Solution Explorer (damit dieses nun das Hauptprogramm der Solution darstellt).
2. Überführen Sie den C-Typ `DigiPort_t` aus Aufgabe 2 in eine Klassendeklaration (in einer neuen Header Datei `DigiPort.h`) und einer neuen Implementierungsdatei zur Klasse: `DigiPort.cpp`. Machen Sie aus den C-Treiber-Routinen der Aufgabe 2 nun C++ (Instanzen-)Methoden der Klasse `DigiPort`.
3. Überlegen Sie dabei, welche `inline` definiert sein können. In welcher Datei müssen folglich die Definitionen dieser Routinen stehen?
4. Wie sieht der Konstruktor der Klasse aus und wie verbinden Sie die technische Ressource Digitaler Port des Prozessors mit dem Software-Objekt?
5. **Überlegen Sie, wie Sie verhindern können, dass mehrere Objekte auf demselben Port erzeugt werden? Implementieren Sie, was dazu nötig ist, um dies zu verhindern! (Hinweis: das sollte in der Klasse gekapselt sein (-> Klassenvariablen, Klassenmethoden)).**
6. Erweitern Sie die Klasse (inkl. aller Methoden) so, dass ein Port wahlweise active-high oder active-low betrieben werden kann. Diese Eigenschaft soll dem Konstruktor als Parameter mitgegeben werden (siehe Testprogramm unten). Das Port-Objekt muss sich den gewählten Betriebsmodus merken und die Instanzen-Methoden müssen Werte der Portregister beim Lesen/Schreiben je nach Betriebsmodus invertieren oder nicht invertieren.
7. Erweitern Sie nun die Methoden noch so, dass der jeweils letzte Parameter (die Bit-Maske) mit Defaultwert 0xFF (alle Bits sind angesprochen) versehen wird, wenn beim Aufruf nichts anderes angegeben wird (kein aktueller Parameterwert angegeben wird).
8. Nutzen Sie nun untenstehendes Testprogramm unverändert (!), um Ihre Klasse zu testen.

Hinweis: In der Aufgabe 2 haben wir Handles benötigt, um die Interna des Deskriptors vom Applikationsprogrammierer zu verbergen. Dies ist in Aufgabe 3 nicht mehr nötig, da C++ Klassen die Möglichkeit bieten ihre Interna zu verstecken (indem diese „private" deklariert werden). Die Objekte sind damit ähnlich abstrakt wie die Handles aus Aufgabe 2.

Lernziele (versuchen Sie für sich zu reflektieren, ob Sie die erreicht haben, sonst: nachfragen!):

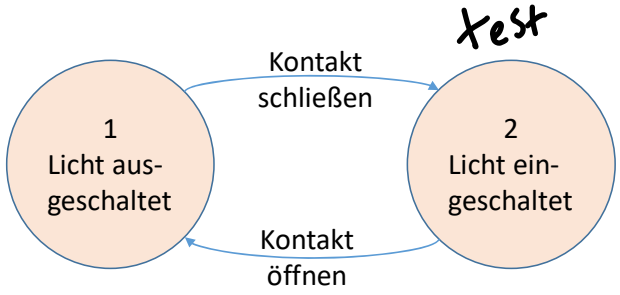
- Sie können erklären, durch was das Handle aus Aufgabe 2 beim Übergang in die Objektorientierung ersetzt wird (warum keine Handles mehr benötigt werden).
- Sie können erläutern wie und wann die Verbindung zwischen Hardware und den zugehörigen abstrahierenden Software-Objekte hergestellt wird.
- Sie können erläutern was passiert, wenn Sie mehrere Software-Objekte mit der gleichen Hardware Ressource verbinden und durch welche Vorkehrungen man dies verhindert.

```
#include <avr/io.h>
#ifndef F_CPU
#define F_CPU  16000000L
#endif
#include <util/delay.h>
#include "DigiPort.h"

// statt wie in Aufgabe 2 C-structs zu erzeugen
// erzeugen wir jetzt globale C++ Objekte:
DigiPort ledport(PA, SET_OUT_PORT, SET_ACTIVE_LOW);
DigiPort swport(PK, SET_IN_PORT, SET_ACTIVE_LOW);

int main(void)
{
   // statt wie in Aufgabe 2 Prozeduren aufzurufen,
   // die die structs als Parameter haben, rufen wir
   // Methoden der Objekte auf:
   ledport.write (0xAA);
   _delay_ms(500);
   ledport.off (0x0F);
   _delay_ms(500);
   ledport.on (0xF0);
   _delay_ms(500);
   ledport.toggle (0x3C);
   _delay_ms(500);
   ledport.toggle (0x3C);

   swport.read_busy_wait();
   ledport.off(0xFF);

    do {
       ledport.write (swport.read_raw());
    } while (1);
}
```

(Auch hier im Simulator wieder die delays ausklammern.)

Aufgabe 4: Ampelsteuerung als Endlicher Automat in C++

Für die präzise und eindeutige Beschreibung von technischen Systemen wird oft ein sogenannter Zustandsautomat, auch **Endlicher Automat** (EA) oder finite state machine (FSM) genannt, verwendet. Der Automat heißt **endlich**, wenn die Anzahl seiner Zustände endlich ist. Der Automat ist ein Modell des Verhaltens des betrachteten Systems. Ein EA kann grafisch oder auch als Tabelle beschrieben werden (siehe auch Wikipedia: „Endlicher Automat“):



Übergangs-bedingung momentaner Zustand	Kontakt schließen	Kontakt öffnen
1	2	--
2	--	1

Die tabellarische Darstellung von Endlichen Automaten (selbst wenn sie hunderte von Zuständen haben) ist besonders einfach in ein Programm umzusetzen: die Zustandstabelle wird als solche im Rechner abgelegt. Ein einfacher Algorithmus ermittelt, ausgehend vom aktuellen Zustand (linke Spalte) und aktuell vorliegender Übergangsbedingung (obere Zeile) den nächsten Zustand aus dem Körper der Tabelle. Bei jedem Zustandsübergang erfolgt eine Aktion des Automaten, z.B. eine Änderung von Daten oder die Ansteuerung von Peripherie.

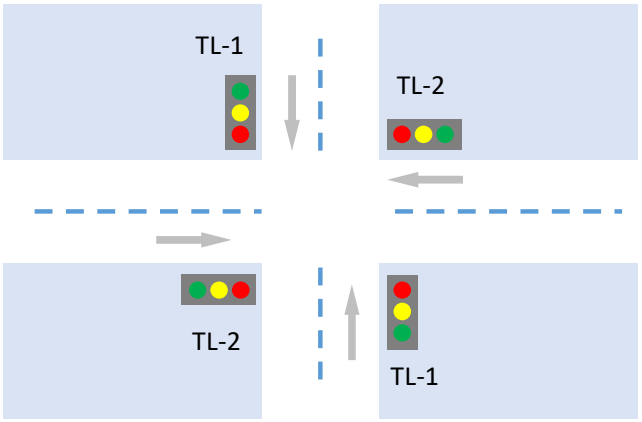
Die Aufgabe 4 besteht in der **Implementierung der Steuerung einer Lichtzeichenanlage (LZA) an einer Kreuzung als Endlicher Automat.**
Die genaue Funktionsweise einer LZA (umgangssprachlich Ampelanlage oder kurz Ampel) wird in Deutschland durch die Richtlinien für Lichtsignalanlagen (RiLSA) der Forschungsgesellschaft für Straßen- und Verkehrswesen festgelegt. Eine LZA steuert den Verkehr mit Hilfe der drei Signalfarben Grün, Gelb und Rot. Zur Regelung des Verkehrs werden diese Farben einzeln oder in Kombination angezeigt. Für die Berechnung der Dauer der Lichtzeichen gibt es umfangreiche Vorschriften[2]. Die Reihenfolge (Signalfolge) einer LZA ist dabei immer:

- Grün: Der Verkehr ist freigegeben
- Gelb: Auf nächstes Signal warten
- Rot: Keine Einfahrerlaubnis

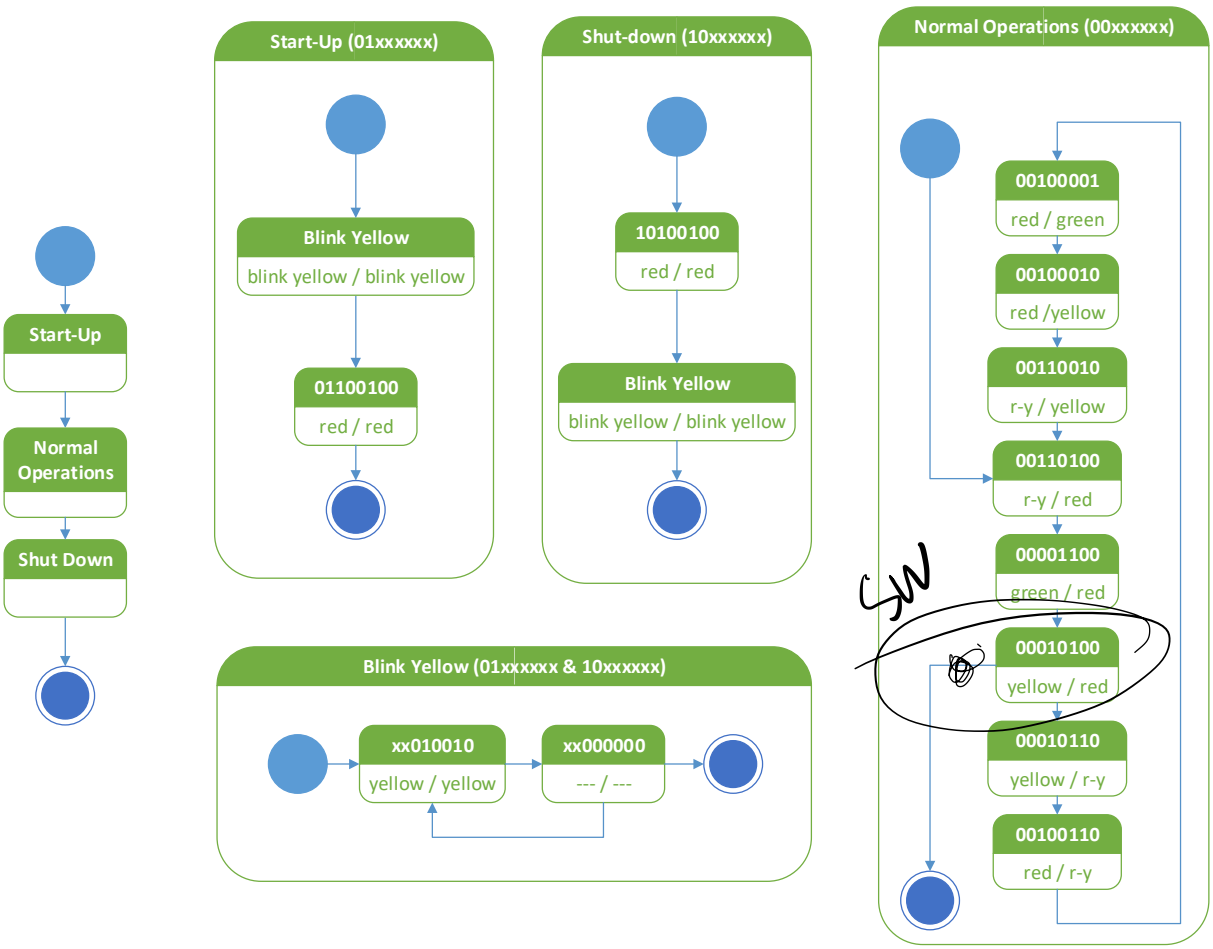
Darüber hinaus können Farbkombinationen (Rot-Gelb oder Grün-Gelb) und Blinkzeichen (Rot blinkend, Gelb blinkend) Verwendung finden. Rot ist immer oben angeordnet. Dies ermöglicht Menschen mit Rot-Grün-Sehschwäche oder Farbenblindheit die Orientierung. Mehr Details dazu siehe z.B. unter https://de.wikipedia.org/wiki/Ampel
Wir wollen für die Aufgabenstellung von einer einfachen Kreuzung mit 4 Ampeln-Pfosten ausgehen, von denen jeweils 2 zeitgleich mit identischer Signalfolge geschaltet werden (TL-1 und TL-2). Induktionsschleifen in Fahrbahnen oder Fußgänger-Anforderungen sowie zusätzliche Signale (Grünpfeil, Fußgängerampel, Radfahrerampel, etc.) sollen nicht berücksichtigt werden. Wir wollen eine LZA-Variante mit 4 Zuständen in der Signalabfolge implementieren, bei der beim Übergang von Rot nach Grün eine Rot-Gelb Phase zwischengeschaltet ist.

[2] https://de.wikipedia.org/wiki/Zwischenzeit_(Verkehrstechnik)

Ferner soll das Einschalten (Start-up) der Ampelanlage so implementiert werden, dass der Verkehr durch gelbe Blinkzeichen und anschließendes gesamt-Rot auf die normale Betriebsart (Normal-Operations) der Ampel gefahrlos synchronisiert wird. Beim Ausschalten (Shut-Down) soll der umgekehrte Fall implementiert werden: durch gesamt-Rot gefolgt von gelbem Blinken und dann gesamt-Aus soll der Verkehr gefahrlos auf die sonst geltenden Regeln (hier wäre es „rechts-vor-links“) synchronisiert werden.



Das folgende Bild gibt die Zustände der LZA in Form eines Zustandsautomaten an. Beachten Sie, wie das Einschalten der LZA bzw. das Ausschalten an den Normalbetrieb angeschlossen ist, so dass insgesamt eine sichere Signalfolge entsteht. Die blauen Kreise stellen jeweils Anfang bzw. Ende eines (Teil-)Graphen des Endlichen Automaten dar.



Die Zustände im Graph sind bereits geeignet binär codiert, so dass der Code unmittelbar zur Ansteuerung von LEDs verwendet werden kann:

- Die ersten beiden Bits (#7 + #6) geben die Betriebsphase an (01 = Start-Up, 10 = Shut-down, 00 = Normal Operations)

- Die Bits #5 - #3 geben die Farben Rot, Gelb, Grün der TL-1 an
- Die Bits #2 - #0 geben die Farben Rot, Gelb, Grün der TL-2 an

Sie können diese Codierung verwenden, um direkt die LEDs des STK600 (Port A) anzusteuern. Eine kleine Modell-Kreuzung mit farbigen LEDs als Ampeln wird im Labor zur Verfügung gestellt – deren Bit-Belegung ist wie oben beschrieben – also behalten Sie die Codierung unbedingt bei.

Programmieren Sie die Steuerung als Zustandsautomat mit 28 Zuständen mit expliziter Ablage der Zustandstabelle (siehe unten) als Datentabelle (keine Implementierung mit switches!). Überlegen Sie, wie Sie diese Tabelle geeignet implementieren (struct und/oder array. Realisieren Sie die Ampelfunktionen in einer **C++ Klasse „Ampel“**. Eine konkrete Ampel ist dann eine Instanz dieser Klasse; die Zustandstabelle kann als Klassenkonstante oder Instanzenkonstante implementiert werden. Worin liegt der Unterschied? Das Ampel-Objekt hat im Wesentlichen drei Instanzenmethoden für start_up(), shut_down_request() sowie next_state() für die Entgegennahme des Zeitsignals (welches dann den Zustandsübergang auslöst – hierzu mehr weiter unten).
Für die Generierung des Zeitsignals verwenden Sie einen Timer mit Grundtakt von 1 Sekunde, um die Zustandsübergänge auszulösen – in der Tabelle unten sind die Verweildauern für die einzelnen Zustände angegeben (hier kürzer als in der Realität, um schneller Testen zu können). Diese Zeiten sind einzuhalten; sie müssen aber jederzeit änderbar sein. Zur Programmierung des Timers bekommen Sie die C++ Klassen Timer8 und Timer16 aus der Bibliothek Runtime zur Verfügung gestellt, die Sie verwenden sollen.

Zur Verbindung des Timer-Objekts mit dem Ampel-Objekt folgende Hinweise: Das Timer-Objekt bekommt bei seiner Instanziierung u.a. eine sogenannte call-back Funktion übergeben. Das ist ein Zeiger p auf eine Prozedur ohne Parameter: void (*p)(void). Diese wird jedes Mal gerufen, wenn ein Timer-IRPT passiert. Prozedurzeiger können Zeiger auf globale Prozeduren oder auf Klassenmethoden sein; nur Adressen solcher Prozeduren können in einer Variablen oder einem Parameter gespeichert werden; niemals aber Zeiger auf Instanzenmethoden.
Problem: Wir haben ein Ampel-Objekt und möchten dessen Instanzenmethode next_state() bei Timer-IRPT aufrufen. C++ erlaubt es explizit nicht, die Adresse einer Instanzenmethode zu speichern (also auch nicht als Parameter zu übergeben).
Daher müssen wir den Aufruf der Instanzenmethode in eine globale Prozedur next_state() (gleicher Name wie Instanzenmethode – kein Problem) verpacken und dann diese als call-back verwenden. Das geht nach dem folgenden Schema:

```
// Ampel Objekt erzeugen:
Ampel dieAmpel(…);

// globale Prozedur; verpackt den Aufruf der Instanzenmethode:
static inline void next_state() {dieAmpel.next_state();};

// Timer Objekt für den Timer-Counter 1 erzeugen und die globale
// Prozedur next_state() als call-back registrieren:
Timer16  ticker(TC1, next_state);
```

Sehen Sie eine Taste des STK600 vor, um aus dem normalen Betrieb (und nur aus dieser Betriebsphase) das Herunterfahren anfordern zu können. Die LZA soll dann bis zum nächsten Erreichen des Zustandes „00010100“ (gelb / rot) weiterlaufen, um erst von dort aus in die Shut-down Phase über zu gehen (vgl. Zustandsgraph „Normal Operations“).

Die Verwendung eines LCDs wird Ihnen empfohlen, um durch Ausgabe der Betriebszustände des Automaten leichter Debuggen zu können. Auch hierfür bekommen Sie eine C++ Klasse als LCD-Treiber bereitgestellt. Hinweis: die LCD Ansteuerung hat in der call-back Routine next_state() (die ja eine ISR ist) nichts verloren, das kann nur im Hauptprogramm erfolgen!

Für die Programmierung von Timern, LCDs und digitalen Ports erhalten Sie jeweils C++ Klassen, die Sie verwenden sollen (siehe Bibliothek Runtime).

Für weitere Details lesen Sie bitte die Doxygen Dokumentation der Runtime Module nach!

Folgendes ist zu tun, um die Bibliothek an Ihr Projekt zu binden[3]:

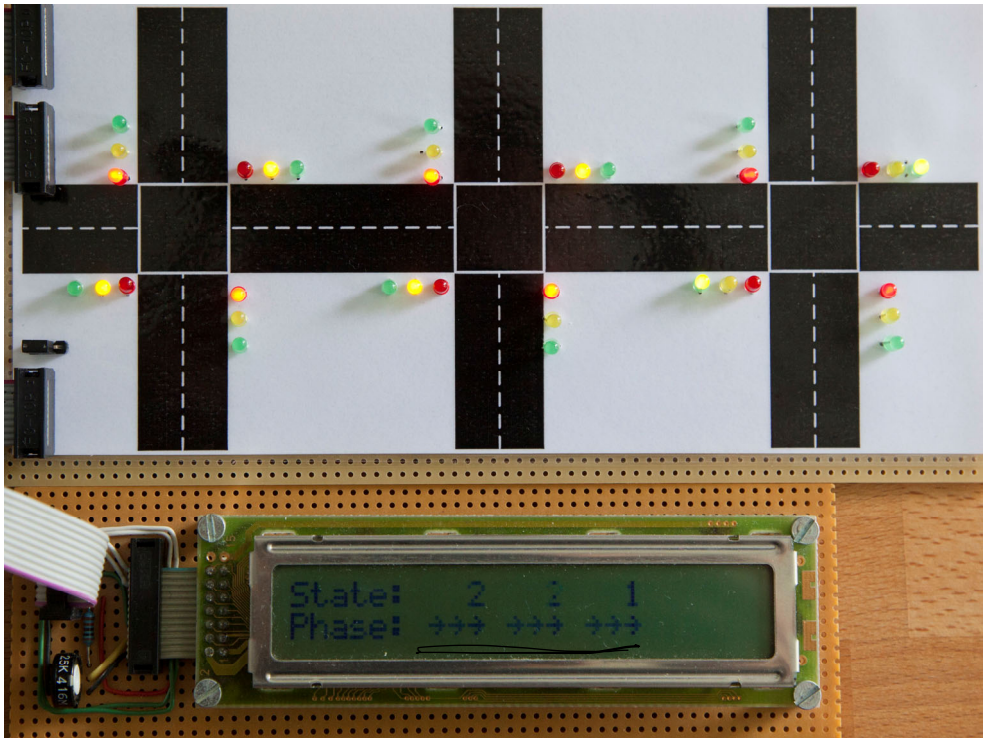
1) Gehen Sie im Solution Explorer auf Ihr Projekt und klicken Sie rechts. Wählen Sie „Add Library“. Es erscheint ein Fenster, in dem Sie „Browse Library“ anwählen und dann die Bibliotheksdatei libRuntime.a anwählen müssen (wo die steht sagt man Ihnen im Labor). Damit kann nun der Binder den Code aus der Bibliothek anbinden.
2) Gehen Sie im Solution Explorer auf Ihr Projekt und klicken Sie rechts. Wählen Sie „Properties“ und im erscheinenden Fenster „toolchain“. Wählen Sie dort „Directories“ unter „AVR/GNU C++ Compiler“. Mittels „add item“ geben Sie nun den Pfad zur Runtime an, in dem die Header stehen (wo die steht sagt man Ihnen im Labor). Damit kann nun der Präprozessor und der Compiler auf die Header Dateien zugreifen.

Im Verzeichnis „doxygen“ unter Runtime finden Sie eine umfangreiche online Dokumentation des Codes in Runtime. Lesen Sie hier nach, indem Sie mit dem Internet Browser die Datei index.html starten. Diese sollte als Icon auf dem Desktop der Labor-Rechner direkt verlinkt sein.

Aufgabenerweiterung:
Sobald Ihre Ampel funktioniert stellt sich die Frage: warum nicht gleich mehrere Ampeln instanziieren? Erweitern Sie Ihr Programm daher so, dass Sie 3 Ampel-Objekte instanziieren, die alle die gleiche Zustandstabelle nutzen. Überlegen Sie, ob sie alle mit dem gleichen Timer treiben wollen, oder je Ampel einen eigenen verwenden möchten. Wenn alle am gleichen Timer hängen laufen sie einfach nur zeitgleich. Bei individuellen Timern können Sie einen zeitlichen Versatz der Ampelzustände (durch unterschiedliche Startzeiten der Timer beim Hochfahren des Programms) erzeugen. Damit können Sie ein „grüne Welle“ erzeugen.
Passen Sie die LCD-Anzeige so an, dass sie die Betriebszustände aller drei Ampeln anzeigt. Sie bekommen ein Board mit 3 Kreuzungen entlang einer Straße mit je einer Ampel gestellt. Jede Ampel ist über einen eigenen Port und mit der eingangs erklärten Pinbelegung angeschlossen.

[3] Sofern Sie ein template-Projekt zur Erzeugung Ihres Projektes verwendet haben, sollten diese Schritte bereits erledigt sein.

Stellen Sie eine grüne Welle ein! Das STK600 benötigt hierfür externe Stromversorgung, USB genügt nicht mehr.
Zeichnen Sie ein UML Klassen- und Objektdiagramm Ihrer Gesamtlösung.



Lernziele (versuchen Sie für sich zu reflektieren, ob Sie die erreicht haben, sonst: nachfragen!):

- Sie können erklären was Sie tun müssen, um den Aufruf von Instanzenmethoden als Interrupt-Routine oder als call-back Routine zu ermöglichen.
- Sie können erklären (begründen), ob Sie auch mehrere Ampeln gleichzeitig betreiben könnten, wenn Sie die Instanzenmethode next_state() statt mit Zugriff auf die Tabelle als Abfolge von switch-statements geschrieben hätten.
- Sie können erläutern, ob und wie sich die Variante „3 Ampeln + gemeinsamer Timer“ von der Variante „3 Ampeln + 3 Timer“ hinsichtlich der zeitlichen Präzision unterscheiden mit der die Zustandsübergänge ausgeführt werden.
- Sie können ein UML-Klassen und Objekt Diagramm Ihres Programms zeichnen.

Cross-Roads Traffic Light State-Machine

No.	Phase	State	Time (sec)	TL-1	TL-2	
0	Normal Operations	00 100001	5	red	green	
1		00 100010	1	red	yellow	
2		00 110010	1	red-yellow	yellow	
3		00 110100	1	red-yellow	red	entry from Start-up
4		00 001100	5	green	red	
5		00 010100	1	yellow	red	exit to Shut-down
6		00 010110	1	yellow	red-yellow	
7		00 100110	1	red	red-yellow	
8	Start-up	01 010010	1	blink yellow	blink yellow	
9		01 000000	1	blink yellow	blink yellow	
10		01 010010	1	blink yellow	blink yellow	
11		01 000000	1	blink yellow	blink yellow	
12		01 010010	1	blink yellow	blink yellow	
13		01 000000	1	blink yellow	blink yellow	
14		01 010010	1	blink yellow	blink yellow	
15		01 000000	1	blink yellow	blink yellow	
16		01 010010	1	blink yellow	blink yellow	
17		01 100100	5	red	red	
18	Shut-down	10 100100	5	red	red	
19		10 010010	1	blink yellow	blink yellow	
20		10 000000	1	blink yellow	blink yellow	
21		10 010010	1	blink yellow	blink yellow	
22		10 000000	1	blink yellow	blink yellow	
23		10 010010	1	blink yellow	blink yellow	
24		10 000000	1	blink yellow	blink yellow	
25		10 010010	1	blink yellow	blink yellow	
26		10 000000	1	blink yellow	blink yellow	
27		10 010010	1	blink yellow	blink yellow	

eMail !!!

Aufgabe 5: Multitasking-System mit inter-Task-Kommunikation über mittels Semaphoren gesicherte Queues

In der vorigen Aufgabe war quasi-Nebenläufigkeit durch eine oder mehrere Timer-gesteuerte call-back Routine(n) implementiert worden. Es gab mehrere Ausführungsstränge: das Hauptprogramm, welches „Hintergrundaktivitäten" ausführt und jede call-back-Routine, die exakt zeitlich gesteuert (durch Timer-IRPT) die Zustandsübergänge ihres Zustandsautomaten bewirkt. Zwischen diesen Ausführungssträngen wurden höchstens geringe Datenmengen über globale Variablen (hier: globale Objekte) ausgetauscht. Da wir in diesem Fall einen gering ausgelasteten single-core Prozessor haben, konnten wir einigermaßen sicher sein, das die Kommunikation zwischen den beiden Ausführungssträngen nicht „schief gehen" wird und kritische Aktivitäten nicht gestört werden: eine im ISR-Kontext ausgeführte call-back Routine kann (beim ATMEL üblicherweise) sowieso nicht unterbrochen werden - sie führt den kritischen Zustandsübergang ungestört durch und steuert die Ampel-Lichtzeichen - und das Hauptprogramm erledigt nur noch nachgeordnete Aufgaben (lesen Sie nochmal die Definition von **foreground** und **background** in der RUM Vorlesung nach!). Sollte hierin eine Störung durch einen IRQ auftreten, wäre schlimmstenfalls die LCD-Anzeige kurzzeitig verzögert.

In komplexeren Systemen kommen viele Ausführungsstränge vor, die nicht notwendigerweise direkt von IRQs gesteuert werden, und diese werden ggf. auch größere Datenmengen austauschen wollen bei garantierter Unversehrtheit der Daten, oder andere Ressourcen gemeinsam, aber ungestört nutzen können. In der Vorlesung RUM wurden hierzu einige grundlegende Verfahren erklärt: **Semaphore** zur Kontrolle von Zugriffen zu gemeinsamen Ressourcen, **Kritische Abschnitte**, und **Queues** (Warteschlangen für die Datenübertragung). Ferner haben wir dort den **Scheduler** als die Instanz kennen gelernt, die die einzelnen **Tasks** zeitweise zur Ausführung bringt und auch wieder suspendiert (pre-emptive scheduling). Lesen Sie diese Abschnitte in der Vorlesung RUM unbedingt nochmal nach, bevor Sie weitermachen – Sie müssen die Konzepte verstanden haben, um sie jetzt anzuwenden!

Für die Aufgabe 5 erhalten Sie aus der Runtime zur Verfügung gestellt: je eine C++ Klasse für Binäre Semaphore (BinarySemaphore), Queues (BoundedQueue) sowie ein Modul, das einen Multi-tasking Betriebssystemkern implementiert (OSKernel). Aus der vorangehenden Aufgabe kennen Sie schon die Klassen für LCDs, Timer und DigiPorts.

Erläuterungen zu OSKernel:
Tasks sind im Grunde endlos laufende Prozeduren (Endlosschleifen) ohne Parameter. Sie werden beim Betriebssystemkern registriert, nach Übergang in den multi-tasking Modus gestartet und dann endlos reihum ausgeführt. Dabei bekommt jeder Task eine limitierte Zeitscheibe zugestanden (hier: 15ms). Wenn der Task nicht von selbst die Kontrolle vor Ablauf seiner Zeitscheibe zurückgibt, wird er gewaltsam suspendiert. Programmierschema für 3 Tasks:

```
// definiere 3 Tasks:
void task_1(void) { do { … //do task_1 stuff }while(1); }
void task_2(void) { do { … //do task_2 stuff }while(1); }
void task_3(void) { do { … //do task_3 stuff }while(1); }
```

```
// in der Prozedur main:
// registriere die 3 Tasks beim Betriebssystem mit
// unterschiedlicher Priorität (default = High):
task_insert (task_1);
task_insert (task_2, Medium);
task_insert (task_3, Low);
// Starte den Betriebssystem-Kern (schalte in multi-tasking
// Mode um); diese Prozedur kehrt nie zurück:
kernel();
```

Das Betriebssystem implementiert einen einfachen Prioritätsmechanismus für die Tasks. Jeder Task kann als Priorität `High`, `Medium` oder `Low` haben (High ist default, wenn man bei `task_insert()` den zweiten Parameter nicht angibt). Die Priorität gibt an, wie oft ein Task relativ zu den anderen Tasks bei der reihum-Aktivierung drankommt: ein high-prio Task kommt in jeder Runde dran, ein medium-prio Task nur in jeder zweiten Runde, ein low-prio Task nur in jeder vierten Runde. Beachte: dies macht keine Aussage über die tatsächliche Ausführungszeit pro Task. Wie häufig ein Task pro Sekunde drankommt, hängt nur von der durch die anderen Tasks verursachten Prozessorlast ab. Die Priorität gibt also nur die relative Aktivierungshäufigkeit, nicht aber die tatsächliche Häufigkeit pro Sekunde oder gar Laufzeiten bzw. Wartezeiten vor.
Jeder Task bekommt eine eindeutige Task-ID zugewiesen. Diese kann als Rückgabewert der Funktion `task_insert()` abgegriffen werden. Tasks können ihre eigene ID auch mittels der Funktion `current_task_id()` abfragen. Einige Prozeduren des Betriebssystems zur Beeinflussung von Tasks benötigen diese ID als Parameter, bspw. wenn die Priorität eines Tasks verändert werden soll, oder Tasks deaktiviert und wieder aktiviert werden sollen.

Erläuterungen zu BinarySemaphore:
Binäre Semaphore können belegt oder frei sein, also 2 Zustände haben (sie können darüber hinaus nicht zählen). Um Fehler durch falsche Benutzung zu minimieren wird jeweils die Task-ID des Tasks gespeichert, der das Semaphor momentan besitzt. Nur dieser Besitzer kann ein Semaphor wieder freigeben. Die Semaphor-Methoden sind als kritische Abschnitte implementiert, mithin also sicher im Multi-tasking Modus benutzbar.

Erläuterungen zu BoundedQueue:
Die Klasse BoundedQueue stellt Ringpuffer fester Länge zur Verfügung. Je Speicherplatz kann ein unsigned char (=uint8_t) abgelegt werden. Die Zugriffsfunktionen sind als kritische Abschnitte implementiert und somit Multi-tasking sicher. Aber: sollen zusammengehörende Folgen von Zeichen (strings) ungestört in eine Queue geschrieben oder von ihr gelesen werden, müssen diese Queue-Zugriffe durch ein zusätzliches Semaphor abgesichert werden. Schreiber müssen String-Anfang und -Ende irgendwie kenntlich machen, um ihre Daten von den Daten anderer Schreiber unterscheidbar zu machen.

Normalerweise verliert die Queue keine Daten – sie kann aber volllaufen, wenn der Erzeuger der Daten schneller ist als der Leser. Dann nimmt die Schreibemethode keine Daten mehr entgegen. Da es auch Systeme gibt, bei denen Daten verloren gehen dürfen, kann die Queue (per Parameter im Konstruktor) auch so konfiguriert werden, dass die jeweils ältesten noch

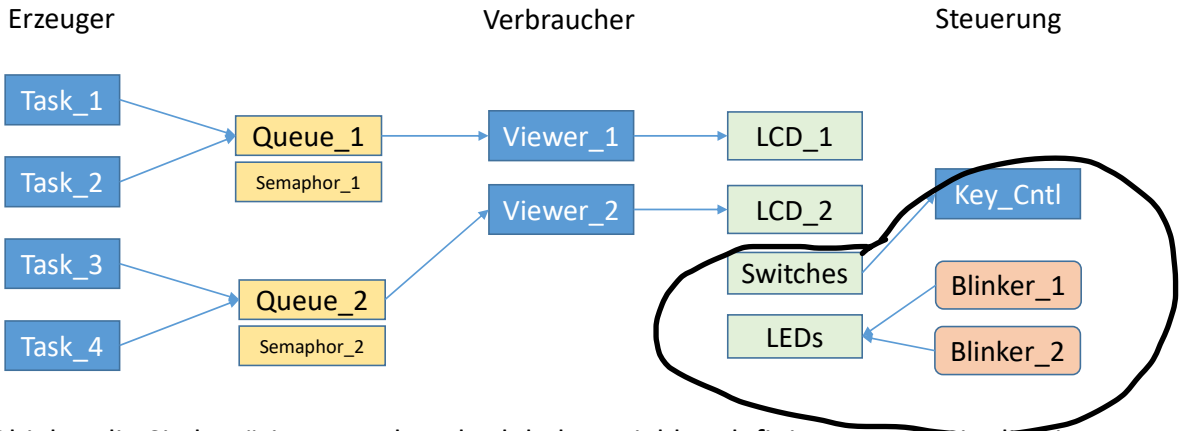
nicht gelesenen Daten einfach mit neuen Daten überschrieben werden. Ein langsamer Leser sieht dann immer nur die jeweils neuesten Daten.

Für weitere Details lesen Sie bitte die Doxygen Dokumentation der drei Module nach!

Aufgabe:
Im Rahmen dieser Aufgabe sollen Sie nun ein Erzeuger-Verbraucher System aus sieben Tasks (vier Erzeuger, zwei Verbraucher, eine Steuerung) implementiert werden. Außerdem sollen 2 rein Timer-gesteuerte Prozeduren, welche LEDs in unterschiedlichen Frequenzen blinken lassen, implementiert werden. Sie haben also in Summe neun nebenläufige Ausführungsstränge, von denen 7 vom Scheduler und 2 von Timern kontrolliert werden. Ein „Hauptprogramm" gibt es nicht mehr, denn das wird nach Start des multi-tasking Modes (Aufruf `kernel()`) gar nicht mehr weiter geführt – es dient lediglich der Initialisierung. Die eigentlichen Funktionen des Systems sind über die 9 Ausführungsstränge (blau und orange im Bild unten) und deren Zusammenwirken verteilt.

Die Erzeuger- und Verbraucher-Tasks sollen über zwei Queues und zugeordnete Semaphore gekoppelt sein (gelb). Die Verbraucher Tasks (Viewer_1 und Viewer_2) lesen die Daten der Erzeuger (Tasks 1 bis 4) und stellen diese auf 2 zweizeiligen LCDs jeweils eine Zeile pro Task dar (verwenden Sie den freien Port B für das zweite LCD). Wenn also eine Nachricht von Task_1 bei Viewer_1 ankommt, stellt dieser die Nachricht auf Zeile 1 von LCD_1 dar. Mit dem Task Key_Cntl sollen die Tasten gelesen werden. Es soll jeweils eine Taste als Start/Stop Funktion für einen der Tasks 1 bis 4 dienen; der Benutzer kann also die 4 Erzeuger-Tasks einzeln an und ausschalten. Vier weitere Tasten sollen die Blinkfrequenz der beiden Blinker schneller bzw. langsamer stellen können.

Das folgende Bild zeigt die Struktur (Datenflüsse) der Aufgabenstellung:



Alle Objekte die Sie benötigen werden als globale Variablen definiert. Lassen Sie die vier Erzeuger-Tasks eindeutig unterscheidbare Strings an die Viewer schicken, damit die korrekte (ungestörte) Übertragung leicht ersichtlich ist.

Beachten Sie: **der Scheduler verwendet den Timer TC0 – der steht Ihnen nicht zur Verfügung**!

Wenn Sie 2 LCDs anschließen und LEDs nutzen genügt die Stromversorgung über USB nicht mehr (eine LED auf dem STK600 blinkt dann rot; kaputtgehen kann aber nichts). Sie müssen in diesem Fall eine externe Stromversorgung ans STK600 anschließen.

Bauen Sie Ihre Lösung schrittweise auf, so dass Sie die Teile jeweils testen können. Bspw.: zunächst nur Task_1, Queue_1 und Viewer_1. Dann erweitern Sie auf Task_2 und Semaphor_1. Wenn das läuft können die das Schema duplizieren (Task_3 + 4, Queue_2 etc)

Diese Anwendung kann nicht mehr sinnvoll im Simulator getestet werden. Sie müssen also Ihre Lösung so gut wie möglich zuhause vorbereiten, fehlerfrei übersetzen und dann im Labor weiterarbeiten.

Wenn Ihre Lösung grundsätzlich läuft: variieren Sie die Einstellung der Queues (Überschreiben der ungelesenen Daten erlauben) und verändern Sie die Prioritäten der Tasks – überlegen Sie zunächst welche Prioritäten die Tasks sinnvoller Weise haben sollten. Was passiert im System, wenn Sie die Prioritäten anders einstellen? Was passiert, wenn Sie die Semaphor-Nutzung ausklammern? Wie wirken sich yield() Aufrufe (oder deren Weglassen) aus?

Optional: im Labor verwenden wir neuerdings auch 4-zeilige LCDS. Damit kann man die beiden 2-zeiligen ersetzen. Wie würden Sie Ihr Programm ändern, um die 4 Datenströme auf die 4 Zeilen dieses Displays zu bringen? Gibt es Alternativen und welche Vor-/Nachteile haben diese?

Lernziele (versuchen Sie für sich zu reflektieren, ob Sie die erreicht haben, sonst: nachfragen!):

- Sie können begründen, warum Sie welche Prioritäten für die Tasks gewählt haben.
- Sie können erklären, warum bei sehr schnellen Schreibern trotzdem kein Text auf dem Weg zu den Lesern verloren geht, insbesondere die Rolle des Semaphors dabei können Sie erklären.
- Sie können erklären wozu kritische Abschnitte benötigt werden und was bei deren Verwendung beachtet werden sollte.