miniUART Dokumentation

Version 0.9

Roman Seiger
MatrNr. 9925325
e9925325@student.tuwien.ac.at

22. Mai 2007

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit entstand im Rahmen eines Bakkalaureatspraktikums am Institut für Technische Informatik, Embedded Computing Systems Group, im Sommersemester 2005. Ziel war es, für einen an diesem Institut entwickelten Prozessor (SPEAR, siehe 1.1) einen UART (Universal Asynchronous Receiver/Transmitter) als Extension Module (siehe **??**) zur Verwendung in der Übung "Hardware/Software CoDesign" in VHDL (Very High Scale Integrated Circuit Hardware Description Language) zu entwickeln. Dieser UART sollte nur eingeschränkte Funktionalität bieten, um gegenüber den bereits vorhandenen Lösungen eine Einsparung in punkto Chipfläche zu erzielen.

Das Ergebnis ist der "miniUART", welcher in diesem Dokument beschrieben wird. Das Dokument enthalt eine detaillierte Beschreibung des miniUART (2, 3), der Registerbelegung (3.1) und der Komponenten (3.2). Abschließend wird auf die Anwendung des miniUART eingegangen (4). Im Anhang finden sich ergänzende Erklärungen zum Bootrom (A) und zu den mitgelieferten Treibern (B), sowie eine Aufstellung gebräuchlicher Baudraten (C).

1.1 SPEAR

2 miniUART Spezifikation

Aufgabe des miniUART ist es, Datenwörter mit einer Länge von 1 bis 16 Bits seriell zu senden beziehungsweise zu empfangen. Zu diesem Zweck werden die Datenwörter mit Kontrollinformationen (Startbit, Parity Bit, Stopbit(s)) versehen und als sogenannte *Frames* (Rahmen, 2.1) über die Sendeleitung (*TxD*) gesendet. Sollen Daten empfangen werden, so geschieht dies über die Empfangsleitung (*RxD*). Nach jedem komplett empfangenen *Frame* wird die Kontrollinformation ausgewertet und etwaige Fehler werden dem Benutzer gemeldet. Die Sende- beziehungsweise Empfangsgeschwindigkeit (*Baudrate*, 2.3) und das gewählte Frameformat müssen beim Sender (z.B. einem PC) und beim Empfänger (z.B.: SPEAR mit miniUART) übereinstimmen, um eine Kommunikation zu ermöglichen.

2.1 Frames

Ein Frame besteht aus einem Datenwort, Synchronisations-Bits (Start- und Stopbits) und einem optionalen Parity Bit (siehe auch 2.2). Der miniUART

akzeptiert sämtliche Kombinationen der folgenden Auflistung als gültige Frames:

- 1 Startbit (logisch 0)
- 1 bis 16 Datenbits
- kein, gerades (*Even*) oder ungerades (*Odd*) Parity Bit
- 1 oder 2 Stopbits (logisch 1)

Ist eine Leitung im Leerlauf (*Idle*), so wird diese auf *high* gesetzt (logisch 1). Abbildung 1 zeigt die möglichen Frameformate.

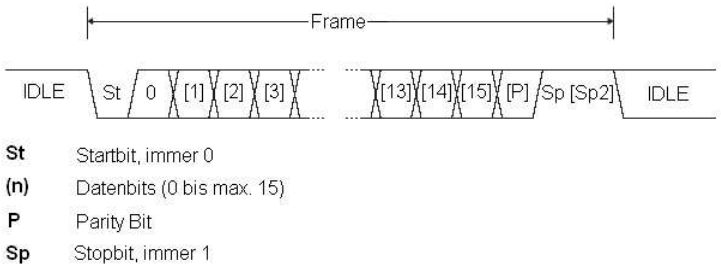

Abbildung 1: Frameformate

2.2 Parity Bit Berechnung

Um einzelne Bitfehler in einem Datenwort zu erkennen, kann ein Paritäts-Bit (Parity Bit) an die Nachricht angehängt werden (vor dem Stopbit). Dieses wird durch eine Exklusiv-Oder (XOR) Funktion über die Datenbits berechnet. Falls ungerade Parität (*Odd Parity*) gewünscht wird, so wird das Ergebnis zusätzlich invertiert. Die Funktionen im Einzelnen:

$$P_{even} = d_{n-1} \oplus ... \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0$$

$$P_{odd} = d_{n-1} \oplus ... \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

P_{even} ... gerades (*even*) Paritybit

P_{odd} ... ungerades (*odd*) Paritybit

d_i ... i-tes Datenbit

2.3 Baudrate

Die Sende- beziehungsweise Empfangsgeschwindigkeit eines UARTs wird als Baudrate bezeichnet, sie wird in Bit pro Sekunde gemessen. Beim miniUART wird die Baudrate über das UBRS (UART Baud Rate Selector) Register eingestellt. Der UBRS-Wert berechnet sich dabei wie folgt:

$$UBRS - Wert = \frac{f_{clk}}{BaudRate}$$

wobei f_{clk} die Clock-Frequenz angibt. Die oberen 12 Bits des 16 Bit UBRS-Registers beinhalten den ganzzahligen Teil des UBRS-Wertes, während die untersten 4 Bits den Bruchanteil aufnehmen.

Ein Beispiel zur Berechnung des UBRS Registerinhaltes: Bei einer Taktfrequenz von 25 MHz ($f_{clk} = 25000000Hz$) und einer gewünschten Baudrate von 38.4 kbit/s ($BaudRate = 38400bit/s$) ergibt sich aus

$$UBRS - Wert = \frac{25000000}{38400} = 651.04167 = \text{0x28B,6}$$

Eine Auflistung einiger UBRS-Werte gebräuchlicher Baudraten findet sich im Anhang (C).

3 miniUART Aufbau

Der miniUART ist als Extension Module für SPEAR realisiert. Als solches kommuniziert er mit dem Prozessorkern über acht Register (siehe 3.1), diverse Kontroll- und eine Interruptleitung, deren Ansteuerung im Hauptmodul des miniUART erfolgt. Die Koordinierung der internen Abläufe obliegt dem **Control Module**. Dieses steuert sowohl den Empfänger (**Receiver**) als auch den **Baud Rate Generator (BRG)**, welcher die Sende- und Empfangszeitpunkte jedes einzelnen Bits bestimmt. Der Sender (**Transmitter**) wird vom Control Module indirekt über den Baud Rate Generator in Betrieb gesetzt. Sowohl Transmitter als auch Receiver sind über den **Busdriver** mit der Sende- beziehungsweise Empfangsleitung (*TxD bzw. RxD*) verbunden.

3.1 Registerbelegung

In Abbildung 2 ist die Registerbelegung des miniUART dargestellt. Nachfolgend werden die Register im Einzelnen erklärt.

miniUART Register – 32 Bit Version

CONFIG_C (–125)	CONFIG (–126)	STATUS_C (–127)	STATUS (–128) ← Baseaddress
MSG_H (–121)	MSG_L (–122)	UART_CMD (–123)	UART_CFG (–124)
Unused	Unused	UBRS_H(–119)	UBRS_L(–120)
Unused	Unused	Unused	Unused
Unused	Unused	Unused	Unused
Unused	Unused	Unused	Unused

Abbildung 2: Registerbelegung des miniUART

3.1.1 Status Register

Der Zustand des miniUART wird im Statusregister dargestellt (Abbildung 3). Dieses Register ist nur lesbar, Schreibzugriffe werden ignoriert. Details zum allgemeinen Teil des Status- beziehungsweise des Config-Registers sind in [Hub02] zu finden.

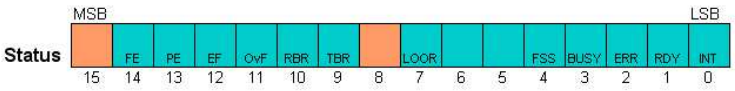

Abbildung 3: Statusregister

Modulspezifischer Teil des StatusRegisters:

Frame Error (FE): wird bei einem falschen/fehlenden Stopbit gesetzt (*)

Parity Error (PE): wird bei einem falschen Paritybit gesetzt (*)

Event Flag (EF): 1, falls der im Event Selector eingestellter Event eingetreten ist. Wird durch das Lesen des Statusregisters zurückgesetzt.

Overflow Flag (OvF): wird gesetzt, wenn eine Nachricht empfangen wird und das RBR-Bit noch gesetzt ist (vorige Nachricht wurde noch nicht gelesen!)

Receive Buffer Ready (RBR): wird gesetzt, wenn eine Nachricht empfangen wurde; um das RBR-Bit zu löschen, muß die Nachricht gelesen werden.

*nur bei gesetztem Transmission Control Bit oder Error Interrupt Bit!

Transmit Buffer Ready (TBR): bei TBR=1 ist der Transmitter bereit, eine neue Nachricht zu versenden. Das Bit wird beim Starten einer Übertragung gelöscht, allerdings schon vor Abschluß der Übertragung wieder gesetzt (sobald der Transmitter die Daten aus dem Register übernommen hat!)

Allgemeiner Teil:

Loop Read (LOOR): zur Feststellung der Anwesenheit des Moduls

Fail Safe State (FSS): wird gesetzt, wenn sich der miniUART im Fail-Safe Zustand befindet

BUSY: nicht gesetzt

Error (ERR): wird gleichzeitig mit Frame- oder Parity-Error gesetzt

Ready (RDY): gesetzt

Interrupt (INT): ein Interrupt wurde ausgelöst (Event-, Error- oder FSS-Interrupt)

3.1.2 Config Register

Das Configregister ist für allgemeine Einstellungen am Modul vorgesehen (Abbildung 4).

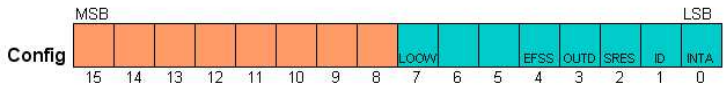

Abbildung 4: Configregister

Allgemeiner Teil:

Loop Write (LOOW): dieses Bit wird direkt an das LOOR-Bit durchgeschleift (siehe dort)

Enter Fail Safe State (EFSS): versetzt den miniUART in den Fail-Safe-State (keine Ausgabe auf der Sendeleitung, Interrupt wird ausgelöst)

Output Disable (OutD): keine Ausgabe über die Sendeleitung, wenn gesetzt

Soft Reset (SRes): resetiert den miniUART

Identification (ID): ist dieses Bit gesetzt, so kann im Register DATA0 (UARTConfig) der Modultyp und im Register DATA1 (UARTCommand) die Versionsnummer ausgelesen werden

Interrupt Acknowledge (INTA): mit diesem Bit wird ein Interrupt bestätigt

3.1.3 UARTCfg Register

In diesem Register werden die Kommunikationsparameter (siehe 2.1) eingestellt und die Fehlererkennung ein- und ausgeschaltet (Abbildung 5).

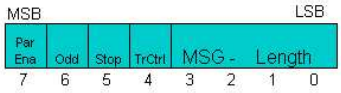

Abbildung 5: UARTConfig Register

Parity Enable (ParEna): beim Senden wird ein Paritybit an jede Nachricht angehängt, beim Empfangen wird ein Paritybit erwartet und verglichen

Odd/Even Parity (Odd): wird Odd gesetzt, so wird Odd-Parity (Ungerade Parität), ist Odd=0, so wird Even-Parity (Gerade Parität) verwendet

Stop: bestimmt die Anzahl der Stopbits (Stop=0 $\Rightarrow$ 1 Stopbit; Stop=1 $\Rightarrow$ 2 Stopbits)

Transmission Control (TrCtrl): wenn TrCtrl gesetzt wird, so werden Parity- und Frame-Errors durch das jeweilige Bit angezeigt

Message Length (MsgLength): Nachrichtenlänge - 1 ("0000" entspricht 1 Bit Nachrichtenlänge; "0111" entspricht 8 Bit Nachrichtenlänge; usw.); max. Länge: "1111" (16 Bit)

3.1.4 UARTCmd Register

Mit diesem Register wird der miniUART in Aktion versetzt. Darüberhinaus kann in diesem Register das Interruptverhalten des miniUART eingestellt werden (Abbildung 6).

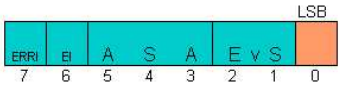

Abbildung 6: UARTCommand Register

Error Interrupt (ERRI): entspricht dem TrCtrl-Bit, allerdings wird beim Auftreten eines Fehlers zusätzlich ein Interrupt ausgelöst

Event Interrupt (EI): löst einen Interrupt aus, wenn das EventFlag gesetzt wird

Assigned Action (AsA): mit diesen drei Bits wird der miniUART in Aktion versetzt. Die zulässigen Bitkombinationen sind in Tabelle 1 ersichtlich

AsA	Durchgeführte Aktion
011	Transmitter starten: Die Nachricht im Messageregister wird übertragen.
100	Enable Receive Mode: Versetzt den miniUART in den Empfangsmodus. Sobald eine Nachricht empfangen wurde, wird diese in das Messageregister geschrieben und RBR gesetzt. Der miniUART bleibt im Empfangsmodus bis er mit “101” (Disable Receive Mode) ausgeschaltet wird.
101	Disable Receive Mode: Schaltet den Empfänger ab.

Tabelle 1: Assigned Actions

Bei allen anderen Bitkombinationen wird keine Aktion ausgeführt, am Zustand des miniUART ändert sich nichts.

Event Selector (EvS): bestimmt das Ereignis, auf das der miniUART reagieren soll. Wenn das spezifizierte Ereignis eintritt, wird das EventFlag gesetzt, bei gesetztem EI-Bit zusätzlich ein Interrupt ausgelöst und die in AsA spezifizierte Aktion ausgeführt. Alle einstellbaren Ereignisse sind in Tabelle 2 zusammengefasst.

3.1.5 Message Register (MSG_L +MSG_H

In diesem Register werden empfangene bzw. zu sendende Nachrichten abgelegt (Abbildung 7). Das Lesen einer empfangenen Nachricht (wenn RBR

EvS	Ereignis
00	No Event: Keine Reaktion auf Ereignisse
01	Startbitdetection: Startbit empfangen
10	Receive Completion: Ganze Nachricht empfangen
11	Transmission started: Zu sendende Nachricht wurde übernommen, Sender gestartet, bereit für die nächste Nachricht

Tabelle 2: Events

gesetzt ist) bewirkt das Zurücksetzten folgender Flags: RBR, FE, PE, ERR, OvF. Aus diesem Grund sollte vor dem Lesen einer empfangenen Nachricht das Statusregister ausgelesen und überprüft werden.

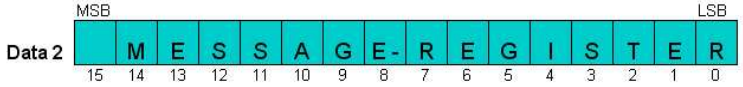

Abbildung 7: Message Register

Hinweis: Solange eine empfangene Nachricht nicht gelesen wird, bleibt RBR gesetzt. Das führt beim Empfang einer weiteren Nachricht zu einem Überlauf (Overflow, OvF wird gesetzt)!

3.1.6 UBRS (UART Baud Rate Selection) Register (UBRS_L+UBRS_H)

In diesem Register wird die Baudrate eingestellt (Abbildung 8). Die Berechnung des UBRS-Wertes wird in 2.3 erklärt.

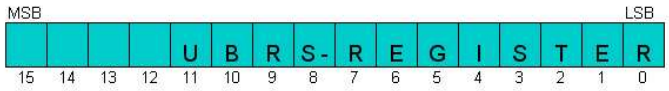

Abbildung 8: UART Baud Rate Selection Register

Eine Auflistung einiger UBRS-Werte gebräuchlicher Baudraten findet sich im Anhang (C).

3.2 Komponenten

Der miniUART ist zwecks Überschaubarkeit in sechs Module aufgeteilt. Abbildung 9 zeigt das Blockschaltbild des miniUART, danach werden die Module im einzelnen besprochen.

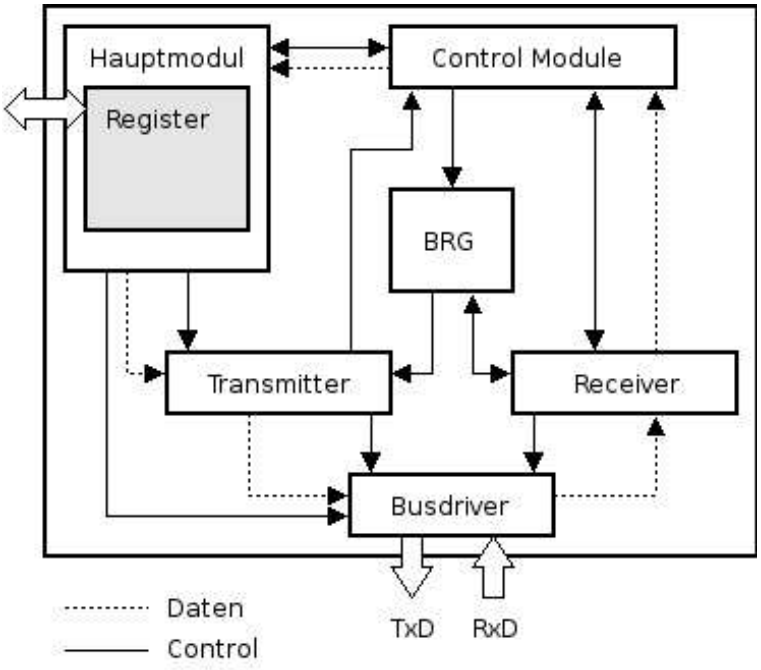

Abbildung 9: miniUART Blockschaltbild

3.2.1 Hauptmodul

Das Hauptmodul stellt die Anbindung des miniUART an den Prozessorkern dar und beinhaltet das Registerfile des miniUART. Hier werden die Statusflags je nach Rückmeldung der anderen Komponenten gesetzt. Außerdem ist das Hauptmodul für das Auslösen von Interrupts und die Paritätsberechnung der zu sendenden Nachricht verantwortlich.

3.2.2 Control Module

Das Control Module steuert sämtliche Abläufe innerhalb des miniUART. Es wertet die Assigned Action (AsA) Einstellung aus und setzt den miniUART dementsprechend in Betrieb. Weiters werden die Rückmeldungen der Komponenten geprüft, aufbereitet und an das Hauptmodul weitergeleitet. Unter Zuhilfenahme der Event Selector (EvS) Bits wird beispielsweise ein eventuell aufgetretenes Ereignis signalisiert. In diesem Modul wird darüberhinaus gegebenenfalls das empfangene Paritätsbit überprüft.

3.2.3 Receiver

Der Receiver wartet, einmal in Betrieb gesetzt, auf ein Startbit auf der Empfangsleitung (logisch 0). Ist dieses empfangen worden, signalisiert er dies an das Control Module und startet den Baud Rate Generator. Bei jedem von diesem generierten "Receive Pulse" wird der augenblickliche Zustand der Empfangsleitung als Datenbit beziehungsweise Kontrollbit (Start-, Stop- oder Paritätsbit) gespeichert. Nach vollständigem Empfang der Nachricht wird diese gemeinsam mit einem möglichen Frame Error an das Control Module übergeben, der Baud Rate Generator wird deaktiviert und der Receiver ist bereit, das nächste Startbit zu empfangen.

3.2.4 Transmitter

Der Transmitter übernimmt mit dem ersten erhaltenen "Transmit Pulse" die zu sendende Nachricht und die Kommunikationsparameter aus dem Messageregister und beginnt die Übertragung mit dem Startbit. Danach wird mit jedem "Transmit Pulse" ein Bit der Nachricht und am Schluß, je nach Einstellung, das Paritybit und ein oder zwei Stopbits gesendet. Die Beendigung der Übertragung wird dann dem Control Module gemeldet, welches den Baud Rate Generator und damit auch den Transmitter deaktiviert.

3.2.5 Baud Rate Generator

Der Baud Rate Generator (BRG) bestimmt den Zeitpunkt, zu dem ein Bit einer Nachricht gesendet beziehungsweise empfangen wird. Zu diesem Zweck wird beim Senden ein sogenannter "Transmit Pulse" und beim Empfangen ein "Receive Pulse" pro Bit generiert. Die Zeit zwischen zwei Pulses wird vom UBRS Register bestimmt. Dieses wird durch zwei geteilt, in einen Zähler geladen und mit jedem Taktsignal heruntergezählt. Ist der Zähler bei null angelangt, wird abwechselnd entweder ein Transmit oder ein Receive Pulse generiert. Danach wird der Zähler erneut beschickt, wobei der Bruchanteil des UBRS-Registers berücksichtigt wird. Diese Prozedur wird bis zur Deaktivierung des Baud Rate Generators wiederholt.

Diese abwechselnde Aktivierung des Transmit und Receive Pulses ermöglicht es, die Empfangsleitung nicht am Anfang oder am Ende eines ankommenden Bits zu übernehmen, was aufgrund von leichten Schwankungen in der Übertragungsgeschwindigkeit zu Fehlern führen könnte, sondern in der Mitte (Abbildung 10).

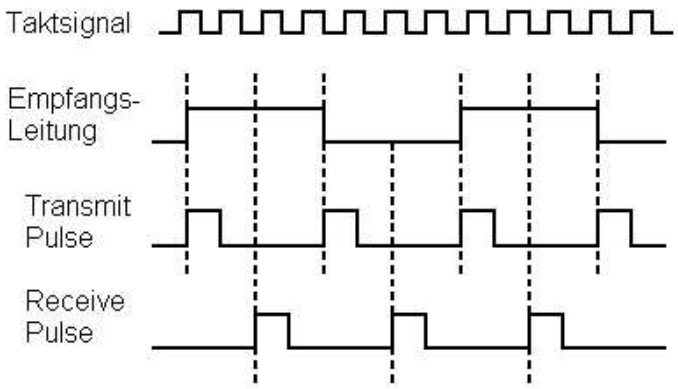

Abbildung 10: Transmit und Receive Pulses

3.2.6 Busdriver

Alle Sende- und Empfangsoperationen des miniUART werden über den Busdriver abgewickelt. Der Busdriver versetzt die Sendeleitung, wenn nicht gerade gesendet wird, in einen hochohmigen Zustand ("Z"). Dies erfolgt auch beim Unterbinden der Ausgabe (*Output Disable* oder *Fail Safe State*, siehe 3.1.2). Außerdem verfügt der Busdriver über einen Filter, der den Zustand der Empfangsleitung nur an den Receiver weitergibt, wenn dieser für mindestens drei Clockzyklen konstant bleibt. Damit werden kurze Störungen ("Spikes") abgefangen.

4 Anwendung

Der miniUART kann generell von jedem Anwenderprogramm als Kommunikationsschnittstelle zum PC oder anderen Peripheriegeräten verwendet werden.

Die nachfolgenden Erklärungen beziehen sich auf die Verwendung des miniUART zusammen mit SPEAR und einigen anderen, schon vorhandenen Extension Modules. Es sollen hier nur Anwendungsbeispiele gegeben werden, ein Anspruch auf Vollständigkeit wird nicht erhoben.

Hinweis: Beispiele in C, defines aus init.c (B) übernommen

4.1 Programmierung

Der miniUART kann zur Programmierung des SPEAR eingesetzt werden. Dabei wird im Bootrom eine Routine ausgeführt, die ein compiliertes Pro-

gramm empfängt, in den Befehlsspeicher überträgt und dieses danach ausführt. Folgendes ist dabei zu beachten: (Die Erklärungen beziehen sich auf das Standard-Bootrom des miniUART (siehe A). Alternativ kann auch das Bootrom umgeschrieben und neu übersetzt werden, dies wird jedoch nicht empfohlen.)

- Die Extension Modules müssen folgendermaßen konfiguriert sein (Tabelle 3):

Module	BASE-Adresse
SYSCTRL	-32
ProgModule	-64
DIS7SEG	-96
miniUART	-128

Module	Interrupt Adresse
miniUART	11

Tabelle 3: BASE- und Interrupt-Adressen der Extension Modules

- Folgende Verbindungseinstellungen müssen im Terminalprogramm (z.B.: HyperTerminal) vorgenommen werden (Tabelle 4):

Baudrate	57600 bit/s
Nachrichtenlänge	8 bit
Parität	Gerade (Even)
Stopbits	2
Flussteuerung	Keine

Tabelle 4: Verbindungseinstellungen

Ablauf des Programmiervorgangs:

Nach einem Reset des SPEAR zeigt der miniUART das Statusregister auf der Siebensegmentanzeige dar (normalerweise “0202”: TBR=1, RDY=1). Im Terminalprogramm wird nun das übersetzte Programm (als Textdatei (!)) gesendet. Der miniUART stellt dabei jedes empfangene Byte auf der Siebensegmentanzeige dar (1. und 2. Digit, 3. und 4. sind immer “0”, außer es tritt ein Fehler auf!). Nach Abschluß des Downloads wird das Anwenderprogramm sofort gestartet. Sollte bei der Übertragung wider Erwarten ein Fehler auftreten, so wird das Statusregister angezeigt (Verbindungseinstellungen kontrollieren!).

4.2 Initialisierung

Um den miniUART zu verwenden, müssen als erstes die Kommunikationsparameter (Parität, Stopbit, Nachrichtenlänge und Baudrate) eingestellt werden.

```
/* Even Parity, 2 Stopbits, TrCtrl=1, MsgLength=8 */
UART_CONFIG = 0xB700;
/* Baud Rate 57600 bit/s */
UART_UBRS = 0x1B20;
```

4.3 Senden

Um eine Nachricht zu senden ist sicherzustellen, dass der miniUART bereit dazu ist (TBR=1). Dann wird die zu sendende Nachricht in das Messageregister geladen und danach der Transmitter gestartet (AsA = “011”).

```
/* auf TBR=1 warten */
while ((UART_STATUS & 0x0200) != 0x0200)
  {
    asm("nop");
  }

/* Zu sendende Nachricht in Messageregister laden */
UART_MSG = data;

/* Starte Transmitter (ERRI=0, EI=0, AsA="011", EvS="00")*/
UART_CMD = 0x0018;
```

4.4 Empfangen

Um eine Nachricht zu empfangen muß der miniUART in den Empfangsmodus (Receive Mode) versetzt werden (AsA = “100”). Zu Beachten ist, dass der Receive Mode aktiviert bleibt, bis der Empfänger explizit ausgeschaltet wird (AsA = “101”).

```
/* Starte Receiver (ERRI=0, EI=0, AsA="100", EvS="00") */
UART_CMD = 0x0020;

/* auf RBR=1 warten */
while ((UART_STATUS & 0x0400) != 0x0400)
  {
```

```
    asm("nop");
  }

/* Status lesen */
status = UART_STATUS;

/* Auf Overflow überprüfen */
if ((status & 0x0800) == 0x0800)
  {
    /* Fehlerbehandlung */
  }

/* Auf Parity Error überprüfen */
if ((status & 0x2000) == 0x2000)
  {
    /* Fehlerbehandlung */
  }

/* Auf Frame Error überprüfen */
if ((status & 0x4000) == 0x4000)
  {
    /* Fehlerbehandlung */
  }

/* empfangene Nachricht lesen */
data = UART_MSG;

/* Receiver ausschalten */
UART_CMD = 0x0028;
```

4.5 Einschränkungen

Bei der Verwendung des miniUARTs sind folgende Einschränkungen zu beachten:

Half Duplex Der miniUART ist nur für den Half-Duplex-Betrieb ausgelegt. Gleichzeitiges Senden und Empfangen ist zwar möglich, allerdings kann es aufgrund des gemeinsamen Messageregisters sehr leicht zu Überläufen kommen.

Kompatibilität Um eine Weiterverwendung älterer Programme möglichst zu gewährleisten, wurde die Registerbelegung des miniUART an die

seines “Vorgängermodells” (EUART, siehe [Raf05]) angeglichen. Aufgrund der umfangreicheren Funktionalität des EUART ist die Weiterverwendung allerdings nur begrenzt möglich. Beispielsweise unterstützt der miniUART im Vergleich zum EUART keine automatische Synchronisation. Umgekehrt steht einer Verwendung von Programmen, die für den miniUART entwickelt wurden, mit dem EUART nichts im Wege, allerdings kann dafür keine Garantie übernommen werden.

Baudraten Generell kann der miniUART mit jeder beliebigen Baudrate betrieben werden (Berechnung des UBRS-Wertes siehe 2.3). Allerdings kann es bei sehr hohen Baudraten zu einem häufigeren Auftreten von Übertragungsfehlern kommen. Die Fehlerrate kann auch aufgrund von Taktschwankungen variieren. Aus diesem Grund wird empfohlen, den miniUART mit der vorgesehenen Frequenz von 25 MHz zu betreiben. Einige UBRS-Werte für gebräuchliche Baudraten bei dieser Frequenz sind in C zu finden.

A Bootrom

Die Routine im Bootrom des SPEAR hat die Aufgabe, ein Programm zu empfangen, dieses in den Befehlsspeicher zu laden und danach zu starten. Zu diesem Zweck wird der miniUART in den Empfangsmodus versetzt und eine Interruptroutine installiert, die jedes empfangene Byte über das Programmer Module (Details siehe [Del02]) in den Befehlsspeicher transferiert und schlussendlich den SPEAR resetiert, um das Programm zu starten. Zur Überwachung dieses Vorgangs wird die Siebensegmentanzeige über das Extension Module "Dis7Seg" angesteuert. Die dazu notwendigen Einstellungen sind in 4.1 angeführt.

```
begin:      LDL r0, lo(rec_int)
            LDH r0, hi(rec_int)
            STVEC r0, -6    ; miniUART Interrupt= 10
            LDL r26, -40    ; miniUART_BASE= FFD8
            LDL r27, -8     ; SYSCTRL_BASE= FFF8
            LDL r28, -32    ; DIS7SEG_BASE= FFE0

            LDL r1, 0xFF    ; Prescaler fuer Display setzen
            LDH r1, 0x00
            STOFZ r1, 4

            LDL r19, 0x00   ;
            LDH r19, 0b10100111; EvenParity,2Stop,MsgLength=8
            STOFX r19, 2    ;  UART Configuration
            LDL r2, 0b00100000; load UBRS
            LDH r2, 0b00011011;  (57600 baud)
            STOFX r2, 7     ;
            STOFZ r2, 5     ; UBRS ausgeben
            ; ======= Init Constants =========
            LDL r10, 0      ; Initialize ByteCounter
            LDL r1, 0       ;
            LDH r1, 0x80    ; set GIE
            STOFY r1, 1     ; in SaveStatusReg
            LDL r31, lo(rcvmode)
            LDH r31, hi(rcvmode)
            RETI            ; SaveStatusReg => StatusReg

            ;       Die ersten 2 Bytes abfangen!!!
            ; ====== Set miniUART in Receive Mode ============
```

```
rcvmode:   LDL r2, 0b10100000 ; ERRI=1 & Enable Receive
           STOFX r2, 3     ; Store in Data1
wait1:     LDOFX r11, 0    ; StatusReg lesen
           STOFZ r11, 5    ; Status ausgeben
           BTEST r11, 10   ; RBR=1? Empfang fertig?
           JMPI_CF wait1   ; No, weiterwarten
           LDL r19, 0
           LDH r19, 0
           STOFZ r19, 5    ; clear Display
           LDOFX r19, 4    ; read Msg, clear RBR
           STOFZ r19, 5    ; Msg ausgeben

           ; ====== Set miniUART in Receive Mode ============
           LDL r2, 0b10100000 ; ERRI=1 & Enable Receive
           STOFX r2, 3     ; Store in Data1
wait2:     LDOFX r11, 0    ; StatusReg lesen
           STOFZ r11, 5    ; Status ausgeben
           BTEST r11, 10   ; RBR=1? Empfang fertig?
           JMPI_CF wait2   ; No, weiterwarten
           LDL r19, 0
           LDH r19, 0
           STOFZ r19, 5    ; clear Display
           LDOFX r19, 4    ; read Msg, clear RBR
           STOFZ r19, 5    ; Msg ausgeben

           ; ======= set Event, On RecCompletion EnaRec ====
           LDL r19, 0b11100100;
           LDH r19, 0x00   ;
           STOFX r19, 3    ; Store in DATA1
           ; ==========================================
loop:      JMPI loop


; ========================================================
; ============= RECEIVE INTERRUPT ========================
; ========================================================
rec_int:   LDOFX r3, 0     ; StatusReg laden
           BTEST r3, 2     ; ERR=1?
           JMPI_CT rec_error;
           LDOFX r17, 4    ; Read Msg to avoid OverFlow
```

```
cont:       STOFZ r17, 5    ; Msg ausgeben
            LDOFX r2, 1     ; Config auslesen
            BSET r2, 0      ; Int Ack setzen
            STOFX r2, 1     ; Config zurueckschreiben
            ; ==== SYSCTRL ========
            LDOFY r3, 2     ; Read Interrupt Protocol Register
            BCLR r3, 10     ; Reset miniUARTs Interrupt Flag
            STOFY r3, 2     ; WriteBack Int Protocol Reg
            LDOFY r2, 1     ; Config auslesen
            BSET r2, 0      ; Int Ack setzen
            STOFY r2, 1     ; Config zurueckschreiben
            ADDI r10, 1     ; Increment ByteCounter

byte1:      CMPI_EQ r10, 1  ; ByteCounter=1? (Address HI-Byte)
            LDL r8, 0xFE    ;
            ;STOFZ r8, 6    ; light LED1 green
            JMPI_CT load_HI_Byte;

byte2:      CMPI_EQ r10, 2  ; ByteCounter=2? (Address LO-Byte)
            LDL r8, 0xFC    ;
            ;STOFZ r8, 6    ; light LED1 red & green
            JMPI_CT load_LO_Byte;

byte3:      CMPI_EQ r10, 3  ; ByteCounter=3? (TypeRec Byte)
            LDL r8, 0x0C    ; --LED4--  --LED3-- LED2 --LED1--
            ;STOFZ r8, 6    ; red&green,red&green,XX,red&green
            JMPI_CF byte4   ;
            LDOFX r4, 4     ; Load Message Register
            CMPI_EQ r4, 1   ; TypeByte=01 -> end_prog
            JMPI_CT end_prog;
            RETI            ;

byte4:      CMPI_EQ r10, 4  ; ByteCounter=4? (Instr HI-Byte)
            LDL r8, 0xF8    ; LED4 LED3 LED2 -LED1----
            ;STOFZ r8, 6    ; XX,XX,Xgreen,red & green
            JMPI_CT load_HI_Byte;

byte5:      CMPI_EQ r10, 5  ; ByteCounter=5? (Instr LO-Byte)
            LDL r8, 0xF0    ; LED4 LED3 LED2 -LED1----
            ;STOFZ r8, 6    ; XX,XX,red & green,red & green
            JMPI_CT load_LO_Byte;
```

```
byte6:      LDL r8, 0       ;
            ;STOFZ r8, 6    ; light ALL green+red LEDs
            LDL r4, 0x00    ;
            LDH r4, 0x80    ;
            LDL r26, -16    ; ProgModule_BASE
            STOFX r4, 1     ; Set ProgExe Flag
            NOP             ;
            LDL r10, 0      ; Reset ByteCounter
            LDL r26, -40    ; Restore miniUART_BASE
            RETI            ;

load_LO_Byte: LDOFX r4, 4     ; load Message Register
            OR r4, r9       ; Paste HI-byte in r4
            ;STOFZ r4, 6    ; show MessageWord on 7SegDisplay
            CMPI_EQ r10, 5  ; ByteCounter = 5?
            LDL r26, -16    ; Yes-> store in ProgMod.Data-Reg
            JMPI_CT data    ; No -> store in ProgMod.Addr-Reg
            STOFX r4, 2     ; store MsgWord in Address-Reg
            LDL r26, -40    ; Restore miniUART_BASE
            RETI

data:       STOFX r4, 3     ; store MsgWord in DATA-Reg
            LDL r26, -40    ; Restore miniUART_BASE
            RETI

load_HI_Byte: LDL r26, -40    ; miniUART_BASE
            LDOFX r9, 4     ; load Message Register
            LDL r15, 8      ; Init ShiftCounter
shift_l:    CMPI_EQ r15, 1  ;
            SL r9           ;
            ADDI r15, -1    ;
            JMPI_CF shift_l ;
            ;STOFZ r9, 6    ; show MsgWord on 7SegDisplay
            RETI            ;

end_prog:   LDL r26, -16    ; ProgModule_BASE
            LDL r4, 0       ;
            LDH r4, 0b00000011; Set ResetExe=InstrSrc=1
            STOFX r4, 1     ; switch to Instruction Memory
            NOP             ;
```

```
            NOP             ;
            NOP             ;
            LDL r26, -40    ; Restore miniUART_BASE
            RETI            ;

rec_error:  LDOFX r12, 0    ;
            STOFZ r12, 5    ; show miniUART Status on Display
            JMPI rec_error  ;
```

B Treiber

init.c

Die Datei *init.c* stellt einige nützliche defines für die Verwendung des miniUART zur Verfügung. Dies sind im speziellen die BASE-Adresse und die verwendeten Register des miniUART.

```
/* miniUART-Modul ****************************************/
#define UART_BADDR (0xFF00 | (MINIUART_BASE << 3))

#define UART_STATUS (*(volatile int *const)(UART_BADDR))
#define UART_CFG    (*(volatile int *const)(UART_BADDR+1))
#define UART_CONFIG (*(volatile int *const)(UART_BADDR+2))
#define UART_CMD    (*(volatile int *const)(UART_BADDR+3))
#define UART_MSG    (*(volatile int *const)(UART_BADDR+4))
#define UART_UBRS   (*(volatile int *const)(UART_BADDR+7))
```

driver.c

In der Datei *driver.c* sind mehrere Routinen zur Ansteuerung des miniUARTs zusammengefasst. Diese können, müssen aber keineswegs zur Programmierung verwendet werden. Der Verwendungszweck jeder einzelnen Funktion ist aus den Kommentaren ersichtlich.

```
/* miniUART-Modul ****************************************/

/* Initialisiert den miniUART */
int UART_init(void)
{
  int dummy;
  /* Even Parity, 2Stop, TrCtrl, MsgLength=8 (Standard) */
  UART_CONFIG = 0xB700;
  UART_UBRS = 0x1B20; /* Baud Rate 57600 (Standard) */

  /* clear possible RBR from previous operation */
  dummy = UART_MSG;
  return 0;
}

/* Sendet Daten */
int UART_send(int data)
```

```
{
  /* wait for TBR=1 */
  while ((UART_STATUS & 0x0200) != 0x0200)
    {
      asm("nop");
    }

  UART_MSG = data;
  /* Start transmission */
  UART_CMD = 0x0018;

  /* wait for TBR=1 */
  while ((UART_STATUS & 0x0200) != 0x0200)
    {
      asm("nop");
    }

  /* Stop transmission */
  UART_CMD = 0x0000;

  return 0;
}

/* schaltet receiver ein */
int UART_receive_enable(void)
{

  /* Start receiver */
  UART_CMD = 0x0020;

  return 0;
}

/* schaltet receiver aus */
int UART_receive_disable(void)
{

  /* Stop receiver */
  UART_CMD = 0x0028;

  return 0;
```

```
}

/* liefert RBR-Flag */
int UART_checkRBR(void)
{
  return (UART_STATUS & 0x0400);
}

/* liefert TBR-Flag */
int UART_checkTBR(void)
{
  return (UART_STATUS & 0x0200);
}

/* Wartet bis Daten empfangen wurden (polling) */
int UART_receive_wait(void)
{

  UART_receive_enable();

  /* wait for RBR=1 */
  while ((UART_STATUS & 0x0400) != 0x0400)
    {
      asm("nop");
    }

  UART_receive_disable();

  return 0;
}

/* liefert die Nachricht */
int UART_getmsg(void)
{
  int data;

  data = UART_MSG;

  return data;
}
```

```
/* liefert die Statusflags */
int UART_getstatus(void)
{
  int status;

  status = UART_STATUS;

  return status;
}
```

C Baudraten

Einige gebräuchliche Baudraten und die jeweiligen UBRS-Werte bei einer Taktfrequenz von 25 MHz sind in Tabelle 5 ersichtlich. Die Baudrate ist dabei in Bit pro Sekunde (bit/s) angegeben, die UBRS-Werte sind Hexadezimalzahlen.

Baudrate	UBRS	Baudrate	UBRS
9600	A2C2	115200	0D90
19200	5161	230400	06C8
38400	28B6	460800	0364
57600	1B20	921600	01B2

Tabelle 5: UBRS-Werte verschiedener Baudraten bei 25 MHz

Literatur

[Del02] Martin Delvai: "SPEAR Handbuch" *Technical report* Institut für Technische Informatik, Embedded Computing Systems Group, TU Wien, 2002

[Hub02] Wolfgang Huber: "Peripherieanbindung an SPEAR: Extension Modules" *Technical report* Institut für Technische Informatik, Embedded Computing Systems Group, TU Wien, 2002

[Raf05] Gallo Raffaele: "Revision and Verification of an Enhanced UART" *Diplomarbeit* Institut für Technische Informatik, Embedded Computing Systems Group, TU Wien, 2005