Kapitel 6

Treiberentwicklung

Dieses Kapitel soll an die Vorgehensweise der Entwicklung vollständiger, modularer Treiber heranführen. Um eine sinnvolle Verwendbarkeit innerhalb des Laborprojekts 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_irq_register()
2 alt_irq_disable()
3 alt_irq_enable()
```

```
4 alt_irq_disable_all()
5 alt_irq_enable_all()
6 alt_irq_interruptible()
7 alt_irq_non_interruptible()
8 alt_irq_enabled()
```

Um eine Funktion beim HAL als ISR bekannt zu machen, wird die Funktion

```
1 int alt_irq_register (alt_u32 id,
2                       void* context,
3                       void (* isr)(void*, alt_u32))
```

verwendet, wobei `id` die zugehörige Interrupt-ID, `context` einen Void-Pointer auf einen frei wählbaren Interrupt-Kontext und `isr` ein Funktionspointer auf die ISR sein soll. Daraus lässt sich unmittelbar der Prototyp einer ISR ablesen, der folgendermaßen aussehen muss:

```
1 void <isr_name>(void* context, alt_u32 id)
```

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.

```
1 #include "system.h"
2 #include "altera_avalon_pio_regs.h"
3 #include "alt_types.h"
4
```

```
5  static void handle_button_interrupts(void* context, alt_u32 id)
6  {
7      /* Cast context to edge_capture's type. It is important that this
8       *  is declared volatile to avoid unwanted compiler optimization.
9       */
10     volatile int* edge_capture_ptr = (volatile int*) context;
11
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_irq_register( BUTTON_PIO_IRQ,
22                        edge_capture_ptr,
23                        handle_button_interrupts );
24  }
```

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_irq_disable(BUTTON_PIO_IRQ);
3
4  ... /* critical section */
5
6  /* Re-enable interrupt */
7  alt_irq_enable(BUTTON_PIO_IRQ);
```

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 SOPC-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 Verfü-

gung 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 IDE ermöglicht im Zusammenspiel mit dem Altera-HAL und den vom SOPC-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 SOPC-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; muss auf dieses Verzeichnis angepasst bzw. neu erzeugt werden) abgelegt. Die Treiberdateien werden dann in zwei verschiedene Unterverzeichnisse aufgeteilt, `inc` und `HDL`. 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.

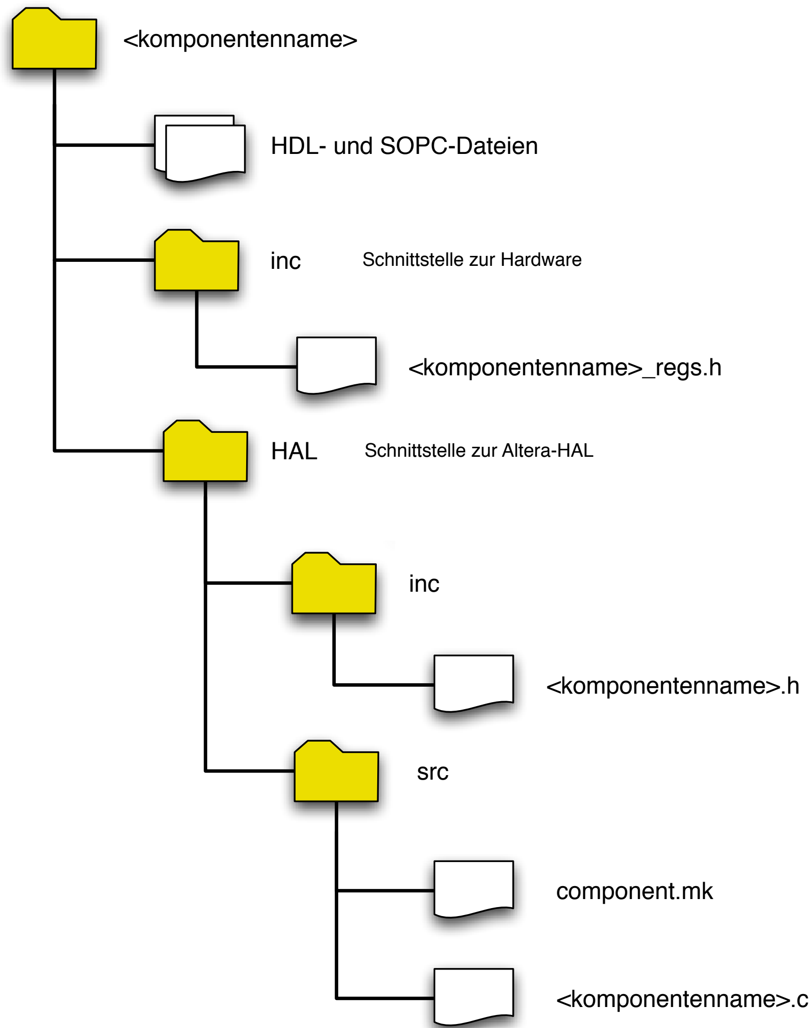

Abb. 6.1 Verzeichnisstruktur eines Treibers

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.

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):

```
1  #ifndef PWM32_REGS_H_
2  #define PWM32_REGS_H_
3
4  #include <io.h>
5
6  /* Register of compare value */
7  #define IOADDR_PWM32_CMP(base)         __IO_CALC_ADDRESS_DYNAMIC(base, 0)
8  #define IORD_PWM32_CMP(base)           IORD_32DIRECT(base, 0)
9  #define IOWR_PWM32_CMP(base, VALUE)    IOWR_32DIRECT(base, 0, VALUE)
10
11 /* Register of divider value */
12 #define IOADDR_PWM32_DIV(base)         __IO_CALC_ADDRESS_DYNAMIC(base, 4)
13 #define IORD_PWM32_DIV(base)           IORD_32DIRECT(base, 4)
14 #define IOWR_PWM32_DIV(base, VALUE)    IOWR_32DIRECT(base, 4, VALUE)
15
16 /* Enable register */
17 #define IOADDR_PWM32_ENABLE(base)        __IO_CALC_ADDRESS_DYNAMIC(base, 8)
18 #define IORD_PWM32_ENABLE(base)          IORD_32DIRECT(base, 8)
19 #define IOWR_PWM32_ENABLE(base, VALUE)   IOWR_32DIRECT(base, 8, VALUE)
```

```
20
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.

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

extern alt_dev* alt_find_dev (const char* name, alt_llist* list);


/*
 * Called by alt_sys_init.c to initialize the driver.
 */
extern int pwm32_init(pwm32_dev* dev);

/*
 * Public driver interface
 */

extern pwm32_dev* pwm32_open(const char* name);
extern void pwm32_close(pwm32_dev* dev);

extern void pwm32_enable(pwm32_dev* dev);
extern void pwm32_disable(pwm32_dev* dev);
extern int pwm32_enabled(pwm32_dev* dev);
extern int pwm32_set_duty(pwm32_dev* dev, unsigned int duty);
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.

```
/*
 * Used by the auto-generated file
 * alt_sys_init.c to create an instance of this device driver.
 */
#define PWM32_INSTANCE(name, dev)     \
    pwm32_dev dev =                   \
        {                             \
        ALT_LLIST_ENTRY,              \
        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`:

```
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.

```
pwm32_dev* pwm_r;
pwm_r = pwm32_open("/dev/pwm_rechts");
if(pwm_r != NULL) {
    pwm32_set_freq(pwm_r, 10000);
    pwm32_set_duty(pwm_r, 50);
    pwm32_enable(pwm_r);
}
```

Der Name der Komponenten-Instanz lautet in diesem Fall „pwm_rechts" und wurde im SOPC-Builder bei der Erstellung des Systems vergeben. 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 der Nios II IDE erweisen, da er die Variablen und die Datenstruktur inklusive der Zeiger automatisch auflösen und anzeigen kann.