Praktikum Softprocessor SP6: Interrupt

2. Juli 2014

Zusammenfassung

Die bisherige Hardware-Konfiguration wird um einen Interrupt-Controller erweitert. Anschließend sind typische Programmieraufgaben mit einer Zähler/Zeitgebereinheit durchzuführen.

1 Interrupt

In der Praxis wartet ein Rechner bzw. ein Task die meiste Zeit auf die Bereitstellung von Eingaben und Ausgaben oder bei Prozessrechnern auf das Verstreichen einer Wartezeit, den nächsten Prozessschritt etc. Für die Programmierung von Wartevorgängen gibt es zwei Möglichkeiten:

- Polling und
- Interrupt.

Das Polling-Verfahren wurde bereits verwendet, z.B. um auf den Empfang eines Zeichens von der seriellen Schnittstelle oder ein Zeitgeberereignis zu warten. Dazu wurde in einer Warteschleife das jeweilige Statusregister der peripheren Einheit gelesen und geprüft, ob das zugehörige Ereignisbit gesetzt ist. In der vorherigen Anleitung in Abschnitt 1.2 wurde auch bereits eine Software-Architektur vorgestellt, in der gleichzeitig auf mehrere Ereignisse gewartet wird. Das ist einer Schleife, in der zyklisch nacheinander alle Ereignisbits abgefragt werden und, wenn eines gesetzt ist, die zugehörige Aufgabe ausgeführt wird. In dieser Architektur ist die garantierbare Obergrenze für die Reaktionszeit vom Setzen des Ereignisbits bis zur Abarbeitung der zugehörigen Aufgabe die Summe der Ausführungszeiten aller anderen Aufgaben, die im ungünstigsten Fall vorher an der Reihe sind. Das können mehreren Millisekunden oder gar Sekunden sein. Um auch kürzere Reaktionszeiten garantieren zu können, unterstützt praktisch jeder Prozessor noch eine zweite Art der ereignisgesteuerte Verarbeitung, den Interrupt-Betrieb.

Im Interrupt-Betrieb unterbricht der Prozessor, wenn ein Ereignisbit gesetzt wird, seine aktuelle Aufgabe, sichert alle Daten für die Weiterverarbeitung und startet den Interrupt-Handler. Ein Prozessor hat mindestens zwei reservierte Hardware-Adressen im Befehlsspeicher. Die eine ist die Programmstartadresse nach einem Reset, normalerweise die Adresse null, und die andere die Interruptadresse[1]. Auf der Interruptadresse erwartet der Prozessor ein Unterprogramm, dass er bei Aktivierung des Unterbrechungssignals aufruft. Dieses Unterprogramm sichert außer der Rücksprungadresse alle Registerinhalte des Prozessors, die sie verändert und ruft den Interrupt-Handler auf. Der Interrupt-Handler fragt nacheinander die Ereignisbits, für die Interrupts vorgesehen sind, ab und startet für die gesetzten Ereignisbits die zugehörigen Interruptroutinen. Nach Rückkehr vom Interrupt-Handler werden die gesicherten Registerinhalte restauriert und zum unterbrochen Programm zurückgekehrt. Die gesamte ab der Interrupteinsprungadresse bis zur Rückkehr zum Unterbrechungspunkt abgearbeitet Anweisungsfolge incl. Interrupt-Handler und den Interruptroutinen ist zur Vermeidung rekursiver Aufrufe des Interrupt-Handlers selbst nicht unterbrechbar.

Die einzelnen Interruptroutinen sollen möglichst kurz und einfach strukturiert sein. Üblicherweise beschränkt sich der Funktionsumfang auf das Kopieren weniger Datenobjekte, die Abfrage,

[1] Diese Adressen findet man mit »rechter Mouse-Klick auf die Software-Plattform« ▷ »Generate Linker Script« in der »Boot Vector Section«. Der Microblace hat zusätzlich noch feste Einsprungadressen für die Behandlung von Ausnahmesituationen (Exceptions).

das Setzen oder Rücksetzen einzelner Bits und das Weiterschalten von Zählern. Komplizierte Berechnungen oder gar Warteschleifen in einer Interruptroutine zeugen von schlechtem Programmierstil. Denn zum einen leidet darunter die garantierbare Reaktionszeit und zum anderen sind die Interruptroutinen immer die Programmteile, die sich am schwierigsten Testen und Debuggen lassen, weil sie sich z.B. nicht im Schrittbetrieb abarbeiten lassen[2].

Der zu verwendende Interrupt-Handler

```
void XIntc_DeviceInterruptHandler(void *DeviceId);
```

ist im Header »xintc_l.h« definiert und wird direkt vom Linker, d.h. nicht über das eigenes C-Programm in das Gesamtprogramm eingebaut. Zusätzlich definiert »xintc_l.h« eine Registrierfunktion

```
void XIntc_RegisterHandler(u32 BaseAddress, int InterruptId,
                XInterruptHandler Handler, void *CallBackRef);
```

über die dem Interrupt-Handler die Basisadresse des Interrupt-Controllers, die Interruptnummer, der Funktionszeiger auf die Interrupt-Routine und der Eingabeparameter der Interrupt-Routine, die Basisadresse der interruptauslösenden Einheit übergeben wird. Darüber hinaus stehen Makros

- für die Freigabe und Sperrung aller Interrupts

  ```
  XIntc_mMasterEnable
  XIntc_mMasterDisable
  ```

- und Makros für die Freigabe, Sperrung und den Test, ob Anforderungsbit für einzelne Interrupts gesetzt sind

  ```
  XIntc_mEnableIntr(BaseAddress, EnableMask)
  XIntc_mDisableIntr(BaseAddress, DisableMask)
  XIntc_mAckIntr(BaseAddress, AckMask)
  ```

zur Verfügung (BaseAddress – Basisadresse des Interrupt-Controllers; InterruptId – Interrupt-Nummer; Handler – Funktionszeiger auf die Interrupt-Routine; CallBackRef – Parameter für die Interrupt-Routine, hier die Basisadresse des Gerätes (Timer, UART, ...) die den Interrupt auslöst; ...Mask – Maske für Bitoperationen, in der das Bit der Interrupt-Nummer eins und die anderen Bits null sind). Jede genutzte Interrupt-Quelle (Timer, UART, ...) bekommt beim Hardware-Entwurf eine Interrupt-Nummer zugeteilt. Das ist die Bitnummer in den Kontroll- und Statusregistern des Interrupt-Controllers, in dem Interrupts gesperrt, freigegeben oder abgefragt werden. Im Beispielentwurf sind im Header »xparameters.h« für die drei Interruptquellen die folgenden symbolischen Konstanten definiert:

1. Eingabeschnittstelle
   - CallBackRef: XPAR_EINGABESCHNITTSTELLE_BASEADDR
   - InterruptID: XPAR_XPS_INTC_0_EINGABESCHNITTSTELLE_IP2INTC_IRPT_INTR = 0
   - Maske: XPAR_EINGABESCHNITTSTELLE_IP2INTC_IRPT_MASK = 1
2. UART
   - CallBackRef: XPAR_UART_BASEADDR
   - InterruptID: XPAR_XPS_INTC_0_UART_INTERRUPT_INTR =1

[2]Die Programmunterbrechung im Debuggermodus nach jeder Anweisung, an vorgegebenen Unterbrechungspunkten etc. um Werte zu kontrollieren oder zu verändern erfolgt über Interrupts. Interruptroutinen sind jedoch zur Verhinderung rekursiver Aufrufe generell nicht unterbrechbar.

- Maske: XPAR_UART_INTERRUPT_MASK = 2

3. Timer
   - CallBackRef: XPAR_XPS_TIMER_0_BASEADDR
   - InterruptID: XPAR_XPS_INTC_0_XPS_TIMER_0_INTERRUPT_INTR = 2
   - Maske: XPAR_XPS_TIMER_0_INTERRUPT_MASK = 4

Der Maskenwert ist dabei immer zwei hoch Interrupt-Nummer. Eigene Programme, die Interrupts nutzen wollen, müssen zuerst für jede Interrupt-Quelle eine Interrupt-Routine definieren, diese zu Beginn des Hauptprogramms registrieren lassen und die zugehörigen Interrupt freigeben. Zusätzlich muss die periphere Einheit für die Generierung von Interrupts vorbereitet werden.

1.1 Timer-Interrupt

Das nachfolgende Beispiel definiert ein Statuswortkonstante für die Timer-Einstellungen: Interrupts aktivieren, Ereignisflag löschen, Autoreload und Anwärtszählen. Variablen, in denen sich eine Interruptroutine Werte von einem Aufruf zum nächsten merken und Daten mit anderen Programmteilen austauschen soll, im Beispiel »x«, sind global zu vereinbaren. Die Interrupt-Routine »TimerIntr(void *baseAdress)« wird vom Interrupt-Handler aufgerufen, wenn der Timer überläuft und sein Ereignisbit setzt, aufgerufen. Der Parameter »baseAdress« ist die Basisadresse der peripheren Einheit und wir bei der Interrupt-Restrierung übergeben Im Beispiel erhöht die Interrupt-Routine den Wert der Variablen »x« um eins. Die anschließende Neuinitialisierung des Timer-Statusregisters setzt nur das Ereignisbit zurück. Das abschließende Makro löscht das Interrupt-Flag im Interrupt-Controller, damit wieder neue Interrupts erzeugt werden können.

Das Hauptprogramm registriert zuerst die Interrupt-Routine, Initialisiert der Zeitgeber, so, dass er 0,5 s das Ereignisbit setzt und gibt Interruts an vier Stellen frei:

- bei der Wertzuweisung an das Kontrollregister in der Zeitgebereinheit
- mit »Intc_mEnableIntr« lokal im Interrupt-Controller nur für die Zeitgebereinheit
- mit dem Makro »Intc_mMasterEnable« im Interrupt-Controller für alle Interrupt-Quellen
- mit dem Makro »icroblaze_enable_interrupts()« für den Prozessor.

Nach Abschluss der Initialisierung gibt das Programm in einer Endlosschleife den Wer der Variablen »x«, die in der Interrupt-Routine hochgezählt wird, auf die Leuchtdioden aus.

```
#include "xparameters.h"
#include "xgpio_l.h"
#include "xintc_l.h"
#include "xtmrctr_l.h"

const int TimerCnf = XTC_CSR_DOWN_COUNT_MASK  // abwärtszählen
                   | XTC_CSR_ENABLE_TMR_MASK  // Zähler ein
                   | XTC_CSR_INT_OCCURED_MASK // Ereignisflag. löschen
                   | XTC_CSR_AUTO_RELOAD_MASK // Autoreload ein
                   | XTC_CSR_ENABLE_INT_MASK; // Interrupt ein
char x=0;

void TimerIntr(void *baseAdress){
  x++;
  XTmrCtr_mSetControlStatusReg(baseAdress, 0, TimerCnf);
  XIntc_mAckIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_XPS_TIMER_0_INTERRUPT_MASK);
}
int main(){
// Interrupt-Routinen registrieren
   XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR, XPAR_XPS_INTC_0_XPS_TIMER_0_INTERRUPT_INTR,
                         (XInterruptHandler) TimerIntr, (void *)XPAR_XPS_TIMER_0_BASEADDR);
// Zeitgeberkanal 0 zur Erzeugung periodischer Ereignisse aller 1s initialisieren
```

```
    XTmrCtr_mSetLoadReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 50000000);
    XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0, TimerCnf);

// Interrupts im Interrupt-Controller lokal und global freigeben
    XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_XPS_TIMER_0_INTERRUPT_MASK);
    XIntc_mMasterEnable(XPAR_XPS_INTC_0_BASEADDR);

// Interrupts im Microblaze aktivieren
    microblaze_enable_interrupts();

// Endlosschleife
    while (1){
      XGpio_mSetDataReg(XPAR_AUSGABESCHNITTSTELLE_BASEADDR, 1, x);
    }
}
```

1.2 Eingabeinterrupt

Der Eingabeinterrupt wird ausgelöst, wenn sich der Wert an einem Schaltereingang der Eingabeschnittstelle ändert. Die Interrupt-Routine ist fast wie im Beispiel zuvor aufgebaut. Bei einem Aufruf wird der Wert der Variablen »x« um eins hochgezählt, das Ereignisbit in der parallelen Schnittstelle und abschließend das Interrupptbit im Interrupt-Controller gelöscht. Das Hauptprogramm beginnt mit der Registrierung der Interrupt-Routine löscht das Interrupt-Bit und gibt die Interruptannahme an vier verschiedenen Stellen in der Hardware frei: in der parallelen Schnittstelle, der Interrupt-Controller zuerst lokal dann global und abschließend im Microblaze. Die anschließede Endlosschleife gibt bei jedem Durchlauf, d.h. quasi ständig den Wer der Variablen »x«, die in der Interrupt-Rotine hochgezählt wird, auf die Leuchtdioden aus. Wenn nach der Programmierung an irgend einem der Schnittstelleneingänge ein Schalter umschaltet, zählt der auf die Leuchtdioden ausgegebene Wert um mindestens eins[3] weiter.

```
#include "xparameters.h"
#include "xgpio_l.h"
#include "xintc_l.h"

char x=0;

void BtnIntr(void *baseAdress){
  x++;
  XGpio_mWriteReg(XPAR_EINGABESCHNITTSTELLE_BASEADDR, XGPIO_ISR_OFFSET, XGPIO_IR_MASK);
  XIntc_mAckIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_EINGABESCHNITTSTELLE_IP2INTC_IRPT_MASK);
}

int main(){
// Interrupt-Routinen registrieren
    XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR, XPAR_XPS_INTC_0_EINGABESCHNITTSTELLE_IP2INTC_IRPT_IN
                          (XInterruptHandler) BtnIntr, (void *)XPAR_EINGABESCHNITTSTELLE_BASEADDR);
// Interrupts in den Kontrollregistern der Schnittstelle freigeben
    XGpio_mWriteReg(XPAR_EINGABESCHNITTSTELLE_BASEADDR, XGPIO_GIE_OFFSET, XGPIO_GIE_GINTR_ENABLE_MASK);
    XGpio_mWriteReg(XPAR_EINGABESCHNITTSTELLE_BASEADDR, XGPIO_IER_OFFSET, XGPIO_IR_MASK);

// Interrupts im Interrupt-Controller lokal und global freigeben
    XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_EINGABESCHNITTSTELLE_IP2INTC_IRPT_MASK);
    XIntc_mMasterEnable(XPAR_XPS_INTC_0_BASEADDR);

// Interrupts im Microblaze aktivieren
    microblaze_enable_interrupts();

// Endlosschleife
    while (1){
          XGpio_mSetDataReg(XPAR_AUSGABESCHNITTSTELLE_BASEADDR, 1, x);
    }
```

[3] Durch das Prellen der Schalter kommt es vor, das der Zähler bei einem einzelnen Schaltvorgang mehrere Schritte weiterzählt.

```
}
```

1.3 Empfangsinterrupt für die serielle Schnittstelle

Die serielle hat einen Sende- und einen Empfangspuffer. Bei Freigabe werden Interrupts ausgelöst, wenn

- sich ein Zeichen im Empfangspuffer befindet
- der Sendepuffer von nicht leer nach leer übergeht.

Den Test, ob der Sendepuffer voll oder der Empfangspuffer leer ist, sowie die zugehörigen Kopieroperationen aus dem Empfangs und in den Sendepuffer findet man in den blockierenden Sende- und Empfangsfunktionen in »xuartlite_l.c«. Die blockierende Sendefunktion für ein Zeichen wartet, wenn der Sendepuffer voll ist, und kopiert dann sein Zeichen in den Sendepuffer. Die blockierende Empfangsfunktion wartet, solange der Empfangspuffer leer ist und entnimmt danach ein Zeichen aus dem Empfangspuffer und gibt es zurück.

```
void XUartLite_SendByte(u32 BaseAddress, u8 Data) {
        while (XUartLite_mIsTransmitFull(BaseAddress));
        XUartLite_mWriteReg(BaseAddress, XUL_TX_FIFO_OFFSET, Data);
}

u8 XUartLite_RecvByte(u32 BaseAddress) {
        while (XUartLite_mIsReceiveEmpty(BaseAddress));
        return (u8)XUartLite_mReadReg(BaseAddress, XUL_RX_FIFO_OFFSET);
}
```

- Eine Interruptroutine sollte auf keinen Fall Warteschleifen enthalten. Der Kontrollfluss eine komplette Interupt-Routine für die UART sollte sein:
- Solange der Sendepuffer nicht voll und noch Zeichen zum versenden bereitstehen, sind diese in den Sendepuffer zu kopieren.
- Solange der Empfangspuffer nicht leer ist, sind diese in einen größeren Puffer im Programm zur späteren Abarbeitung zu kopieren.
- Abschließend wird das Interrupptbit im Interrupt-Controller gelöscht, damit wenn der Sendepuffer später nicht mehr voll oder Empfangspuffer nicht mehr leer ist, wieder ein Interrupt ausgelöst wird.

```
void uart_intr(void *baseAddress) {

void XUartLite_SendByte(u32 BaseAddress, u8 Data) {
        while (!XUartLite_mIsTransmitFull(BaseAddress) & <weiter Zeichen vorhanden>){;
         XUartLite_mWriteReg(BaseAddress, XUL_TX_FIFO_OFFSET, <Zeichen>);
    }
    while (!XUartLite_mIsReceiveEmpty(BaseAddress)){
     <Empfangspuffer> = (u8)XUartLite_mReadReg(BaseAddress, XUL_RX_FIFO_OFFSET);
    }
    XIntc_mAckIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_UART_INTERRUPT_MASK);
}
```

Im nachfolgenden Beispiel wird bei einem Interrupt, wenn der Empfangspuffern nicht leer ist, ein einzelnes Zeichen gelesen und sein Bytewert auf die Leuchtdioden ausgegeben. Wenn sich noch ein zweites Zeichen im Empfangspuffer befindet, wird nach Abarbeitung der Interruptroutine sofort wieder einer neuer Interrupt ausgelöst. Im Fall, dass der Interrupt durch das leerwerden des Sendepuffers ausgelöst wurde, wird nur Interrupptbit im Interrupt-Controller gelöscht. Das Hauptprogramm registriert wieder die Interuptroutine, schaltet die Interruptannahme an vier stellen in der Hardware ein und tut danach in einer Endlosschleife gar nichts mehr.

```
#include "xparameters.h"
#include "xgpio_l.h"
#include "xuartlite_l.h"
#include "xintc_l.h"
 char c; v

void uart_intr(void *baseAddress){
        if (!XUartLite_mIsReceiveEmpty(baseAddress)) {
          c = (u8)XUartLite_mReadReg(baseAddress, XUL_RX_FIFO_OFFSET);
          XIntc_mAckIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_UART_INTERRUPT_MASK);
        }
    XGpio_mSetDataReg(XPAR_AUSGABESCHNITTSTELLE_BASEADDR, 1, c);
}

int main() {
// Interrupt-Routine registrieren
   XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR, XPAR_XPS_INTC_0_UART_INTERRUPT_INTR,
                         (XInterruptHandler) uart_intr, (void *)XPAR_UART_BASEADDR);
// Interrupts in der UART und im Interrupt-Controller lokal und global freigeben
   XUartLite_mEnableIntr(XPAR_UART_BASEADDR);
   XIntc_mMasterEnable(XPAR_XPS_INTC_0_BASEADDR);
   XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_UART_INTERRUPT_MASK);

// Interrupts im Microblaze freigeben
   microblaze_enable_interrupts();
// Endlosschleife
   while (1);
}
```

2 Hardware-Entwurf

Abbildung 1 zeigt das zu entwerfende Rechnersystem. Es umfasst das komplette System aus dem fünften Praktikum (Prozessor, Speicher, parallelen Schnittstellen, UART, Debugschnittstelle und Zähler/Zeigebereinheit). Zusätzlich wird an den Peripheriebus ein Interrupt-Controller angeschlossen. Über diesen werden die Interruptsignale der Zähler/Zeitgebereinheit, der UART und der Eingabeschnittstelle an den Prozessor weitergeleitet.

Kopieren Sie sich aus den bisherigen Projekten folgende Entwurfsdateien

```
.../Aufg5/data/system.ucf
.../Aufg5/system.xmp
.../Aufg5/system.mhs
.../Aufg5/system.mss
```

in ein neu angelegtes Unterverzeichnis:

`.../Aufg6`

Starten Sie das Entwurfsprogramm:

`Anwendungen ▷ Umgebung ▷ Xilinx Platform Studio (EDK)`

Öffnen Sie das Projekt

`File ▷ Open Projekt ▷ <Auswahl von .../Aufg6/system.xmp>`

Alternativ können Sie auch den kompletten Entwurfsablauf vom ersten und fünften Aufgabenblatt wiederholen. Ergänzen Sie aus dem IP-Katalog wie in Abbildung 2 einen Interuptcontroller.

Schließen Sie diesen, wie in Abb. 2 an den peripheren Bus des Rechnersystems PLB an, indem Sie in dem Feld Busnahme den »PLB« auswählen.

Die Eingabeschnittstelle soll so konfiguriert werden, dass sie Interrupts erzeugt. Das erfordert einen Rechtsklick auf »Eingabeschnittstelle«, »Configure IP...« und das Setzen eines Häkchen bei »GPIO Supports Interrupts« (Abbildung 3).

Als nächstes sind dem Interrupt-Controller die Interrupt-Quellen zuzuordnen. Dazu ist im Tab »Ports« der Interrupt-Controller »auszuklappen« und bei dem Eingangssignal »Intr« die Spale »Net« anzuklicken. Der »Interrupt Connection Dialog« öffnet sich. Links sind alle verfügbaren Interrupt Quellen aufgelistet. Davon sind, wie in Abbildung 4 gezeigt, der Interrupt für den Timer, die serielle Schnittstelle und für die parallel Eingabeschnittstelle in das rechte Feld zu übernehmen.

Zum Schluss ist der Ausgang »Irq« des Interrupt-Controllers mit dem Microblaze zu verbinden. Hierzu beim Signal »Irq« auf »new connection« gehen und Namen beibehalten (oder ändern?). Denselben Namen für den Anschluss »INTERRUPT« des Mircoblaze auswählen (Abbildung 6).

Dem Interrupt-Controller ist des weiteren in der Ansicht »System Assembly View, Addresses« mit »Generate Addresses« ein Adressbereich zuzuordnen. Abbildung 6 zeigt das Ergebnis. Abschließend sind mit

```
Project ▷ Export Hardware Design to SDK ▷ Export Only
```

die Schaltung des Prozessors zu synthetisieren, die Header mit den Hardware-Adressen zu generieren etc. und alle Daten für SDK bereitzustellen.

3 Aufgaben

Aufgabe 6.1: Interrupt zur PWM-Erzeugung

1. Schreiben Sie eine Interruptroutine für Timer 0, die den Ausgabewert von LED0 einliest, invertiert und ausgibt und den Timer, wenn der Ausgabebitwert 0 ist, für eine Wartezeit auf den nächsten Interrupt von 1 s, und wenn der Ausgabewert 1 ist, für eine Wartezeit den nächsten Interrupt von 0,5 s neu initialisiert. (Erzeugung eines periodischen Blinksignals für LED0 mit einer Einschaltzeit von 1 s und einer Ausschaltzeit von 0,5 s.)

2. Schreiben Sie zuerst ein Hauptprogramm, dass die Anfangsinitialisierung des Timers durchführt, Interrupts *nicht* freigibt und die Interruptroutine quasi im Polling-Betrieb, immer wenn das Anforderungsbit im Interrupt-Controller gesetzt ist, aufruft. Testen Sie damit, dass auf LED0 ein Blinksignal mit einer Einschaltzeit von 1 s und einer Ausschaltzeit von 0,5 s ausgegeben wird.

3. Ändern Sie in einem neuen Projekt das Hauptprogramm so, dass in der Hauptschleife das Polling entfallen kann. Dazu sind die Interruptfunktion zu registrieren sowie Interrupts lokal und global freigeben. In der Endlosschleife entfällt das Polling, so dass eine Schleife der Form

   ```
   while (1);
   ```

genügt.

Hinweise:

- Testfunktion, ob das Interrupt-Anforderungsbit für Timer 0 gesetzt ist:

  ```
  XIntc_mAckIntr(BaseAddress, AckMask)
  ```

- Die Interruptnummer von Timer 0 ist die Konstante »«:

Aufgabe 6.2: UART-Empfangs-Interrupt

1. Schreiben Sie eine Interrupt-Routine für die serielle Schnittstelle, die das empfangene Zeichen als Byte auf die Leuchtdioden ausgibt.
2. Testen Sie die Interrupt-Routine in analoger Weise, wie in Aufgabe 1 b mit einem Hauptprogramm im Polling-Betrieb.
3. Ändern Sie das als Testrahmen in Aufgabenteil b verwendete Hauptprogramm wie in Aufgabe 1 c so ab, dass die Interuptroutine bei jedem eingehenden Zeichen vom Interrupt-Handler aufgerufen wird (im Interrupt-Handler registrieren, Interrupts freigeben ...).
4. Ändern Sie die Interrupt-Routine so, dass sie auf eine binäre Ziffernfolge gefolgt von einem Zeilenumbruch wartet und den empfangenen Vektor der Binärziffern anzeigt. Vorschlag für die Realisierung:

   - Vereinbarung einer globalen Variablen für den Ausgabewert.
   - Bei Empfang eines Zeilenumbruchs »\n« Ausgabe und anschließend Löschen des Variablenwerts.
   - Bei Empfang des Zeichens »0« Linksverschiebung der Variablenwerts.
   - Bei Empfang des Zeichens »1« Linksverschiebung der Variablenwerts und addition einer »1«.

   Alle anderen Zeichen sind zu ignorieren, d.h., bei ihrem Empfang soll der Variablenwert unverändert bleiben und auch sonst soll nichts passieren.

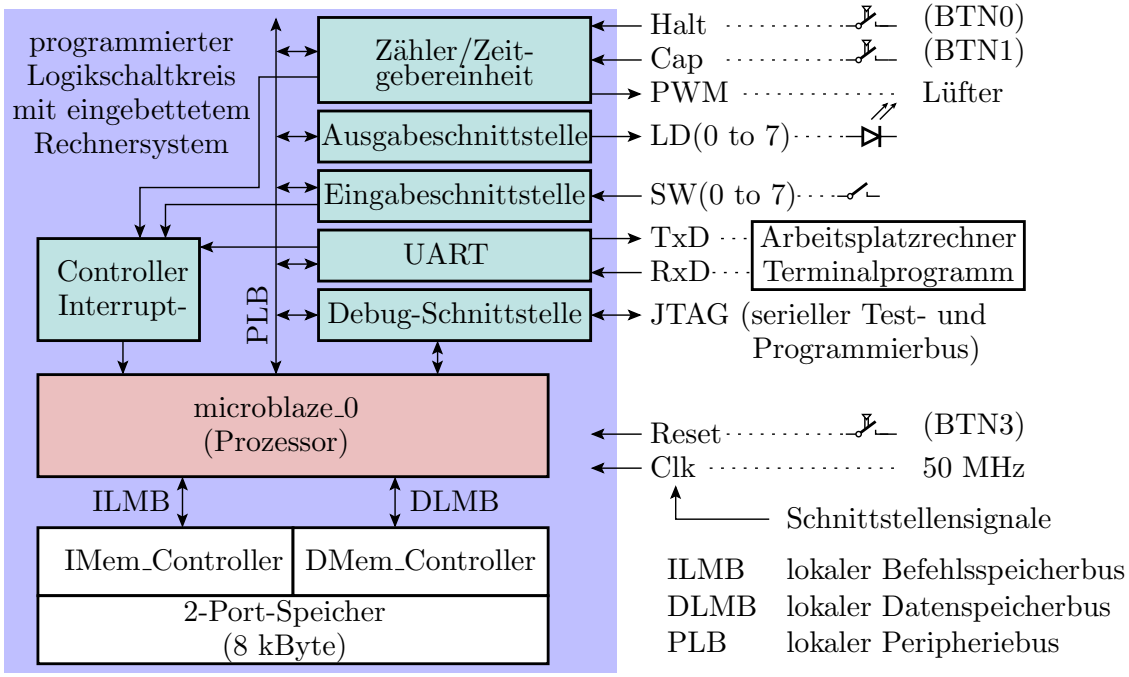

Abbildung 1: Rechnersystem mit Interrupt-Controller

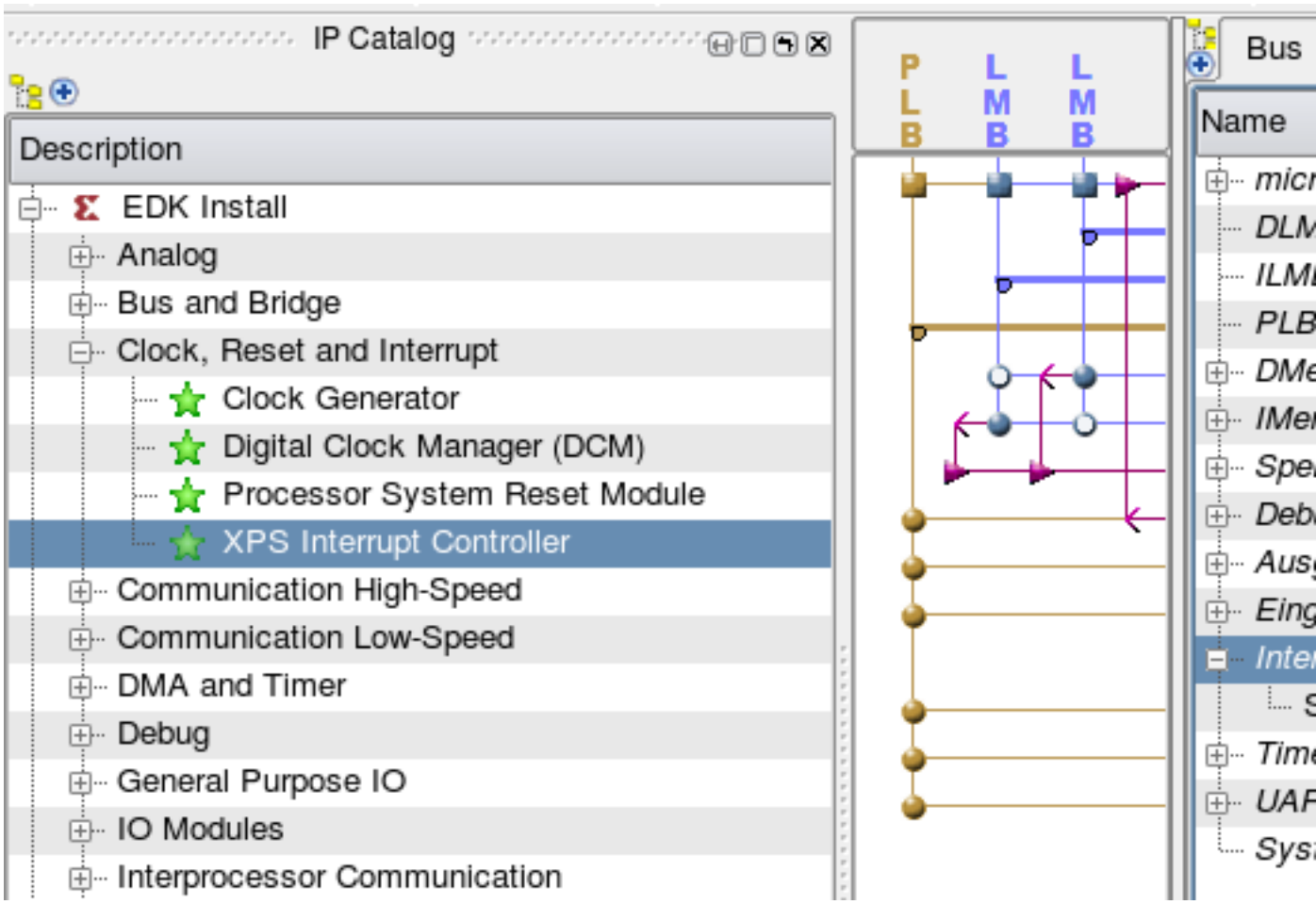

Abbildung 2: XPS Interrupt Controller hinzugefügt und an PLB angeschlossen

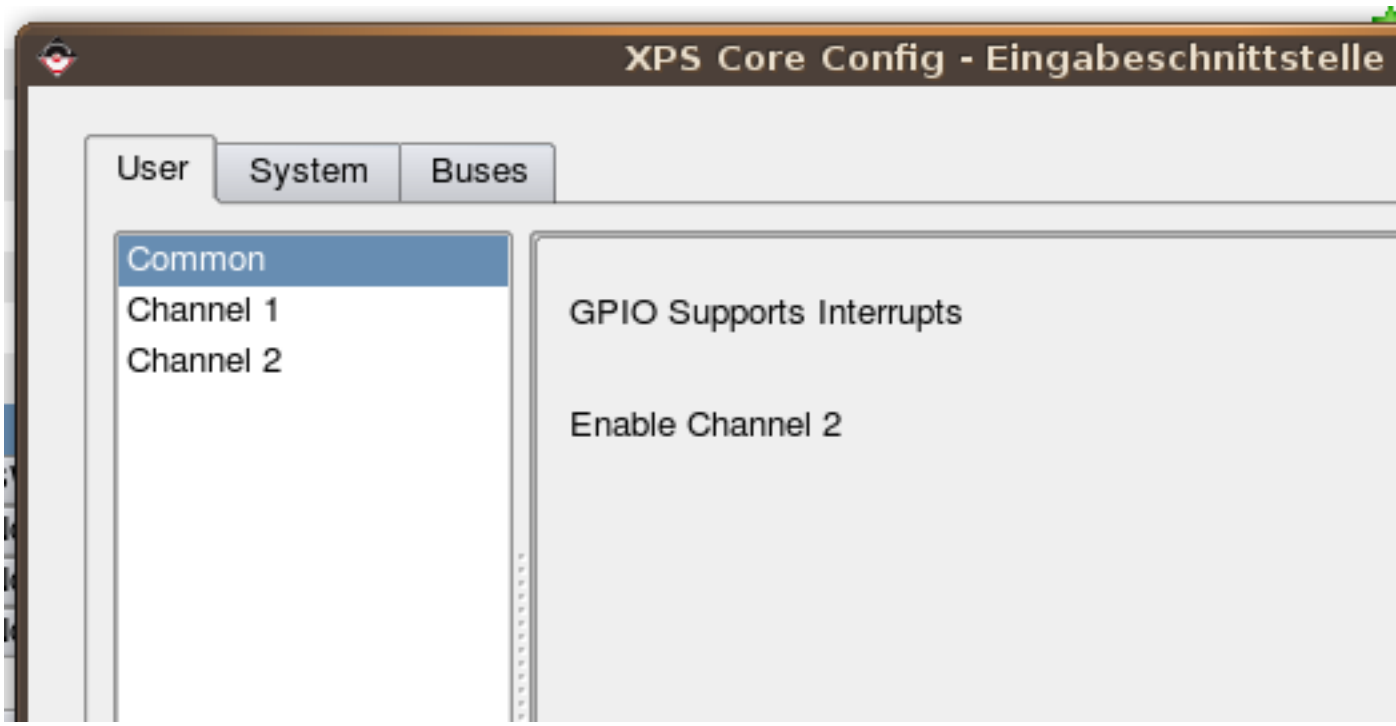

Abbildung 3: Konfiguration der Eingabeschnittstelle für die Erzeugung von Interrupts

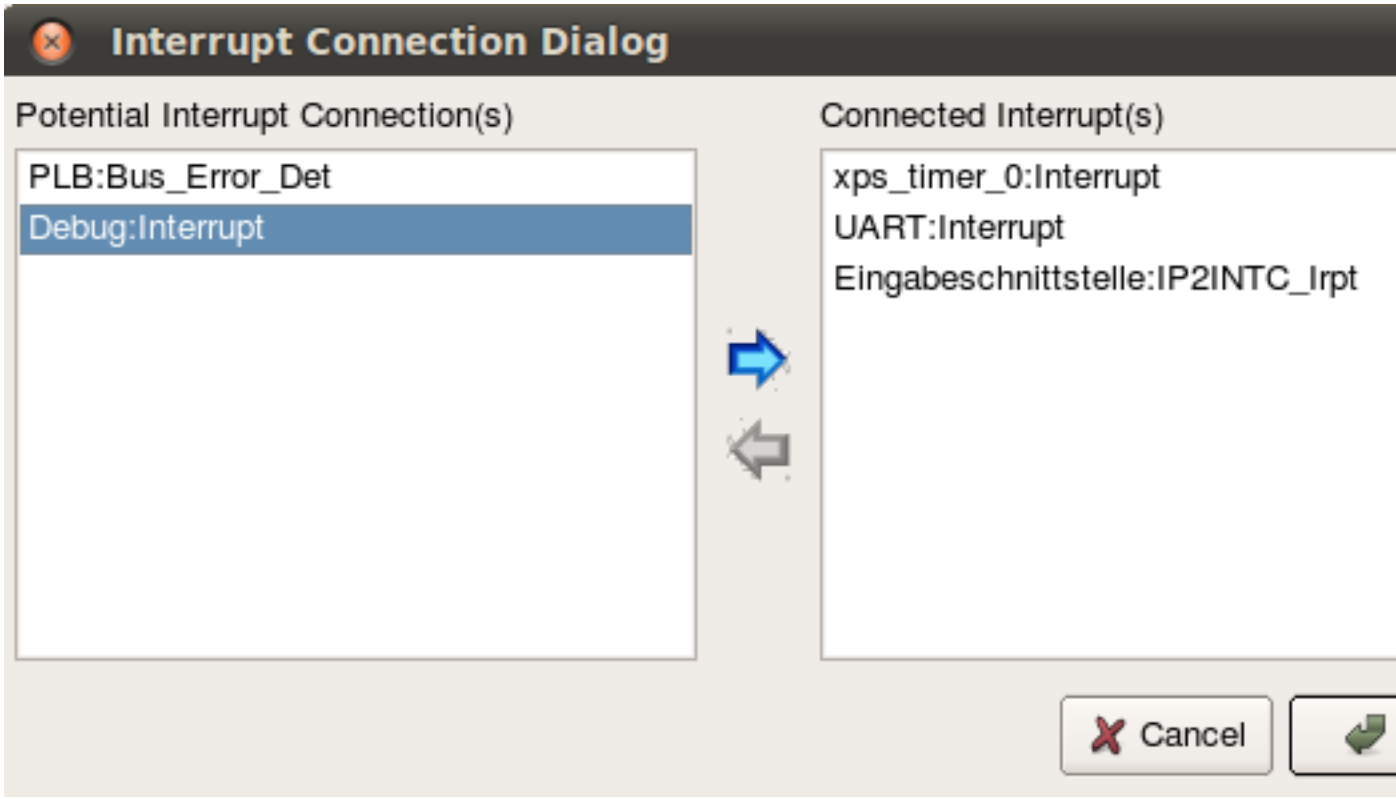

Abbildung 4: Interrupt Quellen werden dem Interrupt Controller zugeordnet

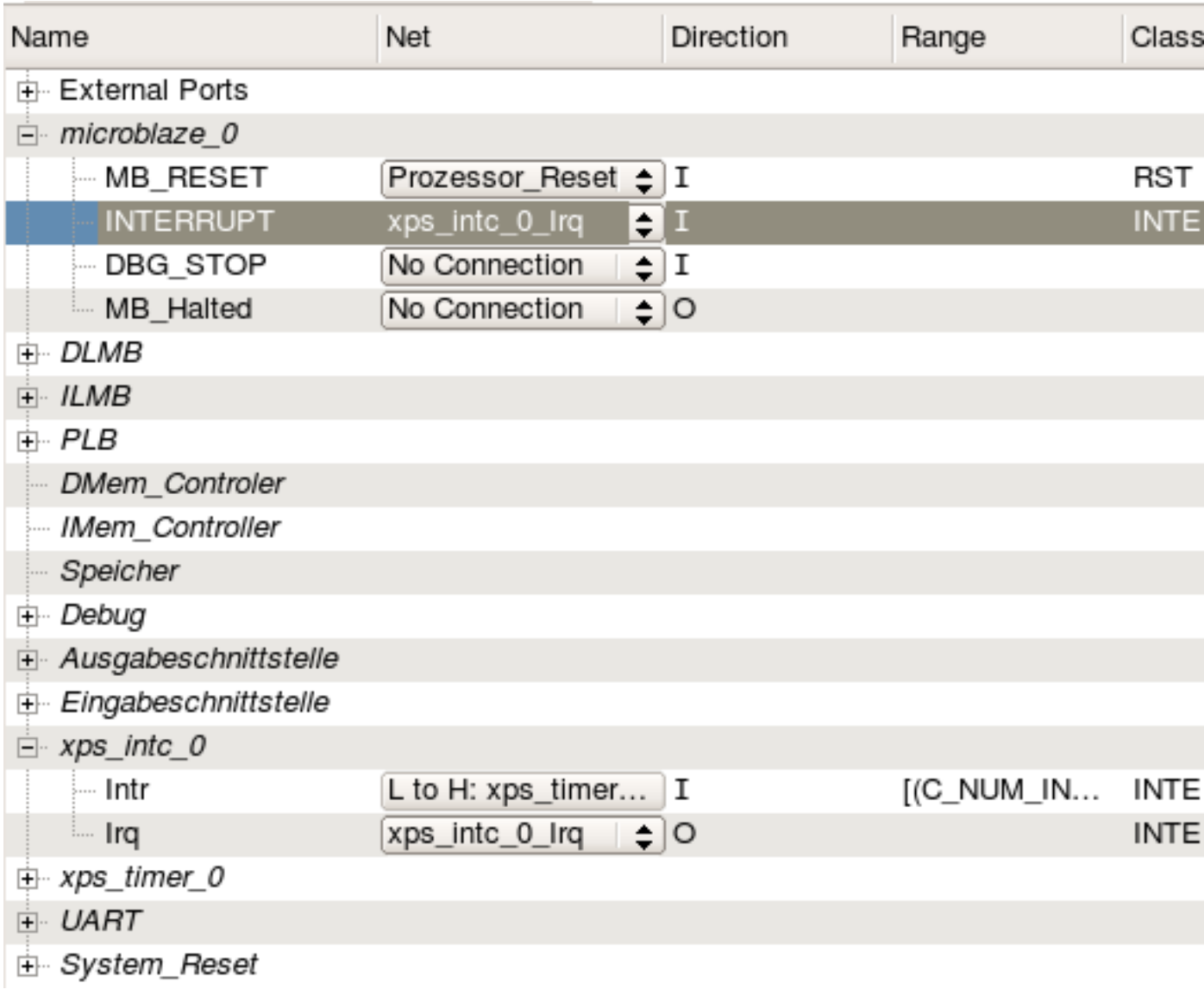

Abbildung 5: Verbindung des Interrupt-Controllers mit dem Microblaze-Prozessor

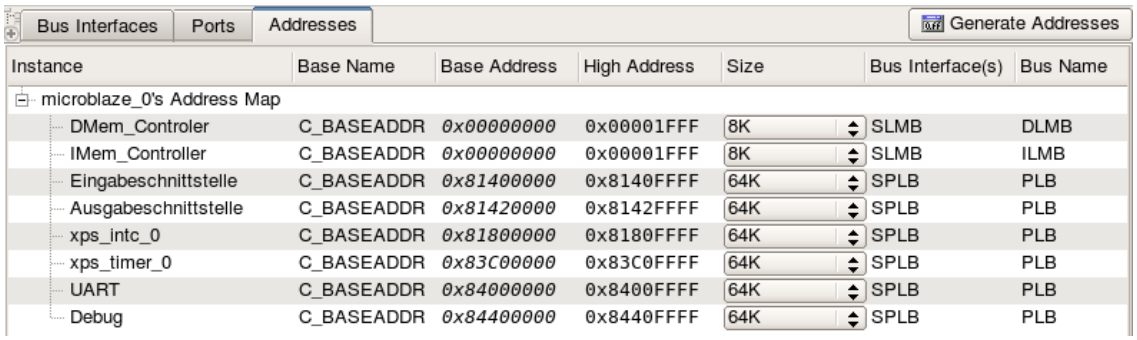

Abbildung 6: Anschlusszuordnung