Benjamin Menhorn
Kilian Kempf

Labor Eingebettete Systeme

Institut für Eingebettete Systeme und Echtzeitsysteme

Sommersemester 2013

Inhaltsverzeichnis

1 VHDL Wiederholung . . . 1
- 1.1 Latches, Flip-Flops, Counter . . . 1
- 1.2 Finite-State-Machines . . . 6

2 Pulsweitenmodulation . . . 11

3 Wiederholung Qsys und Nios II . . . 17

4 Memory-Mapped-IO . . . 23

5 Debugging . . . 29
- 5.1 Hardware-Debugging . . . 29
- 5.2 Software-Debugging . . . 30

6 Treiberentwicklung . . . 33
- 6.1 Interrupts . . . 33
- 6.2 Treiber für den Altera-HAL . . . 37

7 Mini-Betriebssystem . . . 49

8 MicroC/OS-II . . . 53

9 Lüfterregelung . . . 57

Kapitel 1

VHDL Wiederholung

1.1 Latches, Flip-Flops, Counter

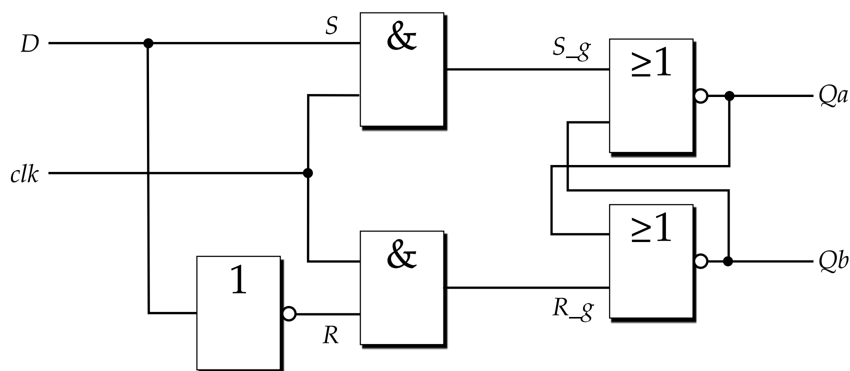

Abb. 1.1 Schaltkreis eines Gated D-Latch

Abbildung 1.1 zeigt den Schaltkreis eines gated D-Latches. In Listing 1.1 wird exemplarisch ein Stück VHDL-Code vorgestellt, der den abgebildeten Schaltkreis in Form von Logikausdrücken realisiert.

Die gleiche Funktionalität lässt sich mit Hilfe eines `PROCESS` realisieren, der sensitiv auf die Signale `D` und `clk` ist. Der entsprechende VHDL-Code ist in Listing 1.2 wiedergegeben.

```vhdl
-- A gated D latch described the hard way
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY d_latch IS
 PORT (
     clk, D   : IN STD_LOGIC;
     Q   : OUT STD_LOGIC
     );
END d_latch;

ARCHITECTURE Structural OF d_latch IS
 SIGNAL R, S, R_g, S_g, Qa, Qb : STD_LOGIC ;

BEGIN
 S <= D;
 R <= NOT D;
 R_g <= R AND clk;
 S_g <= S AND clk;
 Qa <= NOT (R_g OR Qb);
 Qb <= NOT (S_g OR Qa);
 Q <= Qa;
END Structural;
```

Listing 1.1 Gated D-Latch (kombinatorisch)

```vhdl
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY d_latch IS
 PORT (
      D, clk   : IN STD_LOGIC ;
      Q    : OUT STD_LOGIC
     );
END d_latch ;
ARCHITECTURE Behavior OF d_latch IS
BEGIN
 PROCESS ( D, clk )
 BEGIN
  IF clk = '1' THEN
   Q <= D ;
  END IF ;
 END PROCESS ;
END Behavior ;
```

Listing 1.2 Gated D-Latch (Prozess)

Aufgabe 1

Abbildung 1.2 zeigt nun einen Schaltkreis mit drei verschiedenen Speicherelementen. Neben einem gated D-Latch sind auch ein D-Flipflop mit positiver Taktflanke sowie eines mit negativer Taktflanke vorhanden. Aus Abb. 1.3 können die Signale an den Ausgängen abhängig von den Eingangssignalen `Clock` und `D` entnommen werden.

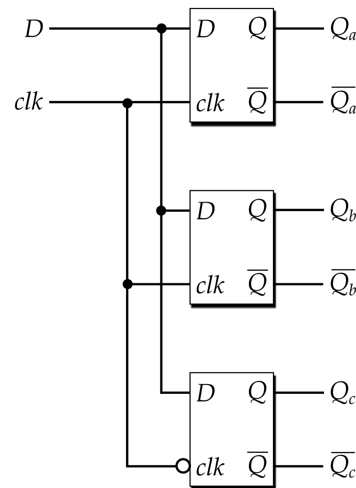

Abb. 1.2 Schaltkreis

1. Implementieren Sie den abgebildeten Schaltkreis. Machen Sie sich mit dem Konstrukt `PROCESS` vertraut. Beachten Sie besonders das Schlüsselwort `event` im Zusammenhang mit (Takt-)Signalen.
2. Schreiben sie eine VHDL-Datei, welche die drei verschiedenen Speicherelemente instanziiert und implementieren Sie die Elemente als Komponenten. Verwenden Sie dazu wie in Listing 1.2 jeweils einen `PROCESS`.

Aufgabe 2

In dieser Aufgabe sollen Sie ModelSim von Altera verwenden, um Ihre Komponenten zu testen. Starten Sie hierzu ModelSim und legen Sie ein neues Projekt an. Als Projektverzeichnis können Sie das Verzeichnis wählen, in welchem Sie die Komponenten und die Top-Level Entity gespeichert haben. Fügen Sie nun dem Projekt Ihre VHDL-Dateien hinzu.

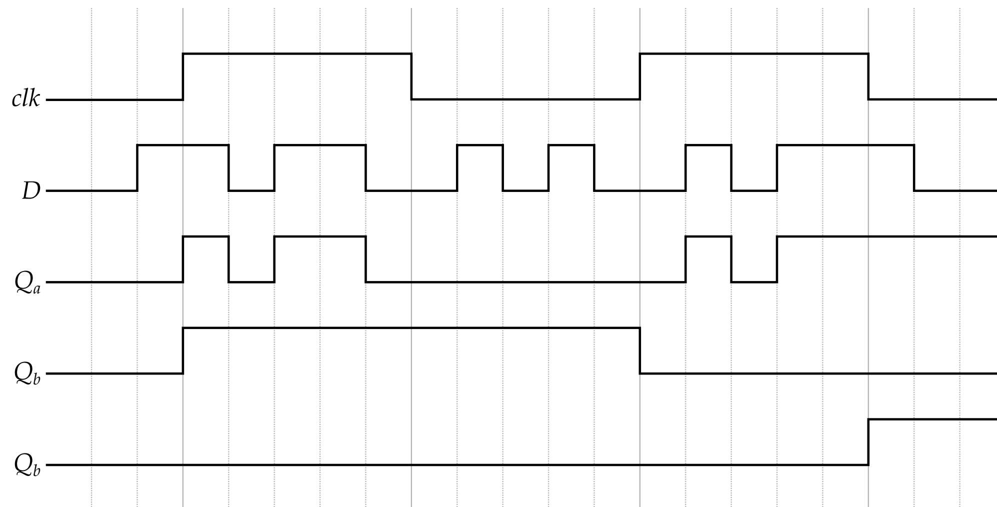

Abb. 1.3 Timing-Diagramm

1. Im ersten Schritt müssen Sie ihre Dateien compilieren (`Compile` → `Compile All`). Anschließend können Sie unter `Simulate` → `Start Simulation...` die Top-Level Entity Ihres Projektes auswählen. Unter `View` → `Objects` sehen Sie die Signale Ihres Projektes und unter `View` → `Wave` wird der zeitliche Verlauf der Signale angezeigt.

2. Um nun die Belegung der Ausgabesignale zu sehen, müssen an die Eingabesignale Werte angelegt werden. Hierzu wählen Sie bei den Objekten einzeln das jeweilige Eingangssignal (hier `Clock` und `D`) aus. Durch einen Rechtsklick können sie nun mit `Create Wave...` die Eingabewerte anpassen. Passen Sie den Input für das Eingabesignal `D` so an, dass sich ein ähnliche Verlauf wie in Abb. 1.3 ergibt. So können Sie am einfachsten verifizieren, dass Ihre Komponenten korrekt arbeiten. Die Ausgangssignale, welche Sie beobachten wollen, müssen Sie nach dem Auswählen über `Add` → `To Wave` → `Selected Signals` hinzufügen.

3. Simulieren Sie nun Ihr Projekt über `Simulate` → `Run` → `Run-All`. Vergleichen Sie Ergebnis mit Abb. 1.3. Beachten Sie, dass eine erneute Simulation im zeitlichen Verlauf jeweils angehängt wird. Es bietet sich daher an, über `Simulate` → `Run` → `Restart...` die Simulation zurückzusetzen.

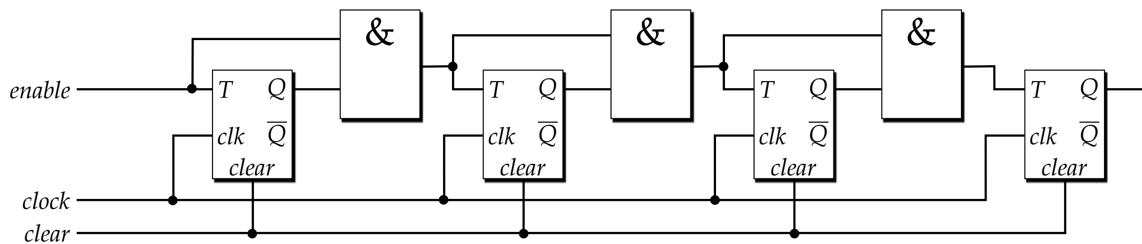

Abb. 1.4 4-Bit-Zähler

Aufgabe 3

Der Schaltkreis in Abb. 1.4 zeigt einen synchronen 4-Bit-Zähler, der aus vier T-Flipflops aufgebaut ist. Ist das Enable-Signal gesetzt, so erhöht er seinen Zählerstand bei jeder positiven Taktflanke. Mit dem Reset-Signal „Clear“ lässt sich der Zähler wieder auf Null zurücksetzen.

Ein T-Flipflop wechselt seinen Zustand („toggle“) bei jeder positiven Taktflanke, solange an „T“ ein High-Pegel anliegt, ansonsten wird der gespeicherte Zustand gehalten. Statt eines Dateneingangs besitzt es einen Clear-Eingang, mit dem der Speicher auf Null zurückgesetzt werden kann.

Implementieren Sie auf diese Weise einen 16-Bit-Zähler:

1. Erstellen sie ein T-Flipflop (als Komponente). Falls gewünscht, können Sie dafür auf ein bereits implementiertes Speicherelement zurückgreifen.
2. Schreiben sie eine VHDL-Datei, welche den 16-Bit-Zähler mit Hilfe der in Abb. 1.4 gezeigten Struktur umsetzt. Simulieren sie den Schaltkreis.
3. Erweitern Sie ihren Code so, dass der Taster `KEY0` (Achtung: active-low) als Takteingang und die Schalter `SW1` und `SW0` als Enable und Reset dienen. Benutzen Sie die 7-Segment-Anzeigen `HEX3-0`, um hexadezimal den Zählerstand auszugeben. Erstellen Sie hierfür einen Decoder (Komponente), der aus einem 4-bittigen Eingang die Ansteuerung einer Segmentanzeige erzeugt.

Abbildung 1.5 zeigt, welches Bit des Vektors einer 7-Segment-Anzeige zu der jeweiligen LED gehört.

Aufgabe 4

Vereinfachen Sie ihren Code so, dass die Spezifikation des Zählers auf dem VHDL-Ausdruck

```
1 Q <= Q + 1;
```

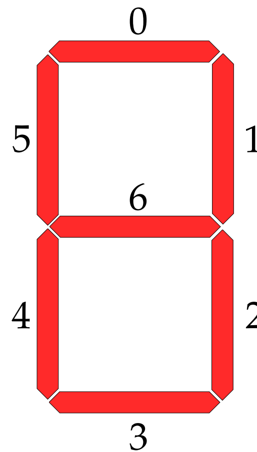

Abb. 1.5 7-Segment-Anzeige

basiert. Um das Pluszeichen verwenden zu könnten, muss am Anfang der Datei zusätzlich über

```
USE ieee.std_logic_unsigned.all;
```

die vorzeichenlose Arithmetik aus der Bibliothek `ieee` geladen werden. Erstellen sie wieder einen 16-Bit-Zähler und überprüfen Sie seine Funktion.

Aufgabe 5

Entwerfen und implementieren Sie einen Schaltkreis, welcher der Reihe nach die Ziffern 0 bis F auf der 7-Segment-Anzeige HEX0 ausgibt. Dabei soll jede Ziffer etwa eine Sekunde lang angezeigt werden. Benutzen Sie einen Zähler, um die Sekunden-Intervalle zu erzielen. Alle vorhandenen Zähler sollen dabei vom auf dem Board vorhandenen 50 MHz Takt gespeist werden. Dieser ist an einen Pin mit der Bezeichnung `CLOCK_50` angebunden, der wie einer der bekannten Schalter-Eingänge verwendet werden kann.

1.2 Finite-State-Machines

Für den Entwurf und die Beschreibung von digitalen Systemen bilden Zustandsautomaten (Finite State Maschines; FSMs) eine wesentliche Grundlage. Mit Zustandsautomaten werden zyklische

Funktionsabläufe realisiert, sie steuern andere Logikschaltungen und in komplexen digitalen Systemen werden sie zur Synchronisation mehrerer Komponenten eingesetzt. Zustandsautomaten sind sequenziell arbeitende Logikschaltungen, die gesteuert durch ein periodisches Taktsignal eine Abfolge von Zuständen zyklisch durchlaufen.

aus: Reichardt, Schwarz, VHDL-Synthese, 4.Auflage

Aufgabe 5

In diesem Teil soll ein Zustandsautomat implementiert werden, der zwei spezifische Sequenzen von Eingangssymbolen erkennen kann. Einerseits vier aufeinander folgende Nullen, andererseits vier Einsen. Als Eingang dient das Signal w, als Ausgang das Signal z. Immer wenn für vier aufeinander folgende Clock-Impulse (hier: steigende Flanken) w=0 oder aber w=1 war, dann soll z auf 1 sein, ansonsten auf 0. Dies soll auch für sich überlappende Sequenzen gelten. Wenn also fünf Clock-Impulse lang w=1 gilt, dann soll z nach dem vierten und dem fünften Impuls auf 1 stehen. Der geforderte Zusammenhang zwischen w und z ist noch einmal in Abb. 1.6 zu sehen.

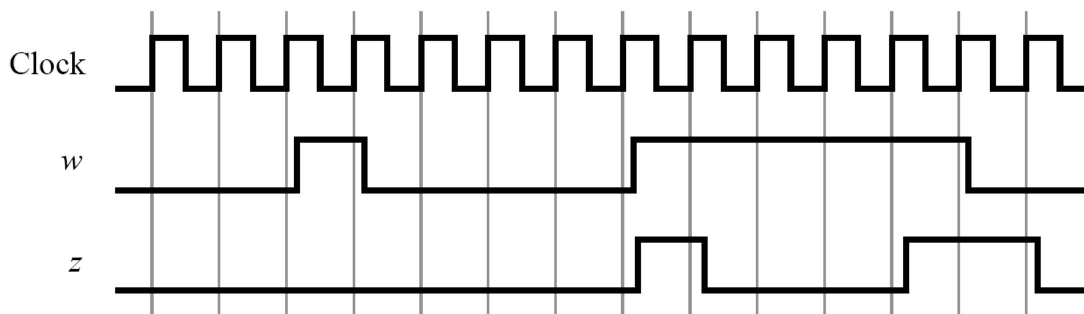

Abb. 1.6 Timing für den Ausgang z

Der entsprechende Zustandsautomat (ein Moore-Automat) wird in Abb. 1.7 gezeigt.

VHDL bietet eine Möglichkeit, einen Zustandsautomaten so zu spezifizieren, dass er vom Compiler und den Synthesewerkzeugen erkannt und entsprechend umgesetzt wird. Innerhalb eines PROCESS wird dabei der aktuelle Zustand mittels CASE abgefragt und dann der jeweils nächste Zustand festgelegt. Dabei sind zwei verschiedene Signale (Vektoren) zu verwenden, von denen eines den aktuellen Zustand bereithält, während in das andere der gewünschte nächste Zustand geschrieben

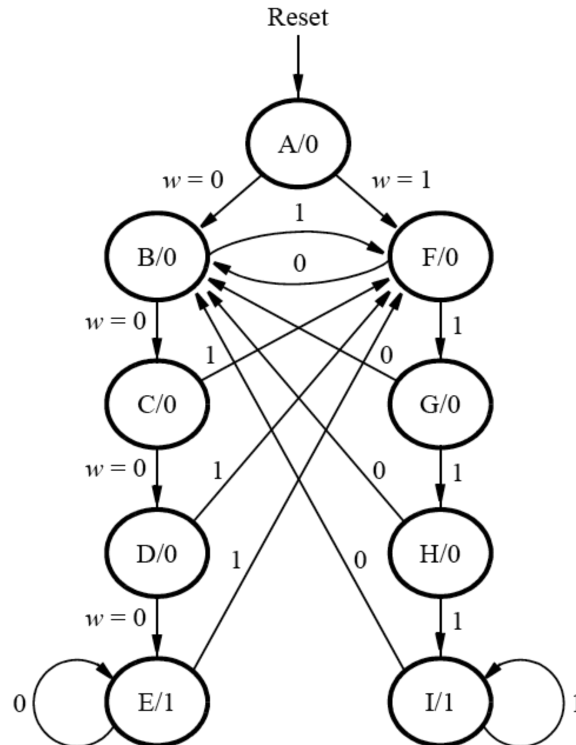

Abb. 1.7 Zustandsautomat für die Sequenzerkennung

wird. In einem zweiten Prozess wird dann abhängig von einem Taktsignal der momentane Zustand aktualisiert. Listing 1.3 bietet ein entsprechendes Gerüst aus VHDL-Code.

Die Codierung der Zustände in Binärwerte wird vom Synthesewerkzeug automatisch erledigt, der Code selbst enthält nur die Namen der Zustände.

Entwerfen und implementieren Sie nun einen Zustandsautomaten, der die oben erwähnten Sequenzen erkennt.

1. Schreiben Sie eine entsprechende VHDL-Datei. Nutzen Sie den Schalter SW0 als synchronen active-low Reset für den Zustandsautomaten, SW1 als Eingang w und den Taster KEY0 (Achtung: active-low) als manuellen Clock-Eingang. Benutzen Sie die grüne LED LEDG0 als Anzeige für den Ausgang z und die neun roten LEDs LEDR8 bis LEDR0 um den aktuellen Zustand auszugeben.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY decoder IS
 PORT (
    ... define inputs and outputs
     );
END decoder;

ARCHITECTURE Behavior OF decoder IS

    ... declare signals

TYPE State_type IS (A, B, C, D, E, F, G, H, I);
SIGNAL y_Q, Y_D : State_type;  -- y_Q is present state, Y_D is next state

BEGIN

  ...

PROCESS (w, y_Q)  -- state table
BEGIN
  case y_Q IS
 WHEN A => IF (w = '0') THEN Y_D <= B;
    ELSE  Y_D <= F;
    END IF;

   ... other states

 END CASE;
END PROCESS;  -- state table

PROCESS (Clock)
BEGIN
   ...
END PROCESS;

   ... assignments for output z and the LEDs

END Behavior;
```

Listing 1.3 VHDL-Code für einen Zustandsautomaten

2. Untersuchen Sie die von Quartus II erzeugte Schaltung mit dem RTL-Viewer. Schauen Sie sich auch den erzeugten Zustandsautomaten an, und stellen Sie sicher, dass er dem Automaten in Abb. 1.7 entspricht. Beachten Sie ebenfalls die Codierung der Zustände.
3. Führen Sie eine funktionale Simulation der Schaltung durch.
4. Testen Sie die Schaltung auf dem DE2-Board. Stellen Sie sicher, dass der Automat die richtigen Zustandsübergänge benutzt (z. B. mit Hilfe der roten LEDs.)

Aufgabe 6

Anstatt der formalen Lösung oben soll nun die selbe Sequenzerkennung über Schieberegister durchgeführt werden. Schreiben Sie dafür einen VHDL-Code, der zwei 4-Bit-Schieberegister verwendet, eins für die vier Nullen und eins für die vier Einsen. Es steht Ihnen dabei frei, ob sie die Schieberegister selbst implementieren (in einem PROCESS), oder auf Alteras Megafunction-Library zurückgreifen; der Aufwand ist in beiden Fällen vergleichbar gering. Entwerfen Sie die entsprechende Schaltungslogik, um den Ausgang z anzusteuern. Die Schalter, Taster und LEDs sollen wie beim vorherigen Teil verwendet werden. Beobachten Sie das Verhalten der Schieberegister und des Ausgangs z.

Kapitel 2

Pulsweitenmodulation

Die sogenannte Pulsweitenmodulation (kurz PWM) ist ein Rechtecksignal mit konstanter Periodendauer, das zwischen zwei verschiedenen Spannungspegeln oszilliert. Prinzipiell wird das Signal also in schneller Folge ein- und ausgeschaltet. Das Verhältnis von Einschaltzeit zu Ausschaltzeit kann dabei variieren und bildet das Tastverhältnis (den Duty-Cycle).

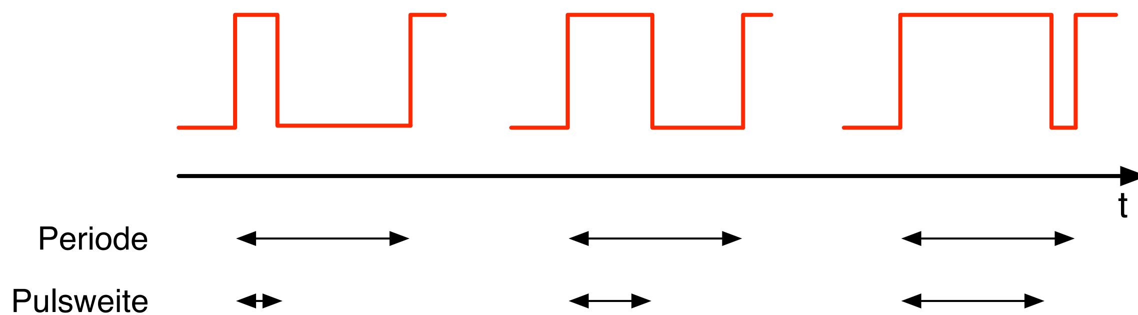

Abb. 2.1 Pulsweitenmodulation

Wie in der Skizze zu sehen ist, bleibt die Periodendauer / Frequenz konstant und nur die Pulsweite ändert sich.

Dieses Signal muss nun demoduliert werden, um daraus eine nutzbare Größe zu erhalten. Das geht z. B. mit einem Tiefpassfilter, mit dem ein einfacher Digital-Analog-Wandler entstehen würde.

Die mittlere Spannung berechnet sich dabei durch

$$U_m = U_{aus} * (U_{ein} - U_{aus}) * \frac{t_{ein}}{t_{ein} + t_{aus}}$$

Da in den meisten Fällen direkt zwischen 0 V und Vcc umgeschaltet wird, lässt sich die Formel in

$$U_m = V_{cc} * \frac{t_{ein}}{t_{ein} + t_{aus}} = V_{cc} * DC$$

vereinfachen. Wie man sieht, ist der Mittelwert der Spannung direkt vom Duty-Cycle DC abhängig.

Träge Verbraucher, wie beispielsweise Glühlampen und Elektromotoren, können direkt an ein PWM-Signal angeschlossen werden und auf diese Weise gedimmt bzw. in ihrer Drehzahl beeinflusst werden. Hingegen wird bei LEDs die Demodulation in diesem Fall durch das Auge vorgenommen, das entsprechend träge bei der Verarbeitung der eintreffenden Lichtsignale ist.

Aufgabe 1

In diesem Teil soll die Helligkeit von LEDs über eine PWM gesteuert werden, da sich LEDs nicht (richtig) über die Spannung dimmen lassen.

Erstellen Sie eine Komponente, die eine PWM mit 8 Bit Auflösung realisiert. Für den Duty-Cycle sollen also Werte von 0x00 bis 0xFF übergeben werden können. Sie kommen dafür mit einem einfachen Zähler und einem Komparator (realisierbar mit einem Vergleichsoperator) aus. Achten Sie darauf, dass die Komponente taktsynchron arbeitet. Dabei sollen wieder alle Einheiten der Schaltung direkt vom 50 MHz Takt CLOCK_50 gespeist werden.

Die PWM-Komponente soll über Eingänge für den Takt und den Duty-Cycle sowie über einen Ausgang für das PWM-Signal verfügen.

Es sollen vier Instanzen der PWM jeweils eine grüne LED treiben. Verwenden Sie bei dreien einen festen Duty-Cycle von 25 %, 50 % und 75 %. Der Duty-Cycle der vierten PWM soll über Schiebeschalter frei eingestellt werden können.

Führen Sie eine Timing-Simulation durch und stellen Sie ein „sauberes“ Verhalten ihrer PWM-Signale sicher.

Programmieren Sie die Schaltung auf das Board und beobachten Sie das Verhalten der LEDs, insbesondere bei der PWM mit dem veränderlichen Duty-Cycle.

Aufgabe 2

Berechnen Sie die Frequenz der von Ihnen erstellten PWM. (Ist sehr einfach.) Überprüfen Sie Ihr Ergebnis durch die Simulation.

Aufgabe 3

Nun soll es ermöglicht werden, die Frequenz der PWM frei einzustellen. Dazu muss die Dauer bis zum Überlauf des Zählers verändert werden können. Wird beispielsweise der Zählraum bis zum Überlauf verdoppelt, so halbiert sich die Frequenz der PWM. Durch freie Einstellbarkeit der Überlaufbedingung lassen sich auch Zwischenfrequenzen mit akzeptabler Genauigkeit erzeugen. Die Vergleichsbedingung am Komparator muss natürlich entsprechend angepasst werden, da sonst der Duty-Cycle nicht mehr stimmt.

Im Folgenden sollen weiterhin 8 Bit Auflösung für den Duty-Cycle verwendet werden. Der Zeitpunkt des Überlaufs soll mit 16 Bit frei einstellbar sein. Sie müssen also den Port Ihrer PWM-Komponente um ein einsprechendes Eingabesignal erweitern.

Stellen Sie eine Formel auf, welche die 8 Bit Eingabe des Duty-Cycle auf die Periodendauer des Zählers bis zum Überlauf abbildet. *Tipp:* Es handelt sich um eine einfache lineare Skalierung. Sie benötigen dazu neben dem 8 Bit Wert des Duty-Cycle den 16 Bit Wert für den Zeitpunkt des Zählerüberlaufs als Eingaben. Beachten Sie die Bitbreite von Zwischenergebnissen und stellen Sie sicher, dass der Ergebniswert wieder 16 Bit breit ist, um als Eingabe für den Komparator dienen zu können.

Führen Sie die entsprechenden Erweiterungen Ihrer PWM-Komponente durch und implementieren Sie die erstellte Formel in Hardware, um die freie Einstellbarkeit der PWM-Frequenz zu erreichen.

Führen Sie Timing-Simulationen durch, um Ihre Implementierung (Frequenz und Duty-Cycle) zu überprüfen. Stellen Sie beispielsweise eine PWM-Frequenz von 30 kHz ein.

Aufgabe 4

Die PWM-Komponente soll nun frei programmierbar werden. Da das Board jedoch nur über 18 Schiebeschalter verfügt, müssen die Werte für den Teilfaktor (an welcher Stelle der interne Zähler

überläuft) und den Duty-Cycle in Registern zwischengespeichert werden. Der Zugriff auf die Komponente soll deshalb über einen 8 Bit breiten Datenbus erfolgen, wobei die Datenleitungen für den Lese- und den Schreibzugriff getrennt sein dürfen. Eine Übersicht finden Sie in Abbildung 2.2.

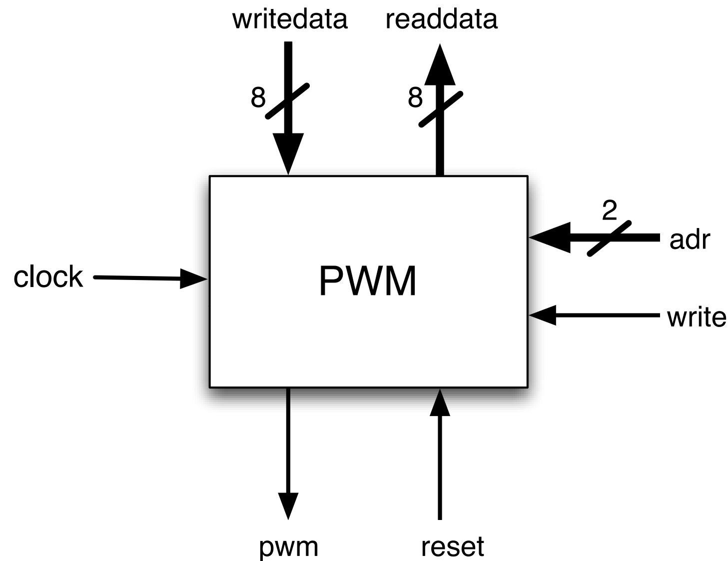

Abb. 2.2 PWM-Komponente

Über den Datenbus sollen vier verschiedene Register angesteuert werden, die über die Adressleitung auswählbar sind. Die Registertabelle soll folgendermaßen aufgebaut sein:

Adresse	**Funktion** Bitreihenfolge: 7 .. 0	
0	Duty-Cycle	
1	Teilfaktor – Bits 7 bis 0	
2	Teilfaktor – Bits 15 bis 8	
3		E

Bit 0 ist enable

Über das Enable-Bit an der Stelle 0 an Adresse 3 soll der PWM-Generator ein- und ausgeschaltet werden können, damit z. B. während der Programmierung der Parameter keine möglicherweise für

die angeschlossene Hardware fatalen Signale entstehen können. Über das write-Signal soll gesteuert werden, wann die Eingabe auf dem Datenbus in das momentan adressierte Register übernommen wird. Das Reset soll asynchron erfolgen und die ganze Komponente (alle Register) auf Null zurücksetzen (und damit natürlich auch die Ausgabe des PWM-Signals unterbinden).

Implementieren Sie die oben vorgestellte Komponente.

Binden Sie die Komponente in Ihre Top-Level-Entity ein. Als Taktsignal soll wieder direkt der 50 MHz Takt verwendet werden. Als Dateneingabe dienen die Schiebeschalter 7 bis 0, als Ausgabe die roten LEDs 7 bis 0. Benutzen Sie SW17 für den Reset, SW16 für das write-Signal, SW15 für das Adressbit 1 und SW14 für das Adressbit 0. Die Ausgabe des PWM-Signals soll über eine grüne LED erfolgen.

Kapitel 3

Wiederholung Qsys und Nios II

Aufgabe 1

Erstellen Sie ein neues System auf der Grundlage eines Nios II Prozessors. Um mehr Platz für den kompilierten Programmcode sowie die Laufzeitdaten zu haben, soll als Speicher der auf dem DE2-Board vorhandene SDRAM verwendet werden. Im Qsys-Builder gibt es bereits eine fertige IP-Komponente für den SDRAM-Controller. Die physikalische Verdrahtung der Taktleitung zum RAM auf dem Board führt jedoch zu einer Taktverzögerung (clock skew), so dass die Taktflanke in Relation zu den anderen Signalen zu spät am RAM-Baustein ankommt. Es ist in diesem Fall notwendig, die Flanke für den SDRAM um 3 ns nach vorn (früher) zu verschieben. Dies wird über eine Phase-Locked-Loop (PLL) realisiert, der ein Taktsignal mit der selben Frequenz aber zusätzlicher negativer Phasenverschiebung liefern kann. Der Takt für die restlichen Komponenten wird dabei unverändert weiterpropagiert. Abbildung 3.1 verdeutlicht diese Zusammenhänge.

Im Qsys-Builder können Sie direkt unter Verwendung der Komponente `Avalon ALTPLL` eine um 3 ns Verzögerte Clock erstellen. Dabei sollten Sie auch die unverschobene Clock für den Prozessor und dessen Komponenten als Ausgangsclock der PLL wählen. So ist sichergestellt, dass beide Clocks jeweils um 3ns verschoben sind und nicht durch die PLL eine weitere Verschiebung hinzukommt. Die Verschobene Clock muss muss dabei als Conduit exportiert werden um dann in Quartus II direkt mit dem SDRAM verbunden werden zu können.

Beim Einbinden des SDRAM sind folgende Werte für den SDRAM auf dem DE2-Board einzustellen:

- Presents: Custom
- Data Width: 16 bits
- Architecture: Chip select 1 und 4 banks
- Address width: 12 rows und 8 columns

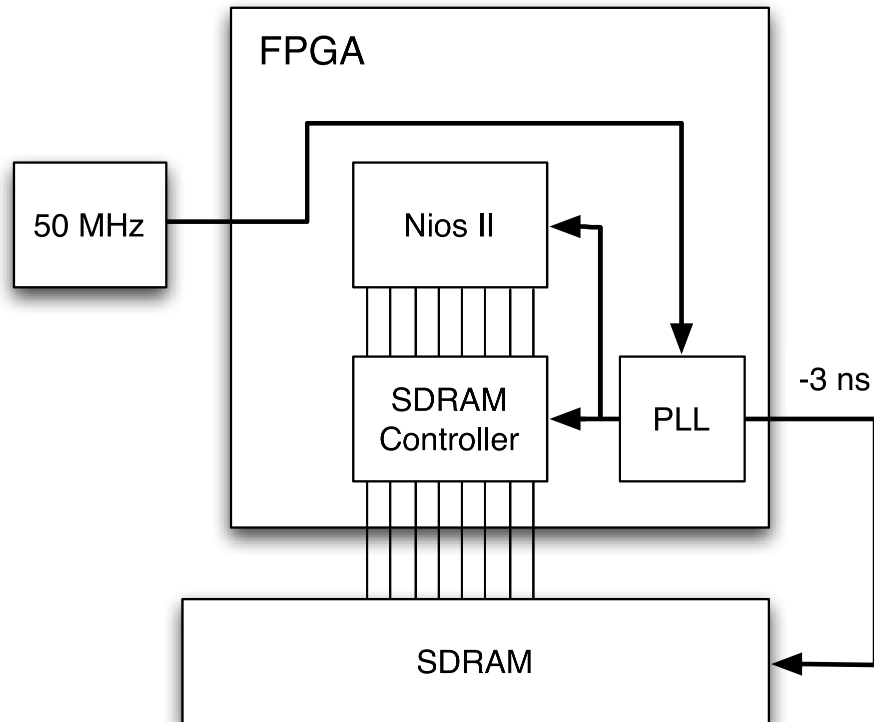

Abb. 3.1 PLL für die Phasenverschiebung des SDRAM-Taktes

Neben einem Nios II Prozessor und dem SDRAM soll Ihr System über einen simplen Taktgeber (Timer) mit 1000 Hz sowie über parallele Ein- bzw. Ausgänge für LEDs und Taster verfügen. Weiterhin ist eine Schnittstelle zur Konsole auf dem PC gewünscht, wofür in diesem Fall das JTAG-UART verwendet werden soll.

Die Wahl des Prozessortyps steht Ihnen frei, wobei vorerst die kleinste Variante ausreicht. Die Priorität des IRQs des Timers sollte höher als die des UARTs sein. Für die Ausgabe sehen Sie bitte 16 rote LEDs und für die Eingabe die vier Taster vor. Verwenden Sie für die PIO-Ports die Komponentennamen `led_pio` bzw. `button_pio`.

Aufgabe 2

Die Software soll nun mittels der auf Eclipse basierenden Entwicklungsumgebung Nios II Software Build Tool (SBT) erstellt und getestet werden. Grundlage dafür ist u. A. die Altera HAL (Hardware Abstraction Library), welche die Initialisierung des Systems, die Anbindung an die Standard-C-Library sowie die Einbindung von Gerätetreibern vornimmt. Mit der Altera HAL als Zwischenschicht wird die Ansteuerung von Hardwarekomponenten ermöglicht, ohne dass ein direkter Zugriff auf die Adressen der Komponenten notwendig ist. Das Nios II SBT generiert dazu aus den Informationen des QSys-Designs eine Systembeschreibung, die einen Zugriff auf Komponenten per Namen statt Adressen ermöglicht. Weiterhin werden auf diese Weise nur für die im konkreten System vorhandenen Komponenten die Treiber kompiliert und hinzugelinkt. Auch die Befehle zur Systeminitialisierung werden automatisch generiert.

All dies ermöglicht es, Software wie auf einem normalen Hostsystem ausgehend von einer `main()`-Methode zu entwickeln, ohne sich um Dinge wie Sicherung und Wiederherstellung des Systemzustands sowie das Routing von Ein- und Ausgabe kümmern zu müssen. Dank der Altera HAL kann beispielsweise die Funktion `printf()` zur Ausgabe von Text verwendet werden, wenn im System eine passende Schnittstelle vorhanden ist (in unserem Fall das JTAG-UART).

Ein Projekt in dem Nios II SBT besteht üblicherweise aus zwei Komponenten: einer System-Library, welche die HAL und weitere Systemkomponenten enthält und konfiguriert, die sogenannte board support packages (BSP) sowie der eigentlichen Anwendung. Letztere sollte nur den eigenen Code enthalten und ist von der System-Library abhängig. Die für Ihr System passende Library wird meist durch das System generiert. Das zugehörige makefile wird Ihnen zusammen mit dem BSP makefile erstellt. Eine allgemeine Einführung sowie den Umgang mit dem Nios II SBT finden Sie im `Nios II Software Developer's Handbook` in `Section I` in `Chapter 2` und `4`.

Erstellen Sie in dem Nios II SBT ein neues Projekt. Es sind bereits einige Templates vorhanden. Beachten Sie jedoch für weitere Versuche, dass einige von ihnen eine stark reduzierte HAL verwenden, die auf viele Funktionen verzichtet. So ist dort z. B. `printf()` nicht vorhanden. Machen Sie sich mit den grundlegenden Funktionen der Nios II SBT vertraut. Lassen Sie sich ein „Hello World!“ auf der Konsole anzeigen.

Aufgabe 3

Erstellen Sie ein Lauflicht, das einen Leuchtpunkt auf den roten LEDs anzeigt, welcher sich hin und her bewegt. Erweitern Sie das Lauflicht so, dass sich die Geschwindigkeit über die Taster auf dem Board ändern lässt. Die per `#include` einzubindende Datei `altera_avalon_pio_regs.h` (aus `altera.components`) enthält Makros für den Zugriff auf PIO-Ports. Die wichtigsten sind

```
1 IORD_ALTERA_AVALON_PIO_DATA(base)
```

für den lesenden Zugriff auf einen Port, wobei als `base` die Konstante der Basisadresse verwendet wird (s. u.). Mit

```
1 IOWR_ALTERA_AVALON_PIO_DATA(base, data)
```

kann entsprechend auf einen PIO-Port geschrieben werden.

Die benötigten Systemkonstanten lassen sich in der Datei `system.h` im `Debug`-Verzeichnis der System-Library des jeweiligen Projekts finden, die bei der Kompilation automatisch generiert wird. Um die in dieser Datei hinterlegten Basisadressen nutzen zu können, muss sie ebenfalls zuvor per `#include` in Ihr Programm eingebunden werden. Die Konstanten der von Ihnen im Qsys-Builder verwendeten Komponenten basieren auf dem Schema

```
1 <Komponentenname in Grossbuchstaben>_BASE
```

also beispielsweise `LED_PIO_BASE`.

Eine allgemeine Einführung in den PIO-Core von Altera ist im Quartus II Handbuch, Volume 5, Sektion I, Kapitel 9 zu finden (Datei `n2cpu_nii51007.pdf` im Verzeichnis dieser Übung).

Aufgabe 4

Erweitern Sie ihr System um einen PIO-Port für die 7-Segment-Anzeigen HEX3 bis HEX0. Stellen Sie unter dem Namen `seven_seg_pio` einen 32 Bit breiten Ausgabe-Port bereit. Sorgen Sie für eine **sinnvolle** Verbindung der 32 Ausgabe-Bits mit den benötigten 4*7 Pins für die Anzeigen (Stichwort: „Byte“).

Entwickeln Sie eine Ansteuerung für die Segmentanzeigen und realisieren Sie darauf als Softwarelösunge einen hexadezimalen sowie einen dezimalen Zähler.

Freiwillige Zusatzaufgabe

Verschönern Sie das Lauflicht, indem Sie eine Spur bzw. einen Schatten aus dunkler werdenden LEDs hinter dem aktiven Lichtpunkt herziehen und somit für weichere Übergänge sorgen („Knight Rider“).

Kapitel 4

Memory-Mapped-IO

In dieser Übung soll das bereits von Ihnen entwickelte programmierbare PWM-Modul an den Nios II Prozessor angebunden und dann per Software gesteuert werden. Altera stellt dafür ein Konzept bereit, um per Memory-Mapped-IO auf die Register von Komponenten eines Qsys zugreifen zu können.

Das Avalon-MM System-Interconnect-Fabric ist ein Bus-ähnliches Konstrukt, das die Verbindung von Komponenten innerhalb eines mit dem Qsys-Builder erstellten Systems ermöglicht. Für die angeschlossenen Komponenten erscheint die Anbindung und Adressierung wie ein gewöhnlicher gemeinsamer System-Bus, intern arbeitet das Avalon-MM allerdings mit Eins-zu-Eins-Verbindungen, genauer gesagt mit Slave-seitiger Arbitrierung. Bei dieser Verbindungsart wird auf der Seite des Slaves entschieden, mit welchem Master er kommunizieren soll, so dass zur selben Zeit verschiedene Master mit verschiedenen Slaves Daten austauschen können.

Im Folgenden wird das Avalon-MM System-Interconnect-Fabric der Einfachheit halber als „Avalon-Bus“ oder schlicht „Avalon“ bezeichnet werden, auch wenn dies (wie oben dargestellt) technisch gesehen nicht ganz korrekt ist.

Um mit einer Komponente am Avalon-Bus teilnehmen zu können, ist es notwendig, das entsprechende Avalon-Interface zu implementierten. Dabei gibt es einige elementare Signale, die jedes Interface beherrschen muss, sowie darüber hinaus gehende Erweiterungen, die optional sind.

Die Idee des Interconnect-Fabrics wird in Kapitel 2 von Volume 4, Sektion I des Quartus II Handbuchs beschrieben (Datei `qts_qii54003.pdf` im Übungsverzeichnis). Die Spezifikation des gesamten Avalon-Systems findet sich in der Datei `mnl_avalon_spec.pdf`.

Aufgabe 1

Erstellen Sie ein neues System mit einem Nios II Prozessor, der Zugriff auf eine Instanz Ihres PWM-Moduls hat.

Dank der von Ihnen bereits implementierten Register kann eine Anbindung an den Avalon-Bus relativ leicht erfolgen. Die vorhandene Trennung von `readdata` und `writedata` passt direkt zum Avalon-Interface.

Benutzen Sie Ihre selbst entwickelte PWM-Komponente als Basis. Achten Sie darauf, dass nur die angegebenen Signale (und evtl. zusätzlich `read`) oder ihre Komplemente in der Schnittstelle nach außen geführt werden. Falls nötig, passen Sie Ihre Komponente noch einmal an.

Wenn Sie zuvor die Komponente wie angegeben erstellt haben, sollte sie bereits den Anforderungen eines Slave-Interfaces des Avalon-Busses genügen. Überprüfen Sie anhand des Timing-Diagramms für Lese- und Schreibzugriffe mit festen Wartezyklen (Seite 21 bzw. 3-9 in der Interface-Spezifikation), ob das auch für Ihre Implementierung gilt. Vorgesehen sind jeweils **null Wartezyklen**.

Anschließend können Sie beginnen, das PWM-Modul in den Qsys-Builder einzubinden. Dazu legen Sie die VHDL-Datei in Ihrem Projektverzeichnis ab (Alteras Entwurfsfluss sieht dafür das Unterverzeichnis `hw` vor, um den selbst erstellten Code von den automatisch erzeugten VHDL-Dateien unterscheiden zu können) und wählen im Qsys-Builder den Menüpunkt „File“ und dann „New Component“. Dort können Sie nun Ihren VHDL-Code auswählen und die Signale des Avalon-Busses mit Ihren eigenen verbinden.

Sie benötigen neben dem Slave-Interface einen Clock-Input; das Hinausführen des PWM-Signals geschieht über ein „export“ in einem Conduit-Output, welches das Signal außerhalb des Qsys-Cores verfügbar macht.

Beim Timing sollten Sie alle Werte vorerst auf **null** stellen. Geben Sie im „Component Wizard“ (rechte Registerkarte) als „Component Class Name“ `PWM` ein.

Sie können nun eine Instanz ihres PWM-Moduls dem System hinzufügen. Der Name dieser Instanz muss sich allerdings aus technischen Gründen vom Namen des Moduls unterscheiden.

Nun können Sie das System generieren. Binden Sie das exportierte PWM-Signal an die grüne LED `LEDG0` und synthetisieren Sie das Gesamtsystem.

Aufgabe 2

Erstellen Sie in dem Nios II SBT ein neues Projekt, dass das PWM-Modul ansteuert.

Ein „Mini-Treiber“ für den Registerzugriff ist bereits mit der Datei `pwm_regs.h` gegeben, die Sie im Übungsverzeichnis finden. In dieser Header-Datei werden einige Makros definiert, die etwas von dem direkten Zugriff auf die IO-Ports der Komponente abstrahieren. So existieren für jedes Register eigene Lese- und Schreibbefehle und es muss nur noch die Basisadresse übergeben werden, der Offset zum gewünschten Register jedoch nicht. Die Form dieser Makros ist bereits von Altera für die Treiberentwicklung vorgesehen.

Erstellen Sie einen „richtigen“ kleinen Treiber in Form einer Auswahl an Funktionen, denen Sie die Werte für den Dutycycle sowie den Frequenzteiler übergeben können. Schreiben Sie auch eine Routine, welche eine direkte Übergabe der Frequenz in Hertz ermöglicht. Wenn das PWM-Modul am selben Takt wie die CPU angebunden ist (so wie es eigentlich sein sollte) dann können Sie die Konstante `ALT_CPU_FREQ` in Ihrer Rechnung für die Taktfrequenz benutzen. Achten Sie bei diesen Funktionen auch auf die korrekte bzw. sinnvolle Benutzung des „Enable“.

Schreiben Sie dann ein Hauptprogramm, dass unter Verwendung Ihrer Treiberroutinen den Dutycycle zyklisch von 0 % bis 100 % (also von `0x00` bis `0xff`) und wieder zurück verändert, so dass sich ein sichtbares Pulsieren der grünen LED ergibt. Für die Wartezeiten können sie die Funktion `usleep(unsigned int useconds)` verwenden, wenn Sie die Header-Datei `unistd.h` einbinden. Denken Sie auch daran, die Header-Dateien `system.h` und bei Bedarf `alt_types.h` (wenn Sie die Altera-Datentypen verwenden wollen) mit einzubinden.

Aufgabe 3

Verändern Sie ihr Hauptprogramm so, dass Sie mit je zwei Tastern den Dutycycle sowie die Frequenz des PWM-Signals frei einstellen können (also kein automatisches Pulsieren mehr).

Auf dem Board ist auch ein 16x2 Zeichen LCD vorhanden. Dieses soll nun angesteuert werden. Dazu müssen Sie einerseits im Qsys-Builder die passende IP-Komponente hinzufügen und andererseits für deren Anbindung in der VHDL-Beschreibung des Systems sorgen. Stellen Sie sicher, dass in Ihrem Design alle Komponenten an den zentralen Systemtakt angebunden sind.

Passen Sie das Port-Mapping der instanziierten Qsys-Komponente an und stellen Sie die Verbindung zu den richtigen Pins des LCDs her (Schauen Sie in die Pin-Assignments). Das Display wird über

den Pin `LCD_ON` ein/ausgeschaltet. Wenn Sie mögen, können Sie einen PIO-Port dafür hinzufügen, um das Display per Software schalten zu können. Ansonsten sorgen Sie dafür, dass der Pin dauerhaft mit `'1'` getrieben wird. Der Pin `LCD_BLON` (Backlight On) ist ohne Bedeutung, da das LCD auf dem Board keine Hintergrundbeleuchtung besitzt.

Die Ansteuerung des Displays wird in Volume 5, Sektion I, Kapitel 8 des Quartus II Handbuchs beschrieben (Datei `n2cpu_nii51019.pdf`). Den Zugriff auf den Treiber sollen Sie nun selbst übernehmen. Nach dem Einbinden der Header-Dateien `stdio.h` und `altera_avalon_lcd_16207.h` muss zunächst ein Filedescriptor erstellt werden, über den das Ausgabegerät später referenziert werden kann:

```
1 FILE *lcd;
```

Binden des LCDs an diesen Filedescriptor:

```
1 lcd = fopen("/dev/lcd_0", "w");
```

wobei für `lcd_0` der Name der entsprechenden Instanz aus dem Qsys-Builder verwendet werden muss. Den Namen aller Device (und damit auch der Instanz des LCDs) sind in der `system.h` aufgeführt. Ist der Rückgabewert ungleich `NULL`, so war die Initialisierung des Displays erfolgreich und es kann mittels

```
1 fprintf(lcd, ...);
```

analog zu einem normalen `printf()` auf das LCD geschrieben werden. Die Steuerkommandos für das Anspringen bestimmter Positionen etc. können aus dem entsprechenden PDF-Dokument entnommen werden.

Hinweise: Für eine statische Anzeige muss die Ausgabe zweimal `\n` enthalten. Werden mehr als 16 Zeichen pro Zeile ausgegeben, läuft die Information durch.

Geben Sie den Dutycycle sowie die Frequenz des PWM-Signals auf dem LCD aus. Geben Sie zusätzlich den 16 Bit Wert, der wirklich als Taktteiler in die Register des PWM-Moduls geschrieben wird, auf den 7-Segment-Anzeigen aus.

Aufgabe 4

Erweitern Sie Ihre PWM-Komponente so, dass der Dutycycle mit einer Auflösung von 16 Bit eingegeben werden kann. Der Taktteiler soll auf mindestens 24 Bit erweitert werden.

Mit Ihrer bisherigen Implementierung, in der Sie die Skalierung des Dutycycle auf den jeweils aktuellen Zählerbereich in Hardware durchführen, würde dabei der Multiplizierer enorm größer werden. Deutlich sinnvoller ist es deshalb, die Berechnung der Skalierung von der Hardware in die Software zu verlagern, zumal die Rechnung nicht kontinuierlich sondern nur bei Änderungen der Parameter durchgeführt werden muss.

Entfernen Sie daher die Berechnung der Skalierung aus Ihrer Komponente und passen Sie den Rest an die neuen Anforderungen an. Es steht Ihnen dabei frei, ob Sie die Register verbreitern (sinnvollerweise nicht breiter als max. 32 Bit, der Datenbreite des Nios II Prozessors) oder statt dessen den Adressraum vergrößern (also mehr Register vorsehen). Achten Sie in jedem Fall auf eine durchdachte Aufteilung der Parameter auf Register.

Es bietet sich an, dafür eine Kopie Ihrer vorhandenen Komponente (sowohl VHDL-Code als auch im Qsys-Builder) zu erstellen. Denken Sie daran, auch die Header-Datei für den Register-Zugriff anzupassen. Hier müssen Sie falls notwendig mindestens die Lese- und Schreibbefehle sowie die Offsets aktualisieren.

Generieren Sie anschließend das neue System, und passen Sie Ihren Treiber an, in welchem Sie nun auch die Skalierung vornehmen müssen. Beachten Sie dabei die nun geänderten Wertebereiche.

Testen Sie Ihr neues System.

Kapitel 5

Debugging

In diesem Teil des Projektes sollen die grundlegenden Techniken des Debuggings von auf einem Qsys basierenden Nios II System betrachtet werden. Um auf der Hardware-Ebene den Signalablauf innnerhalb selbsterstellter Komponenten analysieren zu können, kommt ein in die Quartus II Umgebung integrierter Logic-Analyser zum Einsatz. Die Softwarefunktionalität sowie die Interaktion mit der Hardware über Memory-Mapped-IO lässt sich mittels eines in dem Nios II SBT vorhandenen auf dem GDB basierenden Debugger überprüfen.

5.1 Hardware-Debugging

Der in die Quartus II Umgebung integrierte SignalTap II ist ein Logic-Analyser, welcher auf einem FPGA als Logik instantiiert werden kann. Als Speicher für die anfallenden Daten, auch Samples genannt, dient dabei der freie Teil des On-Board-RAM. Die Konfiguration sowie das Auslesen der Daten erfolgt über die JTAG-Schnittstelle, so dass die Signalverläufe schließlich im Quartus II visualisiert werden können.

Der direkte Zugriff auf in der Hardware vorhandene Bitmuster bietet insbesondere dann einen großen Vorteil, wenn externe Hardware angebunden ist, die nicht mit einer HDL spezifiziert wurde und daher von einer Simulation nicht erfasst werden kann. Aber auch bei simulierbaren Systemen kann die Verwendung des Logic-Analyzers Vorteile mit sich bringen, etwa dann, wenn nur ganz bestimmte Aspekte betrachtet werden sollen, und eine zeitgenaue Simulation gerade in Verbindung mit einem Softcore-Prozessor einen hohen zeitlichen Aufwand bedeuten würde.

Dieser mitgelieferte interne Logic-Analyser besitzt einige komplexe Funktionen, die sonst nur bei aufwendigen externen Analysern zu finden sind, beispielsweise mehrstufig verkettete Trigger, die

erst dann die Aufzeichnung der Analysedaten starten, wenn mehrere Bedingungen auf gleichen oder unterschiedlichen Signalen zutreffen. Darüber hinaus lassen sich mit Hilfe von State-Machines einfache Protokolle nachbilden und zur Triggerung verwenden. Weiterhin werden auch externe Trigger unterstützt. Der Sample-Speicher lässt sich segmentieren, so dass Daten an mehreren Trigger-Zeitpunkten, die weiter auseinander liegen, aufgezeichnet werden können. Der größte Unterschied im Vergleich zu eigenständigen Logic-Analysern liegt in der starken Größenbeschränkung des Sample-Speichers, die in der Verwendung der im FPGA integrierten Speicherblöcke begründet ist.

Eine genaue Beschreibung der Funktionsweise sowie Hinweise zur Konfiguration finden Sie in Volume 3, Sektion IV, Kapitel 14 des Quartus II Handbuchs unter dem Titel „Design Debugging Using the SignalTap II Embedded Logic Analyzer“. Der Dateiname für dieses Kapitel lautet z. Z. `qts_qii53009.pdf`.

Aufgabe 1

1. Fügen Sie zu Ihrem System eine Instanz des Signal-Tap II Logic-Analysers hinzu. Machen Sie sich mit der Konfiguration und Bedienung des Werkzeugs vertraut.
2. Konfigurieren Sie den Trigger auf einen Lesezugriff auf eine per Avalon angebundene Qsys-Komponente und vergleichen Sie die erfassten Daten mit den Werten, welche die Software aus diesem Zugriff erhält. Dazu sollten Sie ein kleines Testprogramm erstellen, welches die Werte einliest und auf der Konsole ausgibt.

5.2 Software-Debugging

Das Debugging der Software basiert auf dem zum GCC gehörenden GNU-Debugger GDB und ist in das NIOS II SBT integriert, wobei die Grundfunktionalität der Bedienoberfläche bereits vom Eclipse-Framework bereitgestellt wird. Die Debugging-Ansicht des SBT dient somit als GUI für den GDB, und beherrscht die übliche vom Debugger bereitgestellte Funktionalität.

Der Ablauf von Programmen kann unterbrochen und fortgesetzt werden, und es kann ein Zugriff auf die Systemparameter erfolgen. So lassen sich z. B. der Inhalt von Registern, Variablen, sowie des gesamten Speicherbereichs auslesen und verändern. Mittels Stepping lassen sich Programme Schritt für Schritt sowohl im C-Code als auch im zugehörigen Assembler-Code ausführen. Es können also die Auswirkungen einzelner Befehle genau beobachtet werden. Über so genannte Breakpoints lässt

sich ein laufendes Programm automatisch stoppen, wenn während der Ausführung bestimmte, zur Laufzeit definierbare Stellen im Code erreicht werden.

Aufgabe 1

Starten Sie für eines Ihrer vorhandenen Programme den Debugging-Modus, in dem Sie statt dem üblichen „Run as...“ das „Debug as...“ verwenden.

1. Führen Sie zunächst eine schrittweise Abarbeitung ihres C-Codes durch und beachten Sie die Auswirkungen auf die Register und Variablen. Experimentieren Sie mit den unterschiedlichen Schritt-Möglichkeiten („Step into“, „Step over“ „Step out“).
2. Schalten Sie den Schrittmodus auf Assembler-Instruktionen um. Vergleichen Sie den Assembler-Code mit Ihrem C-Code.
3. Fügen Sie in der Mitte Ihres Codes, z. B. innerhalb einer großen Schleife, einen Breakpoint ein. Lassen Sie das Programm bis zu diesem Punkt durchlaufen.
4. Schauen Sie sich bekannte Adressbereiche (z. B. den Ihrer eigenen Avalon-Komponente; die Adresse können Sie der Datei `system.h` oder der Übersicht im Qsys-Builder entnehmen) im Speicher an. Beobachten Sie, wie Ihre Treiberfunktionen die in den Speicher eingeblendeten Register verändern.
5. Schreiben Sie direkt mittels des Debuggers in die Register Ihrer Komponente. Beobachten Sie die Auswirkungen auf die Hardware.

Beachten Sie dabei, dass der direkte schreibende Zugriff aus dem Debugger momentan nur für 8 Bit breite Register (also die Ihrer ersten Implementierung) funktioniert. Bei breiteren Registern verwendet der Debugger einen Byte-weisen Zugriff, der nur dann zu korrekten Ergebnissen führt, wenn Ihre Komponente über das `byteenable`-Signal des Avalon-MM verfügt und dieses korrekt interpretiert. Der Zugriff aus Ihrem C-Code sollte hingegen mit 32 Bit breiten Zugriffen arbeiten, so dass er nicht von dieser Einschränkung betroffen ist. Falls Ihre Komponente Register verwendet, die breiter als 8 Bit sind, können Sie durch nachträgliche Implementierung der `byteenable`-Funktionalität (siehe Avlalon-Spezifikation) eine korrekte Behandlung der Schreibzugriffe des Debuggers erreichen.

Kapitel 6

Treiberentwicklung

Dieses Kapitel soll an die Vorgehensweise der Entwicklung vollständiger, modularer Treiber heranführen. Um eine sinnvolle Verwendbarkeit innerhalb des Labors zu ermöglichen, werden sich die einzelnen Schritte an den Treibern für den Altera-HAL orientieren. Die vermittelten Konstrukte und Konzepte sind aber auch gut für die Verwendung in anderen Umgebungen geeignet.

6.1 Interrupts

Bevor nun die ersten Treiber entwickelt werden, soll zunächst die Verwendung von Interrupts im Altera-HAL auf Nios II Prozessoren erlernt werden. Es wird an dieser Stelle davon ausgegangen, dass das Konzept von Interrupts bekannt ist, so dass nun vor allem die praktische Verwendung vorgestellt wird.

Bei Verwendung des Altera-HAL konnen die Interrupt-Service-Routinen (ISRs) direkt in C programmiert werden, wobei darin natürlich allgemein keine blockierenden Funktionen aufgerufen werden dürfen. Das Sichern und Wiederherstellen des aktuellen Prozessor-Kontexts wird automatisch vom HAL erledigt, die ISR ist nicht viel mehr als eine normale Funktion.

Für die Verwendung von Interrupts stellt der Altera-HAL folgende Funktionen zur Verfügung, die über die Header-Datei `sys/alt_irq.h` eingebunden werden:

```
1 alt_ic_isr_register()
2 alt_ic_irq_disable()
3 alt_ic_irq_enable()
4 alt_irq_disable_all()
5 alt_irq_enable_all()
6 alt_ic_irq_enabled()
```

Eine detaillierte Beschreibung der Funktionen finden Sie im NIOS II Developer's Handbuch in Section IV, Chapter 14. Um eine Funktion beim HAL als ISR bekannt zu machen, wird die Funktion

```
int alt_ic_isr_register (alt_u32 id,
        alt_u32 irq,
        alt_isr_func isr,
        void* context,
        void flags)
```

verwendet, wobei `id` die zugehörige Interrupt-ID, `irq` die zugehörige Interrupt-IRQ, `isr` ein Funktionspointer auf die ISR und `context` einen Void-Pointer auf einen frei wählbaren Interrupt-Kontext sein soll. `flags` wird nicht benutzt und kann auf `0x0` gesetzt werden. Daraus lässt sich unmittelbar der Prototyp einer ISR ablesen, der folgendermaßen aussehen muss:

```
void <isr_name>(void* context)
```

Der Interrupt-Kontext dient beispielsweise dazu, der ISR Zugriffsmöglichkeit auf einen oder mehrere Parameter zu erlauben, was sonst auf Grund des fest vorgegebenen Prototypen nicht flexibel möglich wäre. Dazu sollte `context` auf eine Datenstruktur zeigen, die dann der ISR bei deren Ausführung übergeben wird. Wichtig wird dies besonders für die nachfolgend vorgestellten HAL-kompatiblen Treiber, die dadurch auch mehrere gleiche Geräte verwalten können.

Generell sollte der Code von ISRs so kurz wie möglich gehalten werden, da während ihrer Ausführung üblicherweise alle Interrupts deaktiviert sind und andere zeitkritische Funktionen nicht ausgeführt werden können. Typische Aufgaben sind die Reaktion auf Ereignisse von Hardwaregeräten wie z. B. das Leeren von Puffern und das Setzen von Signalen, die Funktionen im normalen Programmablauf aktivieren, welche dann die eigentliche Arbeit erledigen.

Die Kommunikation mit dem restlichen Programm findet üblicherweise über die Datenstruktur der jeweiligen Treiberinstanz statt, für uns soll an dieser Stelle aber vorerst eine globale Variable genügen. Es folgt ein Beispiel einer ISR für den Zugriff auf die Register einer erweiterten PIO-Komponente.

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

static void handle_button_interrupts(void* context, alt_u32 id)
{
    /* Cast context to edge_capture's type. It is important that this
     *  is declared volatile to avoid unwanted compiler optimization.
     */
    volatile int* edge_capture_ptr = (volatile int*) context;

```

```
12      /* Read the edge capture register on the button PIO.
13       * Store value.
14       */
15      *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
16
17      /* Write to the edge capture register to reset it. */
18      IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);
19
20      /* Read the PIO to delay ISR exit. This is done to prevent a
21       * spurious interrupt in systems with high processor -> pio
22       * latency and fast interrupts.
23       */
24      IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
25  }
```

Eventuell sollten Sie nochmals auf die Dokumentation des PIO-Core zurückgreifen, um die Vorgänge in der ISR nachvollziehen zu können. Der letzte Lesebefehl ist (wie im Kommentar angedeutet) nicht zwangsweise notwendig, soll aber den Fall abdecken, dass der Prozessor nach dem Schreibbefehl zum Zurücksetzen des IRQs fortfährt, noch bevor sich der neue IRQ-Zustand bis zum Prozessor zurückpropagiert hat, und somit sofort wieder ein Interrupt ausgelöst wird, ohne dass neue Daten zum Auslesen vorliegen.

Die Registrierung dieser ISR könnte beispielsweise folgendermaßen aussehen.

```
1   #include "sys/alt_irq.h"
2   #include "system.h"
3   ...
4   /* Declare a global variable to hold the edge capture value. */
5   volatile int edge_capture;
6   ...
7   /* Initialize the button_pio. */
8   static void init_button_pio()
9   {
10      /* Recast the edge_capture pointer to match the
11      alt_irq_register() functionprototype. */
12      void* edge_capture_ptr = (void*) &edge_capture;
13
14      /* Enable all 4 button interrupts. */
15      IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
16
17      /* Reset the edge capture register. */
18      IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
19
20      /* Register the ISR. */
```

```
21     alt_ic_isr_register (BUTTON_PIO_IRQ_CONTROLLER_ID,
22         BUTTON_PIO_IRQ,
23         handle_button_interrupts,
24         edge_capture_ptr,
25         0x0);
26 }
```

Vor der Registrierung der ISR werden die Interrupts in der Komponente durch aufheben der Maskierung aktiviert sowie die Register für die Flankenerkennung (und damit ein eventuell bereits bestehender IRQ) zurückgesetzt. Die Variable `edge_capture` ist in diesem Fall die Kommunikationsschnittstelle zwischen der ISR und dem Hauptprogramm. Bei nichtatomaren Zugriffen auf diese Variable sollte dementsprechend der Interrupt für die Dauer der Operation auf Seiten des Prozessors deaktiviert werden, weil es sonst vorkommen kann, dass die ISR zwischenzeitlich den Wert verändert. Folgendes Beispiel zeigt die Klammerung eines solchen kritischen Abschnitts.

```
1 /* Disable interrupt */
2 alt_ic_irq_disable(BUTTON_PIO_IRQ_CONTROLLER_ID);
3
4 ... /* critical section */
5
6 /* Re-enable interrupt */
7 alt_ic_irq_enable(BUTTON_PIO_IRQ_CONTROLLER_ID);
```

Da in den meisten Fällen aber vermutlich nur einfache Lese- und Schreibzugriffe auf eine solche gemeinsame Variable erfolgen werden, ist die vorgestellte Klammerung nicht so häufig notwendig.

Aufgabe 1

Passen Sie die Verwendung von Tastern in ihrem Programm so an, dass die Interrupts und Flanken-Register verwendet werden. Dazu müssen sie die für die Taster zuständige PIO-Komponente im Qsys-Builder auf flankengesteuerten Interruptbetrieb umstellen und den IRQ entsprechend konfigurieren. Erstellen Sie eine ISR, welche den neuen IRQ behandelt und verändern Sie ihr Hauptprogramm so, dass nicht mehr direkt auf die PIO-Komponente zugegriffen wird. Die Verwendung der Flanken-Register sollte ihr Programm zusätzlich vereinfachen, da nun nicht mehr genau der Augenblick getroffen werden muss, in dem der bzw. die Taster gedrückt sind.

Zum Debugging Ihrer ISR können Sie einen Breakpoint innerhalb der Funktion verwenden.

6.2 Treiber für den Altera-HAL

Im Folgenden soll nun die eigentliche Treiberentwicklung für den Altera-HAL vorgestellt werden. Der Großteil des Aufwands ist hier die korrekte Implementierung zahlreicher Schnittstellen, während die Logik der Registerzugriffe nur einen kleinen Teil einnimmt. Belohnt wird dieser Aufwand allerdings mit der universellen und einfachen Verwendbarkeit des Treibers, was sowohl für mehrere Instanzen der gleichen Hardware wie auch für den gleichzeitigen Zugriff aus verschieden Tasks eines Echtzeitbetriebssystems gilt.

Folgende Schritte bzw. Stufen werden üblicherweise bei der Entwicklung eines Treibers durchlaufen.

Zunächst werden die direkten Registerzugriffe auf die Hardware realisiert, dabei spielen neben den Lese- und Schreibzugriffen auch Adress- bzw. Offsetberechnungen sowie gegebenenfalls Bitmasken und deren Offsets eine Rolle. Anschließend wird die Funktionalität der anzusteuernden Hardwarekomponente sichergestellt. Dies entspricht in etwa der Verwendung der zuvor definierten Register-Makros (z. B. `pwm_regs.h`) direkt im Hauptprogramm. In einem nächsten Schritt werden dann einzelne Basisroutinen zusammengefasst und in Form von Funktionen zur Verfügung gestellt, die wiederum von höheren Treiberfunktionen verwendet werden können.

An dieser Stelle sollten Sie sich mit ihren eigenen Treibern bereits befinden. In einem nächsten Schritt wird dann die Integration ins System (in unserem Fall den Altera-HAL) vorbereitet, indem eine Vereinheitlichung und Strukturierung der Treiberfunktionen vorgenommen wird. So sollte z. B. genau zwischen inneren und äusseren Funktionen differenziert werden, wobei letztere die vom Anwender für den Gerätezugriff verwendeten Methoden darstellen, während er auf die inneren Funktionen keinen direkten Zugriff benötigen sollte. Weiterhin sollten von mehreren Funktionen verwendete gemeinsame Variablen (in welchen beispielsweise der momentane Status der Hardware hinterlegt ist) in einer übersichtlichen Datenstruktur zusammengefasst werden. Dort wird unter anderem die Basisadresse der Hardwareinstanz hinterlegt. Die für die Initialisierung notwendigen Schritte (Belegung von Variablen, Registrierung der ISR, etc.) werden in eine eigene Init-Funktion ausgelagert. Das Zusammenspiel der Funktionen sowie die Gesamtfunktionalität können dann aus dem Hauptprogramm getestet werden.

Der letzte Schritt ist dann die Implementierung der Treiberinterfaces des jeweiligen Systems sowie die Bündelung aller benötigten Dateien in ein Treiber-„Paket“.

Der Build-Vorgang des Nios II SBT ermöglicht im Zusammenspiel mit dem Altera-HAL und den vom Qsys-Builder erzeugten Systeminformationen eine Automatisierung der Treibereinbindung. Wenn sich die Treiberdateien an der richtigen Stelle befinden und die benötigen Schnittstellen korrekt

implementiert sind, wird der Build-Vorgang nur die für das jeweilige System benötigten Treiber einbinden und auch die Instantiierung und Initialisierung der Treiberinstanzen durchführen.

Zunächst einmal muss ein Treiber folgende Verzeichnis- und Dateistruktur aufweisen:

Das Hauptverzeichnis muss den Namen der Komponente tragen (den Sie bei der Erstellung der Komponente im Qsys-Builder angegeben haben, bitte nicht mit dem Namen der *Instanzen* der Komponente verwechseln). In diesem Verzeichnis werden dann die benötigten HDL-Dateien inklusive der Komponentenbeschreibung (`.tcl`-Datei) abgelegt. Die Komponentenbeschreibung der Hardware (`_hw.tcl`) wird Ihnen von Qsys automatisch generiert. Eine Komponentenbeschreibung ihres Treibers (`_sw.tcl`) muss von Ihnen erstellt werden.

Die Treiberdateien werden dann in zwei verschiedene Unterverzeichnisse aufgeteilt, `inc` und `HAL`. Das `inc`-Verzeichnis soll die vom HAL unabhängigen Headerdateien enthalten, welche das Hardwareinterface der Komponente definieren; auf jeden Fall die Datei mit den Register-Makros (<komponentenname>_regs.h). Das `HAL`-Verzeichnis beinhaltet dann die HAL-spezifische Anbindung des Treibers, aufgeteilt in die Unterverzeichnisse `inc` und `src` für die Header- bzw. die C-Dateien. Weiterhin liegt im `HAL/src`-Verzeichnis das Makefile namens `component.mk`. Für die Haupt-Header- bzw. C-Dateien sind die Namen <komponentenname>.h bzw. <komponentenname>.c vorgesehen. Abbildung 6.1 zeigt diese Struktur.

Wichtig: Damit der Build-Vorgang die Treiber finden kann, muss für projektspezifische Treiber die vorgestellte Struktur in einem Verzeichnis namens `ip` liegen, welches ein Unterverzeichnis des jeweiligen Quartus-Projektverzeichnisses sein muss.

Die Komponentenbeschreibung (`pwm32_sw.tcl`) eines Treiber für eine PWM Komponente names „pwm32" sieht folgendermaßen aus und muss ggf. für Ihre Zwecke angepasst werden. Eine detaillierte Beschreibung zum Aufbau finden Sie im Developer Handbuch des NIOS II in Chapter 7 der Section II

```
create_driver pwm32

set_sw_property hw_class_name pwm32

set_sw_property version 11.0

set_sw_property min_compatible_hw_version 1.0

set_sw_property auto_initialize true

set_sw_property bsp_subdirectory drivers

add_sw_property c_source HAL/src/pwm32.c
```

```
add_sw_property include_source HAL/inc/pwm32.h
add_sw_property include_source inc/pwm32_regs.h

add_sw_property supported_bsp_type HAL
add_sw_property supported_bsp_type UCOSII
```

Um die automatische Instantiierung und Initialisierung der Gerätetreiber zu ermöglichen, muss die Haupt-Headerdatei (`<komponentenname>.h`) zwei Makros mit den Namen `<KOMPONENTENNAME>_INSTANCE` und `<KOMPONENTENNAME>_INIT` definieren, die automatisch in die vom System generierte Datei `alt_sys_init.c` eingebunden und dann während der Initialisierung des Systems ausgeführt werden. Das `_INSTANCE` Makro ist für die Allokation der für die jeweilige Instanz des Treibers verwendete Datenstruktur vorgesehen, über das `_INIT` Makro kann die Initialierungs-Routine des Treibers aufgerufen werden.

Der Altera-HAL sieht verschiedene Klassen von Treibern mit verschiedenen Interfaces vor. Dies sind unter anderem zeichenbasierte Geräte, Dateisysteme und Netzwerktreiber. Nachfolgend werden schrittweise am Beispiel eines Treibers für eine PWM-Komponente mit 32 Bit breiten Registern (namens „pwm32“) die benötigen Dateien vorgestellt. In diesem Fall wählen wir keine der Standardklassen, sondern implementieren nur eine minimale Untermenge der Schnittstelle zum HAL.

Die Datei `pwm32/inc/pwm32_regs.h` mit den Makros für den Registerzugriff (sollte bei Ihnen bereits so oder ähnlich vorhanden sein):

```
#ifndef PWM32_REGS_H_
#define PWM32_REGS_H_

#include <io.h>

/* Register of compare value */
#define IOADDR_PWM32_CMP(base)         __IO_CALC_ADDRESS_DYNAMIC(base, 0)
#define IORD_PWM32_CMP(base)           IORD_32DIRECT(base, 0)
#define IOWR_PWM32_CMP(base, VALUE)    IOWR_32DIRECT(base, 0, VALUE)

/* Register of divider value */
#define IOADDR_PWM32_DIV(base)         __IO_CALC_ADDRESS_DYNAMIC(base, 4)
#define IORD_PWM32_DIV(base)           IORD_32DIRECT(base, 4)
#define IOWR_PWM32_DIV(base, VALUE)    IOWR_32DIRECT(base, 4, VALUE)

/* Enable register */
#define IOADDR_PWM32_ENABLE(base)         __IO_CALC_ADDRESS_DYNAMIC(base, 8)
#define IORD_PWM32_ENABLE(base)           IORD_32DIRECT(base, 8)
#define IOWR_PWM32_ENABLE(base, VALUE)    IOWR_32DIRECT(base, 8, VALUE)

```

```
21 #endif /*PWM32_REGS_H_*/
```

In der Datei folgenden Datei `pwm32/HAL/inc/pwm32.h` werden die Datenstruktur, die für den Anwender exportierten Funktionen sowie die Initialisierungs-Makros definiert:

```
1  #ifndef PWM32_H_
2  #define PWM32_H_
3
4  #include "sys/alt_dev.h"
5  #include "os/alt_sem.h"
6  #include "alt_types.h"
7  #include "sys/alt_errno.h"
8  #include "priv/alt_file.h"
9  #include "system.h"
10
11 typedef struct pwm32_dev_s
12 {
13     alt_llist        llist;
14     const char*      name;
15     void*            base;
16     int              enabled;
17     unsigned int     freq;
18     unsigned int     duty;
19     unsigned int     cmp;
20     unsigned int     div;
21     ALT_SEM          (lock) /* Semaphore used to control access to the
22                            * pwm hardware in multi-threaded mode */
23 } pwm32_dev;
```

Die Includes binden von der Schnittstelle benötigte Systemfunktionen sowie die Semaphoren ein. Die Datenstruktur `pwm32_dev` bietet der jeweiligen Treiberinstanz unter anderem gemeinsam benutzte Variablen an, die den Zustand des Treibers beinhalten. Die ersten beiden Einträge, `llist` und `name`, stellen die oben angesprochene benötigte minimale Untermenge der Treiberschnittstelle dar. Ihr Hauptzweck ist die in der Init-Funktion durch geführte Registrierung der Geräteinstanz per Namen. `ALT_SEM` ist ein von Altera zur Verfügung gestellter Wrapper, der eine vom Betriebssystem unabhängige Schnittstelle zu einem Semaphor bereitstellt. Dieses dient später zur Synchronisierung, falls mehrere Tasks auf die selbe Komponente zugreifen.

```
1 /*
2  * The function alt_find_dev() is used to search the device list "list" to
3  * locate a device named "name". If a match is found, then a pointer to the
4  * device is returned, otherwise NULL is returned.
5  */
6
```

```
7  extern alt_dev* alt_find_dev (const char* name, alt_llist* list);
8
9
10 /*
11  * Called by alt_sys_init.c to initialize the driver.
12  */
13 extern int pwm32_init(pwm32_dev* dev);
14
15 /*
16  * Public driver interface
17  */
18
19 extern pwm32_dev* pwm32_open(const char* name);
20 extern void pwm32_close(pwm32_dev* dev);
21
22 extern void pwm32_enable(pwm32_dev* dev);
23 extern void pwm32_disable(pwm32_dev* dev);
24 extern int pwm32_enabled(pwm32_dev* dev);
25 extern int pwm32_set_duty(pwm32_dev* dev, unsigned int duty);
26 extern int pwm32_set_freq(pwm32_dev* dev, unsigned int freq);
```

Die unteren Deklarierungen der Funktionsprototypen bilden die vom Anwender verwendbare Treiberschnittstelle. Alle Funktionsnamen beginnen wieder mit dem Namen der Komponente (konsistente Namespaces). Bereits hier ist zu sehen, dass bis auf den Fall „open“ ein Zeiger auf die Datenstruktur der Treiberinstanz verwendet wird, um das jeweilige Gerät zu identifizieren. Die Implementierung der Schnittstelle erfolgt dann in der zugehörigen C-Datei.

```
1  /*
2   * Used by the auto-generated file
3   * alt_sys_init.c to create an instance of this device driver.
4   */
5  #define PWM32_INSTANCE(name, dev)     \
6      pwm32_dev dev =                   \
7          {                             \
8          ALT_LLIST_ENTRY,              \
9          name##_NAME,                  \
10         ((void*)( name##_BASE)),      \
11         0,                            \
12         0,                            \
13         0,                            \
14         0,                            \
15         0                             \
16         }
17
```

```
18 /*
19  * The macro PWM_INIT is used by the auto-generated file
20  * alt_sys_init.c to initialize an instance of the device driver.
21  */
22 #define PWM32_INIT(name, dev)        \
23     pwm32_init(&dev)
24 
25 #endif /*PWM32_H_*/
```

In diesem letzen Abschnitt der Headerdatei werden nun die zuvor angesprochenen Makros für die Initialisierung definiert. Das `_INSTANCE` Makro allokiert statisch die Datenstruktur und füllt sie mit Default-Werten. An dieser Stelle werden der Name der Komponenten-Instanz sowie deren Basisadresse übergeben. Das `_INIT` Makro erledigt den Aufruf der Initialisierung-Routine.

Es folgen nun einige Ausschnitte aus der zugehörenden C-Datei, in welcher die eigentliche Treiberfunktionalität implementiert ist (`pwm32/HAL/src/pwm32.c`).

```
1  #include <stddef.h>
2  #include <errno.h>
3  #include "alt_types.h"
4  #include "sys/alt_errno.h"
5  #include "priv/alt_file.h"
6  
7  #include "pwm32.h"
8  #include "pwm32_regs.h"
9  
10 
11 /*
12  * The list of registered pwm32 components.
13  */
14 
15 ALT_LLIST_HEAD(pwm32_list);
16 
17 /*
18  * Initialize pwm driver
19  */
20 int pwm32_init(pwm32_dev* dev)
21 {
22     int ret_code;
23 
24     /* init semaphore */
25     ret_code = ALT_SEM_CREATE (&dev->lock, 1);
26     /* insert into device-list */
27     if (!ret_code)
```

```
28      {
29          ret_code = alt_dev_llist_insert((alt_dev_llist*) dev, &pwm32_list);
30      }
31      else
32      {
33          ALT_ERRNO = ENOMEM;
34          ret_code = -ENOMEM;
35      }
36
37      return ret_code;
38  }
```

Die hier gezeigte Initialisierungs-Routine erstellt das Semaphor und erledigt die Registrierung der Treiber-Instanz beim System.

```
1   pwm32_dev* pwm32_open(const char* name)
2   {
3       pwm32_dev* dev;
4       dev = (pwm32_dev*) alt_find_dev (name, &pwm32_list);
5
6       if (dev == NULL) {
7           ALT_ERRNO = ENODEV;
8       }
9
10      return dev;
11  }
12
13  void pwm32_close(pwm32_dev* dev)
14  {
15      /*   */
16  }
```

Mittels der `open()`-Funktion wird das gewünschte Gerät per Namen (s. u.) identifiziert und ein Zeiger auf die zugehörige Datenstruktur zurückgeliefert. Die entsprechende `close()`-Funktion besitzt in diesem Fall keine Funktionalität, da beim Öffnen des Geräts keine Vorgänge stattfinden, die später beim Schließen wieder rückgängig gemacht werden müssten. Bei der Verwendung von dynamischen Speicherzuweisungen müssten die entsprechenden Bereiche wieder freigegeben werden.

```
1
2   void pwm32_enable(pwm32_dev* dev)
3   {
4       void* base = dev->base;
5
6       /* begin critcal section */
```

```
7     ALT_SEM_PEND(dev->lock, 0);
8
9     if(!dev->enabled) {
10        IOWR_PWM32_ENABLE(base, 1);
11        dev->enabled = 1;
12    }
13
14    /* end critical section */
15    ALT_SEM_POST(dev->lock);
16 }
17
18 int pwm32_enabled(pwm32_dev* dev)
19 {
20    unsigned int enabled = 0;
21
22    /* begin critcal section */
23    ALT_SEM_PEND(dev->lock, 0);
24
25    enabled = dev->enabled;
26
27    /* end critical section */
28    ALT_SEM_POST(dev->lock);
29
30    return enabled;
31 }
```

Obige Funktionen zeigen exemplarisch die Implementierung von Benutzerzugriffen auf das Gerät. Sehr wichtig ist in diesem Fall die Verwendung von per Semaphor geschützten kritischen Abschnitten, um einen konkurrierenden Zugriff mehrerer Tasks sicher abhandeln zu können. Es werden hier wieder die Wrapper von Altera verwendet. Beachten Sie, dass nur die unbedingt notwendigen Bereiche (Zugriffe auf die gemeinsame Datenstruktur sowie auf die Hardware selbst) geschützt werden, um die Performanz nicht mehr als notwendig zu beeinträchtigen.

Die Funktion `pwm32_enabled()` zum Abfragen des aktuellen Status zeigt, wie durch die Verwendung der internen Datenstruktur des Treibers vergleichsweise aufwändige Hardwarezugriffe vermieden werden können.

Zuletzt soll noch die etwas interessantere Funktion zum Setzen eines neuen Duty-Cycle mit Angabe in Prozent gezeigt werden:

```
1 int pwm32_set_duty(pwm32_dev* dev, unsigned int duty)
2 {
3     unsigned int cmp;
4     void* base = dev->base;
```

```
5
6     if(duty <= 100) {
7
8         /* begin critcal section */
9         ALT_SEM_PEND(dev->lock, 0);
10
11        if(dev->enabled)
12            IOWR_PWM32_ENABLE(base, 0);;
13
14        cmp = (dev->div * duty) / 100;
15        dev->cmp = cmp;
16        dev->duty = duty;
17
18        IOWR_PWM32_CMP(base, cmp);
19
20        if(dev->enabled)
21            IOWR_PWM32_ENABLE(base, 1);
22
23        /* end critical section */
24        ALT_SEM_POST(dev->lock);
25
26        return 0;
27    } else {
28        return -1;
29    }
30
31 }
```

Inhalt des Makefiles `pwm32/HAL/src/component.mk`:

```
1 C_LIB_SRCS += pwm32.c
```

Es folgt nun ein Anwendungsbeispiel des oben in Ausschnitten vorgestellten Treibers. Um aus dem Anwendungsprogramm auf die Treiberfunktionen zugreifen zu können, ist lediglich das Einbinden der Headerdatei `pwm32.h` erforderlich.

```
1 pwm32_dev* pwm_r;
2 pwm_r = pwm32_open("/dev/pwm_rechts");
3 if(pwm_r != NULL) {
4   pwm32_set_freq(pwm_r, 10000);
5   pwm32_set_duty(pwm_r, 50);
6   pwm32_enable(pwm_r);
7 }
```

Der Name der Komponenten-Instanz lautet in diesem Fall „pwm_rechts“ und wurde im Qsys-Builder bei der Erstellung des Systems vergeben. Dieser ist auch in der Datei `systems.h` zu finden. Über den Zeiger `pwm_r` kann dann nach erfolgreichem Öffnen auf das Gerät zugegriffen werden.

Aufgabe 1

Erstellen Sie nach dem oben vorgestellten Schema einen zum Altera-HAL kompatiblen Treiber für ihre PWM-Hardware. Übertragen Sie Ihre bereits implementierten Treiberfunktionen auf die neue Schnittstelle. Testen Sie Ihre Implementierung ausführlich. Hilfreich könnte sich dabei wieder der Software-Debugger des Nios II SBT erweisen, da er die Variablen und die Datenstruktur inklusive der Zeiger automatisch auflösen und anzeigen kann.

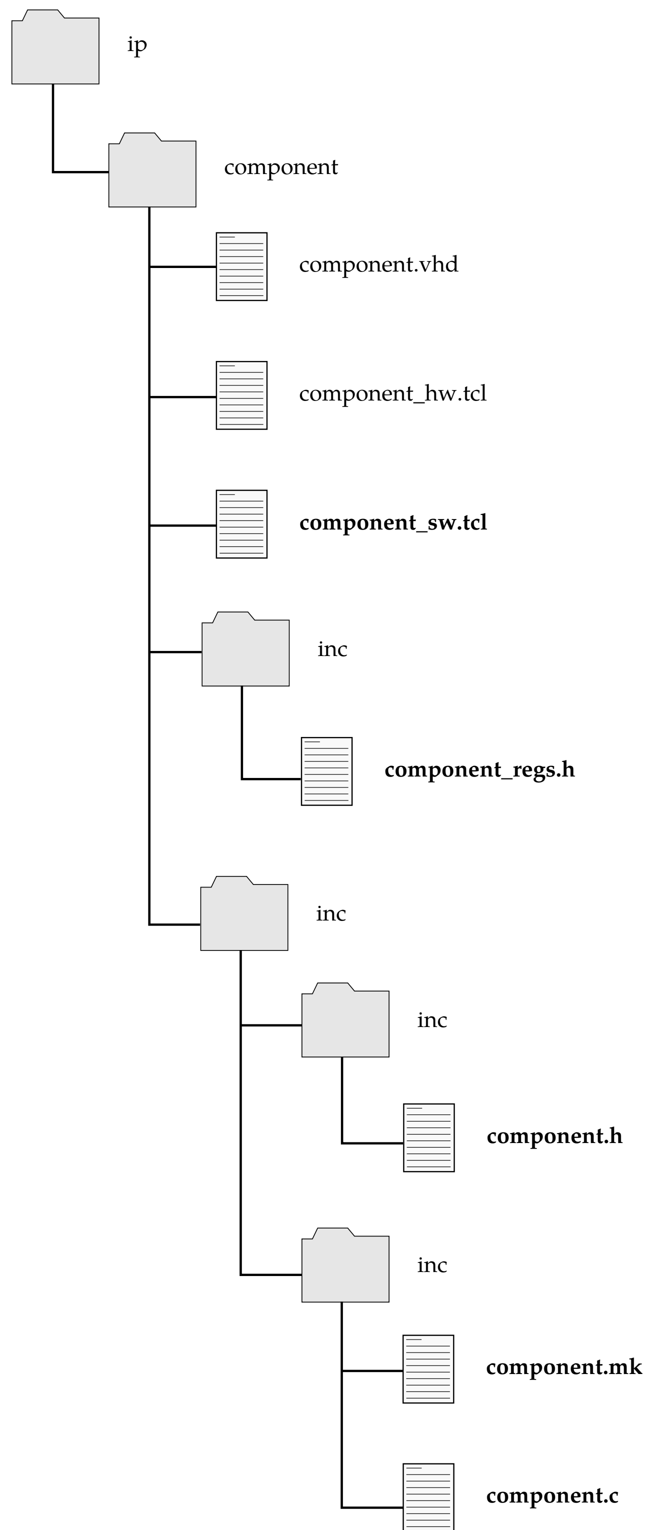

Abb. 6.1 Verzeichnisstruktur eines Treibers

Kapitel 7

Mini-Betriebssystem

Das folgende Kapitel führt durch die Entwicklung eines kleinen Betriebsystems, das für die quasinebenläufige Abarbeitung mehrerer Tasks geeignet ist. Es sollen dabei verschiedene Möglichkeiten der Realisierung vorgestellt werden. Allen gemein ist dabei, dass kein wirklicher Wechsel des Prozessorkontextes vorgenommen wird, während dies bei „richtigen" Betriebssystemen eine elementare Funktion darstellt. Auch weitere Aufgaben eines Betriebssystems wie beispielsweise die Verwaltung von Ressourcen, sollen an dieser Stelle nicht betrachtet werden.

Die Grundidee der Mini-Betriebssysteme ist wie auch bei ihren großen Vorbildern die Verwendung von Interrupts, um einen laufenden Task unterbrechen zu können. Um die Notwendigkeit eines Kontextwechsels zu vermeiden, sollen zunächst aber für jeden Task ein separater Interrupt verwendet werden. Daraus ergibt sich unmittelbar ein erster Lösungsansatz, nämlich die Verwendung von Interrupt-Service-Routinen als Tasks. Die Interrupts sind untereinander priorisiert, daher lassen sich prioritätenbasierte Scheduling-Verfahren mit so einem System abbilden.

Standardmäßig lassen sich die Interrupt-Service-Routinen beim Nios II nicht unterbrechen, auch nicht durch höherpriore Tasks oder Interrupts. Die Auslösung der Interruptrequests kann nun durch Geräte erfolgen, für zeitbasierte Anforderungen können Timer verwendet werden.

Aufgabe 1

Erstellen Sie mittels der oben vorgestellten Idee ein eigenes kleines Betriebssystem, welches die Abarbeitung von mindestens vier Tasks ermöglicht. Um einfacher an reproduzierbare Ergebnisse zu gelangen, sollen alle notwendigen Interrupts von unabhängigen Timern erzeugt werden. Um die

volle Programmierbarkeit der Timer zu erreichen, sollen diese im „full-featured“ Modus eingebunden werden. Passen Sie Ihr Qsys-Design entsprechend an.

Erstellen Sie dann die entsprechende Software, bei der die Tasks direkt in den ISRs ablaufen. Für einen ersten Versuch soll jeder Task einen eigenen Zähler bei jeder Ausführung inkrementieren. In der main()-Funktion sollen zunächst die Timer konfiguriert werden und dann in einer Schleife ein Idle-Zähler inkrementiert sowie periodisch eine Übersicht über die Zähler auf der Konsole ausgegeben werden. Funktionen für den Zugriff auf die Timer finden Sie in der Datei `altera_avalon_timer_regs.h`.

Experimentieren Sie mit verschiedenen Perioden und überprüfen Sie die Einhaltung der Prioritäten. Innerhalb der ISRs können Sie durch aktives Warten mittels `usleep()` Prozessorauslastung generieren und somit unterschiedlich lange Ausführungszeiten simulieren. Weiterhin kann es sinnvoll sein, erweiterte (statistische) Analysen der Zählerwerte in der Hauptschleife durchzuführen.

Aufgabe 2

Sollen die Tasks nicht innerhalb der ISRs, sondern im Kontext des Hauptprogramms abgearbeitet werden, beispielsweise, weil blockierende Funktionen verwendet werden sollen, so bietet sich eine zweite Möglichkeit für die Implementierung eines Mini-Betriebssystems an.

Die jeweiligen ISRs werden nur dafür benutzt, die Tasks auf „aktiv“ zu setzen, während in der main()-Funktion eine Schleife abläuft, welche die Tasks sortiert nach ihrer Priorität auf eben jene Aktivitätsanforderung überprüft und dann bei Bedarf die entsprechende Funktion des Tasks aufruft. Zuvor muss verständlicherweise das Signal zurückgesetzt werden. Wird nach der Abarbeitung eines Tasks sichergestellt, das die Schleife direkt wieder von vorn mit der Überprüfung beginnt, so ist sichergestellt, dass immer der Task mit der höchsten aktiven Priorität läuft.

Setzen Sie nun diese Idee in einem eigenen System um. Mit geeigneter Wahl einer Datenstruktur für die Verwaltung der Tasks sowie der Verwendung von Funktions-Pointern können Sie die Scheduler-Schleife sehr einfach und unabhängig von der Anzahl an Tasks gestalten.

Führen Sie wie in Aufgabe 1 verschiedene Experimente durch und untersuchen Sie das Verhalten des Systems.

Freiwillige Sonderaufgabe

Bei zeitgesteuerter Taskaktivierung ist es sehr unschön, auf Grund unterschiedlicher Perioden für jeden Task einen eigenen Hardware-Timer verwenden zu müssen. Entwickeln Sie daher eine möglichst flexible Lösung, um alle zeitbasierten Aktivierungen von Tasks von nur einem Timer erledigen zu lassen. Dabei sollen die Perioden auch während der Laufzeit konfigurierbar sein. Beachten Sie die Bedeutung der Granularität des entsprechenden Timers.

Kapitel 8

MicroC/OS-II

In diesem Kapitel wollen wir uns nun einem „richtigen“ Betriebssystem zuwenden, dem Echtzeitbetriebssystem MicroC/OS-II (eigentlich „μC/OS-II“) von Jean J. Labrosse, welches er über seine Firma Micriμm vermarktet. Das System wird ausführlich (jede Zeile des Quellcodes) im gleichnamigen Buch des Verlags CMP Books erläutert.

Für die Nutzung mit der Nios II CPU wurde das System von Altera portiert und der Altera-HAL entsprechend angepasst. So werden beispielsweise einige Makros bereitgestellt, die je nach Konfiguration im Multitasking-Betrieb Betriebssystemfunktionen und im Singletask-Betrieb Funktionen des HAL verwenden. Weitere Informationen entnehmen Sie bitte Kaptel 10 des Nios II Software Developer's Handbook.

Aufgabe 1

Erstellen Sie ein System auf Grundlage des MicroC/OS. Das zu Grunde liegende Nios II System benötigt dafür nur noch einen Timer, den Systemtimer, der die höchste Interrupt-Priorität besitzen sollte und in diesem Fall eine Periode von 1 ms (also eine Frequenz von 1 kHz). Er muss nicht programmierbar sein. Im Nios II SBT erstellen Sie dann ein neues Projekt basierend auf dem Beispiel „Hello MicroC/OS-II“. Bitte ändern Sie die Prioritäten der vorhandenen Beispieltasks (Task 0 und Task 1) auf 4 und 5, da die obersten sowie untersten vier Prioritäten eigentlich für das Betriebssystem reserviert sind.

Starten Sie das System und versuchen Sie, die Funktionsweise nachzuvollziehen.

Aufgabe 2

Der Software-Debugger des Nios II SBT erkennt die Tasks eines MicroC/OS Systems und kann sie und ihren Funktions-Stack im pausierten Modus anzeigen. Ein manuelles „Umschalten“ zwischen den Tasks ist aber verständlicherweise nicht möglich.

Führen Sie im Debugger den Start des Systems Schritt für Schritt aus (Stepping). Können Sie den Beginn des Multitasking-Betriebs erkennen? Es gibt außer den explizit erstellen Tasks noch den Idle-Task, der mit der niedrigsten Priorität läuft sowie je nach Konfiguration noch einen Statistik-Task, für den die zweitniedrigste Priorität vorgesehen ist. Können Sie die einzelnen Tasks identifizieren?

Experimentieren Sie auch mit Breakpoints innerhalb der Tasks.

Aufgabe 3

Der Statistik-Task stellt jede Sekunde die aktuelle Auslastung der CPU (in Prozent) in der globalen Variable `OSCPUUsage` bereit. Dies funktioniert allerdings erst nach der Initialisierung mittels der Funktion `OSStatInit()`, für deren korrekte Verwendung das Beispielprogramm etwas modifiziert werden muss.

Die Auslastung des Prozessors soll natürlich nur die Benutzer-Tasks umfassen, da der Idle- und Statistiktasks nicht wichtig für die Funktion des Systems sind. Daher muss die Initialisierung mittels `OSStatInit()` genau dann erfolgen, wenn der Multitasking-Betrieb bereits aktiviert ist, aber noch keine Benutzerfunktionen ablaufen. Ausser den Idle- und Statistik-Tasks soll daher zur Zeit der Initialisierung nur ein einziger weiterer Task existieren, nämlich genau der, welcher die Initialisierungsfunktion aufruft. Üblicherweise erstellt dieser dann zunächst die weiteren Tasks und führt dann (bei Bedarf) eigene Aufgaben aus.

Passen Sie Ihr System entsprechend an, um die Auslastung der CPU regelmäßig ausgeben zu können. Erstellen Sie dafür einen weiteren Task, der wie beschrieben nach dem Beginn des Multitasking-Betriebs zunächst die Funktion `OSStatInit()` aufruft und dann die restlichen Tasks erstellt. Seine weitere Funktionalität könnte dann beispielsweise die regelmäßige Ausgabe der Systemauslastung sein. Achten Sie in diesem Fall darauf, die Priorität des Tasks nicht zu hoch zu wählen. Da die Prioritäten `OS_LOWEST_PRIO` bis `OS_LOWEST_PRIO - 3` für das Betriebssystem reserviert sind, ist die niedrigste, Ihnen zur Verfügung stehende Priorität `OS_LOWEST_PRIO - 4`.

Experimentieren Sie mit unterschiedlichen Auslastungen des Systems, beispielsweise, indem einige Ihrer Tasks einen Teil mit aktivem Warten verbringen.

Aufgabe 4

Übertragen Sie die Experimente aus dem vorigen Kapitel auf das Echtzeitberiebssystem. Wo gibt es Gemeinsamkeiten, wo Unterschiede im Ablauf?

Kapitel 9

Lüfterregelung

Als abschließendes Projekt wird nun ein etwas komplexeres eingebettetes System aus einer Kombination von Hard- und Software entwickelt und getestet. Es soll eine Motorregelung entworfen und implementiert werden, die ein System mit einem Gleichstrommotor auf einer konstanten Drehzahl hält. Als Grundlage für die Ansteuerung dient dabei das bereits entwickelte PWM-Modul. Für die Rückkopplung der Motordrehzahl muss noch ein weiterer Hardwarebaustein erstellt werden, der das Tachosignal des Motors auswerten kann. Die Regelung des Systems soll dann in Software auf einem integrierten Nios II Softcore-Prozessor erfolgen.

Aufgabe 1 - Drehzahlmessung - Hardware

Der zu regelnde Motor besitzt ein rechteckiges Tachosignal mit mehreren Impulsen pro Umdrehung, wobei das Signal in den üblichen Drehzahlbereichen einen Duty-Cycle von etwa 50% aufweist. Entwickeln Sie eine Hardwarekomponente in VHDL, welche aus einem gegebenen Tachosignal die Motordrehzahl ermittelt und über den Avalon-Bus von einem Nios II Prozessor ausgelesen werden kann. Berücksichtigen Sie in Ihrem Konzept, welche der Funktionen und Umrechnungen in Hardware ablaufen müssen/sollen und welche sinnvoller in der Treibersoftware implementiert werden. Da es bei dem von uns verwendeten Tachosignal je nach Motor nur einen bis acht Pulse pro Umdrehung gibt, erscheint es aus Gründen der Messgenauigkeit wenig sinnvoll, die Anzahl der Flanken in einem festen Zeitraum zu zählen. Deutlich genauer und mit einer höheren Update-Rate bzw. geringeren Latenz verbunden ist die Messung der Zeitdauer zwischen den Impulsen oder die Dauer der Impulse. Bezogen auf das Rechteck-Signal also die Dauer zwischen zwei gleichartigen Flanken (beide steigend bzw. fallend) oder die Dauer von einer steigenden zur nächsten fallenden Flanke (oder invers). Das entsprechende Messverhalten (die Wahl der Flanken) soll in Ihrer Komponente zur Laufzeit änderbar

sein. Überlegen Sie sich nun, wie Sie die jeweiligen Zeiten messen können und implementieren Sie eine Komponente, welche diese Messungen durchführt. Sie können davon ausgehen, dass der Motor mindestens eine Umdrehung pro Sekunde macht. Leider kommt es nach den Flanken jeweils zu einem Prellen von einigen μs, Sie sollten daher bereits in der Hardwarekomponente eine Entprellung vorsehen. Legen Sie diese so an, dass deren Dauer zur Laufzeit konfigurierbar ist.

Aufgabe 2 - Drehzahlmessung - Hardwareanbindung

Erweitern Sie Ihr Modul aus Teil 1 nun um eine Anbindung an den Avalon-Bus. Überlegen Sie, welche Daten in welche Richtung übertragen werden müssen (Messdaten, Konfigurationsdaten, etc.). Machen Sie sich Gedanken um die Datenformate und eine sinnvolle Aufteilung in vom Prozessor adressierbare Register. Behalten Sie bei allen diesen Überlegungen auch die Treiber auf der Softwareseite im Hinterkopf. An welche Stelle wird der Teilfaktor ($1 - 8$ Impulse pro Umdrehung) berücksichtigt, wo erfolgt die Umrechnung in Umdrehungen pro Minute? Wie kann der Treiber zur Laufzeit das Messverhalten konfigurieren (Wahl der Flanken)?

Aufgabe 3 - Drehzahlmessung - Software

Erstellen Sie einen Treiber für das zuvor entwickelte Hardwaremodul zur Drehzahlmessung. Schreiben Sie Funktionen, welche die grundlegenden Aufgaben erfüllen wie

- Rücklieferung der Motorgeschwindigkeit in Umdrehungen pro Minute
- Anpassung an den Motor bzw. das Tachosignal (Pulse pro Umdrehung)
- Konfiguration des Messverhaltens (Flanken)

Die bereitgestellte Platine erwartet am Stecker auf Pin 2 das PWM-Signal für den Motor und liefert auf Pin 4 das Tachosignal zurück. Dies entspricht dann den Signalen `GPIO_x(1)` bzw. `GPIO_x(3)`.

Aufgabe 4 - Regelung

Zu einer lauffähigen Motorregelung fehlt nun noch die Software, welche die eigentliche Regelung durchführt. Einen Überblick über die Methoden der Regelungstechnik ist beispielsweise unter
`http://www.rn-wissen.de/index.php/Regelungstechnik`
zu finden.

Erstellen Sie nun ein Softwareprojekt, das eine Regelschleife enthält, die die Motordrehzahl auf einem vorgegebenen, zur Laufzeit wählbaren Wert hält. Dafür sollte bereits ein PI-Regler ausreichen. Zu empfehlen ist ein PI-Regler mit Vorsteuerung, welcher die Struktur in Abbildung 9.1 haben solle. Können Sie mit einem PID-Regler ein besseres Ergebnis erzielen?

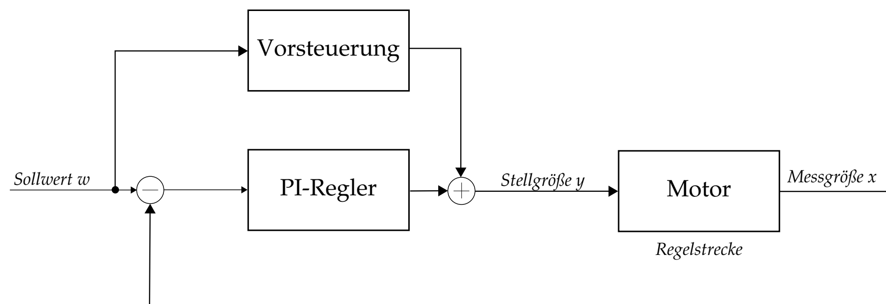

Abb. 9.1 Struktur des PI-Reglers mit Vorsteuerung

Verwenden Sie das Echtzeitbetriebssystem MicroC/OS-II und teilen Sie die benötigten Funktionen sinnvoll in verschiedene Tasks auf. Zu trennen sind beispielsweise die Regelschleife, Ausgaben, Benutzereingaben, etc.

Einige Anmerkungen, die Ihnen vielleicht bei der Implementierung helfen können:

- Beginnen Sie mit einem P-Regler. Welchen Effekt können Sie beobachten?
- Wird der Lüfter mit der Saug-Seite flach auf den Tisch gelegt, so kann er höhere Drehzahlen leichter erreichen.
- Geben Sie die gemessene Drehzahl und den Duty-Cycle der PWM aus. So können Sie sofort feststellen, wenn die Störgröße (Belastung) zu groß ist und nicht mehr kompensiert werden kann.
- Stellen Sie sicher, dass die Regelschleife nicht zu schnell nacheinander durchlaufen wird
- Berücksichtigen Sie den Schleifenzyklus bei der Bestimmung der I- und D-Anteile

- Schließen Sie aus, dass unsinnige Werte des Tachosignals in die Regelung einfließen
- Begrenzen Sie die Fehlersumme (Wind-Up-Effekt)
- Benutzen Sie zur Eingabe der Regelparameter die Tasten auf dem Board
- Bei geringer Spannung am Lüfter gibt es Fehler im Tachosignal bzw. fällt es ganz aus, so dass Ihre Regelung instabil werden kann (also Vorsicht beim zu schnellen/starken Herunterregeln)
- Wählen Sie bei der Frequenz Ihrer PWM eher niedrigere Werte (ca. 1 kHz)
- Welche Auswirkungen hat das Ändern des Duty-Cycle in ihrer PWM-Komponente im laufenden Betrieb? Wie können Sie eventuell daraus resultierende Probleme vermeiden?
- Wie genau ist Ihre Regelung? Wie schnell erreicht der Lüfter seine Sollgeschwindigkeit aus dem Stillstand? Wie groß ist das Überschwingen dabei?

Bei der endgültigen Realisierung der *Lüfterregelung* soll dann die Soll-Drehzahl über die Taster (Interrupts verwenden) eingegeben werden und die Ist- und Soll-Werte für die Umdrehungen pro Minute auf dem LCD angezeigt werden.

Aufgabe 5 - Dokumentation

Die Umsetzung des Projektes *"Lüfterregelung"* soll in einem kurz gehaltenen Abschussbericht festgehalten werden. Dieser muss beantworten, warum eine Umsetzung des Projektes mittels eines Hardware/Software Designs sinnvoll ist, warum bestimmte Teile in Hardware und andere Teile in Software realisiert wurden, was die Funktionsweise von Treibern ist, wie Ihre Hardwarekomponenten mit Registern aussehen und wie Sie Ihre Treiber und Hardwarekomponente realisiert haben. Weiterhin soll der Einsatz eines Echtzeitbetriebssystems begründet werden sowie eine Erklärung von Tasks. Auch hier muss auf ihre Realisierung eingegangen werden und erklärt werden wie Sie die Aufgabe umgesetzt haben.