Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Bachelorarbeit

Studienrichtung
Technische Informatik

Entwurf eines Demonstrators für intelligente Tier-Implantate auf Basis eines ParaNut/RISC-V-Prozessors

Verfasser:
Elias Schuler
Bitzenhofer Weg 11
86453 Dasing
+49 1525 3599196
elias.schuler@hs-augsburg.de
Matrikelnr.: 2030805

Erstprüfer: Prof. Dr. Gundolf Kiefer
Zweitprüfer: Prof. Dr. Hubert Högl

Hochschule für angewandte
Wissenschaften Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@hs-augsburg.de

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADC	Analog-Digital-Wandler
API	Application Programming Interface
ASIC	Anwendungsspezifische integrierte Schaltung
ATT	Attribute Protocoll
AXI	Advanced eXtensible Interface
BLE	Bluetooth Low Energy
FIFO	First In First Out
FPGA	Field Programable Gate Array
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GPIO	General Purpose Input/Output
HLS	High Level Synthese
ICSC	Intel Compiler for SystemC
IoT	Internet of Things
JTAG	Joint Test Action Group
LSB	Least Significant Bit
PDF	Portable Document Format
PMOD	Peripheral Module Interface
PMW	Pulsweitenmodulation
UART	Universal Asynchronous Receiver/ Transmitter
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation

Nicht nur Menschen leiden unter Varianten der Stoffwechselstörung Diabetes, bei der der körpereigene Blutzuckerspiegel nicht korrekt reguliert werden kann [6], sondern auch Tiere wie zum Beispiel Hunde, Katzen oder Pferde [7]. Für Menschen gibt es hierfür schon von mehreren Anbietern Lösungen, über die mit Hilfe eines Sensors unter der Haut der Blutzuckerspiegel bequem auf dem Smartphone abgelesen werden kann.

Um Tierbesitzern eine leichtere Möglichkeit zur Überwachung des Blutzuckerlevels ihrer Haustiere zu verschaffen, beschäftigt sich diese Bachelorarbeit mit dem Konzept eines intelligenten Tierimplantats, welches vorher von Studierenden des Studiengangs „Interaktive Mediensysteme“ unter der Betreuung durch Doktorin Claudia Gerth entworfen wurde. Ihr Grobkonzept sah ein Bluetooth-fähiges Tierimplantat auf Basis des ParaNut-Prozessors vor, welches mit Hilfe eines Glukosesensors den Blutzuckerspiegel eines Tieres misst, die gemessenen Daten auf dem ParaNut verarbeitet und per Bluetooth die Ergebnisse an ein mobiles Endgerät zurückliefert. Die Studenten beschäftigten sich hierbei weitestgehend mit der Marktanalyse und der Vermarktung, weniger jedoch mit der technischen Umsetzung.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist nun der Entwurf eines Demonstrators für genau solch ein intelligentes Tierimplantat und die weitestmögliche Umsetzung eines ersten Prototypen dafür. Hierfür wurde analysiert, welche Komponenten erforderlich sind, und wie die Anbindung an den ParaNut aussehen soll.

Mit diesem verfeinerten Konzept wird ein erster Demonstrator entworfen, welcher sich mit der Anbindung der Komponenten an den ParaNut befasst und den Kommunikationsablauf mit einem Bluetooth-Gerät regelt.

Um die Komponenten nicht nur für das spezielle Problem der Messung eines Glukosesensors nutzbar zu machen, sondern für beliebige andere digitale und analoge

Sensoren, sollte die Anbindung dabei mittels generischer Komponenten erfolgen. Dies umfasst zum einen die Schnittstellen-Module auf Seiten des ParaNut, zum anderen die Ansteuerung, die als Schnittstellen-APIs implementiert wurden: einer UART-API sowie einer GPIO-API. Als Erweiterung der beiden Schnittstellen-APIs wurde zudem eine API für die Verwendung des Bluetooth-Moduls umgesetzt und ein Demoprogramm für weitere Anwendungen erstellt.

Mit Hilfe dieser Komponenten wurde ein erster Prototyp umgesetzt, welcher die Funktionalität des entworfenen Konzepts zeigen und erste Einblicke in eine mögliche finale Version des intelligenten Tierimplantats geben soll. Anhand des Prototypen lassen sich auch erste Abschätzungen über die Energieaufnahme eines derartigen Systems ermitteln und mögliche Verbesserungen aufzeigen.

1.3 Aufbau der Arbeit

In Kapitel 2 Grundlagen werden die benötigten Grundlagen zum Verständnis der Arbeit dargestellt. Hierbei wird zuerst ein kurzer Überblick über den ParaNut-Prozessor und seine Erweiterbarkeit über das Whisbone-Bussystem gegeben. Die UART-Schnittstelle wird kurz angeschnitten. Der von „Texas Instruments“ produzierte TLC16C750-Chip wird erklärt, ebenso die Bluetooth-Komponenten, wie der BM70-Chip von „Microchip“.

Weiterhin wird im Kapitel 3 Stand der Technik die Thematik der kontinuierlichen Glukosemessung anhand bereits umgesetzter Messimplantate des Menschen dargestellt.

In Kapitel 4 Konzept und Analyse soll ein erstes technisches Konzept für einen Demonstrator vorgestellt werden. Zudem wird auf die Auswahl des Bluetooth-Moduls und die Kommunikation mit diesem eingegangen.

In Kapitel 5 Schnittstellenentwicklung am ParaNut werden die Schnittstellenerweiterungen am ParaNut-Prozessor dargestellt. Hierbei wird zuerst die Erweiterung um ein UART-Modul und anschließend die Erweiterung um ein GPIO-Modul behandelt. Zusätzlich wird jeweils die Entwicklung einer API für die beiden Module beschrieben.

In Kapitel 6 Bluetooth-Anbindung wird die Entwicklung einer API für das Bluetooth-Modul und das Demoprogramm des Tierimplantats dargelegt.

Das Kapitel 7 Ergebnisse gibt einen Überblick über die Messungen des ParaNut sowie des Bluetooth-Moduls. Des Weiteren werden Tests des Bluetooth-Moduls erläutert,

um die Anbindung des Moduls an den ParaNut zu überprüfen und die Ergebnisse der Entwicklung darzustellen.

Das letzte Kapitel 8 Zusammenfassung und Ausblick gibt ein abschließendes Fazit zu den Erkenntnissen und Erfolgen der Bachelorarbeit. Der abschließende Ausblick zeigt mögliche Weiterentwicklungen und die nächsten Schritte des Projekts.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der verwendeten Technologien und Komponenten dargestellt, die für das Verständnis der nachfolgenden Kapitel erforderlich sind.

2.1 ParaNut-Architektur

Unter dem Namen ParaNut versteht man eine Prozessorarchitektur, die an der Hochschule Augsburg durch die Forschungsgruppe Effiziente Eingebettete Systeme entwickelt wurde. Diese Prozessorarchitektur für FPGA-Systeme ist offen, skalierbar und RISC-V-kompatibel [23].

Die Implementierung des Prozessors ist hierbei in SystemC modelliert. Somit ist auf Basis desselben Codes sowohl eine Hardware-Synthese, als auch die Simulation des Prozessors möglich, wobei in der Hardware einige zeitkritische Module in VHDL modelliert sind [23].

Die Besonderheit der ParaNut-Architektur ist dabei ein spezielles Konzept der Parallelisierung, bei der der Fokus auf Parallelität auf Thread-Level und Daten-Level liegt. Die Kerne des Systems sind auf Platzbedarf optimiert, so verfügen die einzelnen Kerne über vier Operationsmodi, in denen sie betrieben werden können. Der Modus 3 stellt einen Kern mit vollem Funktionsumfang dar und wird auch als „Central Processing Unit“ (CePU) bezeichnet [12]. Des Weiteren gibt es noch den Modus 2, in dem die Hardware des Kerns keine Interrupts, Exceptions oder Privileged System Instructions ausführen kann. Dieser wird „thread mode“ genannt und ist für das Ausführen von multi-threaded Code ausgelegt. Modus 1 ist der sogenannte „Linked/Vector Mode“, dieser ist für die SIMD-Vektorisierung ausgelegt. Der Prozessor verfügt in diesem Modus nur über eingeschränkte Hardwareelemente und wird komplett über die Kontrollsignale der CePU gesteuert. Er stellt also einen noch eingeschränkteren Modus als Modus 2 dar. Der letzte Modus ist Modus 0, in dem der Kern komplett abgeschaltet ist [2].

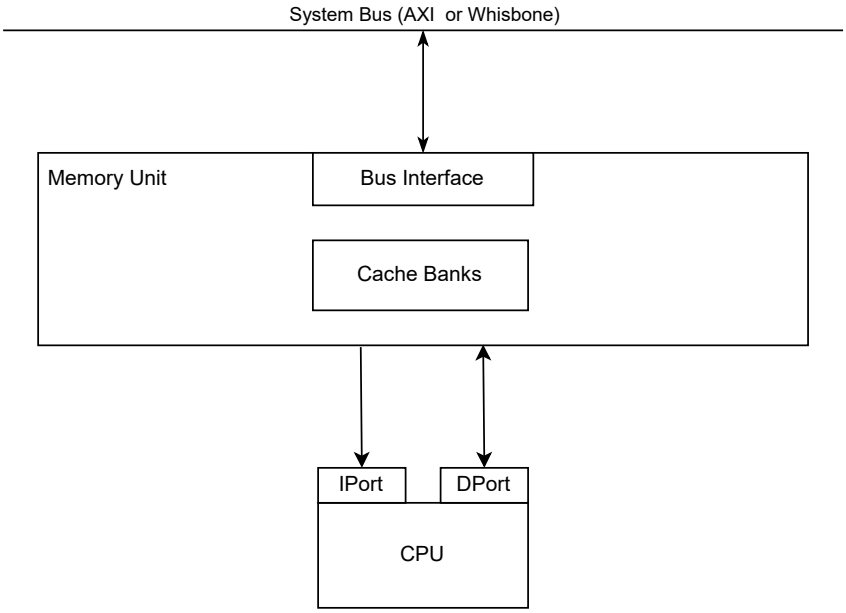

Abbildung 2.1: Vereinfachte Darstellung eines ParaNut-Prozessors nach [2]

Wie in Abbildung 2.1 dargestellt, verfügt der ParaNut auch über ein System Bus Interface, welches sich in der Memory Unit des Systems befindet. Von diesem Bus Interface aus können der AXI- und Whisbone-Bus angesteuert werden.

2.2 Asynchrone serielle Kommunikation (UART)

UART steht für „Universal Asynchronous Receiver/ Transmitter“ und ist eine Hardware-Schnittstelle, die eine serielle Kommunikation ermöglicht. Hierfür werden die beiden Leitungen RX und TX benötigt, wobei die RX-Leitung für das Empfangen von Daten und TX für das Senden von Daten zuständig ist. Das Übertragungsprotokoll basiert auf einem asynchronen Verfahren, sodass außer der RX- und TX-Leitungen keine weitere gemeinsame Taktquelle benötigt wird. Stattdessen startet die Kommunikation mit einem sogenannten Startbit, welches den Beginn einer Kommunikation signalisiert mit einem logischen „0“. Diesem Startbit folgen nun die Datenbits, welche zwischen 5 und 9 Bits beinhalten können und Zeichen genannt werden. Das Least Significant Bit (LSB) wird meist zuerst übertragen. Den Datenbits folgend kann ein Paritätsbit übermittelt werden, welches der Fehlererkennung dient. Hierbei gibt es zwei Möglichkeiten, wie ein solches Paritätsbit gesetzt wird: Entweder mit Hilfe einer geraden Parität, bei welcher die Gesamtanzahl an logischen „1“ in einer Nachricht gerade sein muss, oder einer ungeraden Parität, hierbei muss die Anzahl ungerade sein. Am Ende folgen ein oder zwei Stopbits mit einem logischen „1“. Diese Parameter müssen vor einer Kommunikation auf beiden Seiten konfigu-

riert sein, um eine erfolgreiche Kommunikation zu gewährleisten. Zudem muss die sogenannte Baudrate eingestellt werden, welche die Anzahl der übertragenen Bits pro Sekunde angibt. Auch die Start-, Stop- und Paritätsbits sind hier inkludiert [20].

2.3 Der TL16C750-Chip

Der TL16C750-Chip ist ein asynchrones Kommunikationselement, das dem in Abschnitt 2.2 aufgeführtem UART-Protokoll folgt. Er ist der Nachfolger beziehungsweise das funktionale Upgrade des weitverbreiteten TL16C550C UART-Moduls der Firma „Texas Instruments“. Der Chip ist über mehrere Register konfigurierbar, wobei folgende Parameter des UART-Protokolls einstellbar sind: 5- bis 8-Bit Zeichenauswahl, gerade-, ungerade- und keine Paritätsbit Generierung, 1, 1 1/2 oder 2 Bit lange Stopbits und ein einstellbarer Baudrate Generator, welcher über das interne „Divisor Latch Register“ konfiguriert wird. Mit Hilfe dieses Generators kann der TL16C750 die interne Taktrate selbst generieren [21].

Außerdem verfügt der Chip über einen aktivierbaren „First In First Out“ (FIFO)-Speicher, welcher die ein- und ausgehenden Daten zwischenspeichert und somit Last vom verbundenen Prozessor nimmt. Der hier verbaute FIFO-Speicher verfügt dabei über zwei Operationsmodi, den 16-Byte-Mode und den 64-Byte-Mode, es können also entweder 16 Bytes oder 64 Bytes an Daten zwischengespeichert werden [21].

2.4 Bluetooth Low Energy

Bluetooth Low Energy, kurz BLE, ist eine drahtlose Kommunikationstechnologie, welche erstmals in der Bluetooth Spezifikation 4.0 vorgestellt wurde. Im Gegensatz zum klassischen Bluetooth-Standard ist Bluetooth Low Energy auf einen deutlich geringeren Energieverbrauch ausgelegt, was es gerade für IoT-Geräte mit vergleichbar geringem Datenumsatz attraktiv macht [10].

In der Bluetooth Core Spezification sind folgende Layer eines Bluetooth Low Energy Geräts festgelegt, der „Physical Layer“, der „Link Layer“, der „Logical Link Control and Application Protocol“ (L2CAP), das „Security Manager Protocol“ (SMP), das „Attribute Protocol“ (ATT), das „Generic Attribute Profile“ (GATT) und das „Generic Access Profile“ (GAP). Im Folgenden werden kurz die drei für die Arbeit relevanten Layer erklärt: das GATT, das ATT, sowie das GAP [5].

2.4.1 Das Generic Access Profile

In Bluetooth Low Energy definiert den GAP-Layer vier spezifische Rollen: den „Broadcaster“, den „Observer“, das „Peripheral“ und das „Central“. Für die Arbeit ist jedoch nur Peripheral von Relevanz. Diese Rolle ist optimiert für Geräte, die eine einzelne Verbindung aufbauen wollen. Ein Peripheral ist eine Art Slave und stellt nur die Möglichkeit zur Verbindung mit sich bereit [5].

Die Datenübertragung findet als Peripheral in zwei Phasen statt, der „Advertising-Phase“ und der „Connection-Phase“. In der „Advertising-Phase“ sendet das BLE-Modul periodisch kurze Datenpakete. Andere Geräte können über diese Pakete das sendende BLE-Gerät entdecken. In der „Connection-Phase“ baut das Gerät eine Verbindung mit dem BLE-Gerät auf, und ein Datenaustausch kann stattfinden [22].

2.4.2 Das Attribute Protocol

Das ATT definiert die Rollen der zwei verbundenen Geräte in einer Client-Server-Architektur. Zudem ist das ATT für die Organisation der Daten in Attributen zuständig. Diese enthalten den Datenwert, einen „Universal Unique Identifier“ (UUID), einen Handle und die Berechtigungen [22].

2.4.3 Das Generic Attribute Profile

Das GATT erweitert den ATT-Layer und legt fest, wie die Daten und Informationen in einer BLE-Verbindung eines Profiles ausgetauscht werden. Das Profil definiert mögliche Anwendungen und gibt das generelle Verhalten des Geräts vor. Weiterhin sind im GATT verschiedene „Services“ und „Characteristics“ festgelegt, welche dem verbundenen Client mitteilen, wie eine Verbindung strukturiert ist [22].

Ein „Service“ stellt einen Container dar, der verschiedene „Characteristics“ in passende Gruppen einteilt [22].

Die „Characteristics“ sind die Attribute, die ein „Service“ übermitteln kann und enthalten einen Datenwert, einen Deskriptor, welcher zusätzliche Informationen bereitstellt, und Eigenschaften [22].

2.5 Microchip BM70

In diesem Teil wird auf das Bluetooth-Modul BM70 der Firma „Microchip“ eingegangen und die grundlegende Funktionalität des Geräts mit den Rahmenparametern erläutert. Infolgedessen wird die Funktionsweise des sogenannten „Auto Pattern Mode“, einem Operationsmodus, in dem die Hardwarefunktionalität des Bluetooth-Moduls eingeschränkt ist, dargelegt. Zuletzt wird der im „Auto Pattern Mode“ verwendete Bluetooth-Service „Microchip Transparent UART Service“ erläutert.

2.5.1 Grundlegende Funktionalität

Das BM70 Bluetooth Low Energy Module ist ein Bluetooth 5 zertifiziertes Modul der Firma „Microchip“. Es verfügt über ein UART-Interface, mehrere konfigurierbare General Purpose Input Output (GPIO)-Anschlüsse, einen Analog-Digital-Converter (ADC) sowie Möglichkeiten zur Pulsweitenmodulation (PMW). Das BM70 hat eine Reichweite von 50 Metern und muss mit einer Spannung zwischen 1,9 V und 3,6 V versorgt werden. Bei einer Stromversorgung von 3,0 V ist ein Durchschnittsstrom von Tx = 3,3 mA und Rx = 3,2 mA bei einem Verbindungsintervall von 18,75 ms angegeben [16].

Das Modul verfügt über mehrere Operationsmodi, den Configure Mode und den Run Mode, wobei letzterer in mehrere Untermodi aufgeteilt ist. Diese Modi werden über einen externen Pin am Modul gesteuert. Dabei entspricht ein logisches Level „0“ dem Configuration Mode und ein logischer Level „1“ dem Run Mode. Dieser Pin wird nur nach einem Reset über den RST_N Pin abgetastet oder nach einem Einschalten des Moduls. Intern sind sowohl RST_N als auch der Pin zur Modeauswahl mit einem Pullup-Widerstand versehen und somit standardmäßig auf einem logischen Level „1“.

Im Configure Mode können verschiedene Konfigurationen vorgenommen werden. Es können die Funktionalitäten der einzelnen GPIO-Pins festgelegt, Einstellungen des UART-Interface getroffen, sowie der Modus für den Run Mode ausgewählt werden. Es sind dazu auch noch weitere Einstellungen möglich. Für diese wird ein UI-Tool von „Microchip“ bereitgestellt. Das Protokoll für diesen Konfigurationsmodus ist nicht in der Dokumentation enthalten [17].

Je nach der getroffenen Einstellung im Configuration Mode wird im Run Mode entweder in die sogenannte „Auto Operation“ gestartet, welcher näher in Abschnitt

2.5.2 erläutert wird, oder die sogenannte „Manual Operation“. Der Unterschied zwischen diesen Modi liegt darin, dass im Auto Operation Mode die Hardwarefunktionalität des Moduls limitiert ist, während der Entwickler im Manual Operation Mode vollen Zugriff auf alle Steuerungselemente des Moduls hat. Weiterhin muss jede Operation des BM70-Moduls gesteuert werden, sowie das Konfigurieren aller Parameter. Dieser Modus ermöglicht den vollen Zugriff auf alle GPIO-Funktionalitäten des Boards, ebenso wie auf den internen ADC und PMW. [17]

2.5.2 Der Auto Operation Mode

Nun wird näher auf den Auto Operation Mode eingegangen, da dieser in der Umsetzung des ersten Demonstrators zur Verwendung kommt. Wie in Abschnitt 2.5.1 schon beschrieben, ist der Auto Operation Mode ein funktionseingeschränkter Modus des BM70 Moduls. In diesem stellt das Modul eine reine Datenpipe zwischen dem Hostcontroller und dem mobilen Endgerät dar und emuliert mit Hilfe des in Abschnitt 2.5.3 genannten Microchip Transparent UART Service eine direkte UART-Verbindung der beiden Geräte. [17]

Funktionseingeschränkt bedeutet, dass kein dauerhafter Zugriff auf das sogenannte „Command Set Protocol“ zur Verfügung steht. Dieses ist nur in einem Zeitfenster, das im Configuration Mode definiert wird, nach Eintreten in den Auto Operation Mode möglich. Dies ist in Abbildung 2.2 zu sehen. Dieses Protokoll ermöglicht es im Auto Pattern Mode einzelnen Parameter des Moduls zu steuern, wobei der Funktionsumfang des Protokolls im Vergleich zum Manual Mode eingeschränkt ist. Des Weiteren ist die Funktionalität eingeschränkt, da das Modul im Auto Operation Mode einer vorgegebenen Konfiguration des GAP-Layers folgt, so kann nur die Rolle eines Peripherals eingenommen werden. Der Zustand des Bluetooth-Moduls und die Zustandsübergänge sind durch das Modul selbst geregelt. Hierfür ist der in Abbildung 2.2 gezeigte Zustandsautomat zuständig. Auf die Zustände der Abbildung wird in den folgenden Paragraphen noch weiter eingegangen. [17]

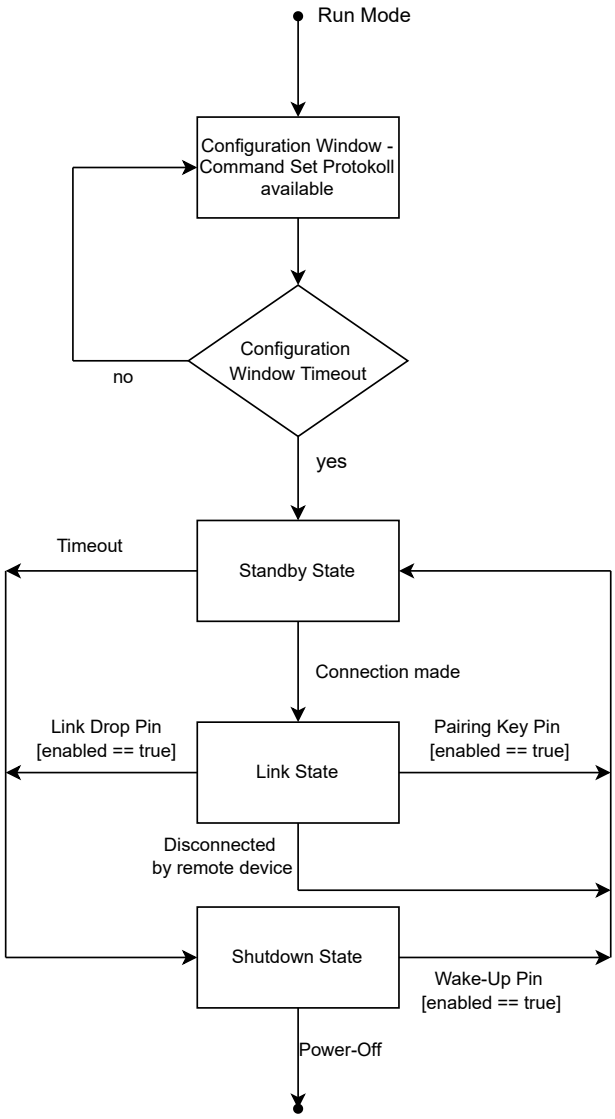

Abbildung 2.2: Zustandsmaschine des Auto Operation Mode nach [17]

2.5.2.1 Der Standby State

Im Standby State ist das Modul discoverable und connectable. Es schickt Advertisement-Packages nach außen und andere Geräte können sich verbinden. Wie in Abbildung 2.2 zu sehen, wird nach einem einstellbaren Timeout in den Shutdown Mode gewechselt. Dieser Timeout ist per „Command Set Protocol“ einstellbar und teilt sich wiederum in zwei Phasen auf, dem Fast Advertisement und dem Slow Advertisement. Die jeweiligen Intervalle zwischen den Advertisement Packages können für den jeweiligen Modus auch über das Protokoll eingestellt werden. Bei einer Verbindung durch ein Gerät wird in den Link State gewechselt, was in Abbildung 2.2 zu sehen ist. [17]

2.5.2.2 Der Link State

Im Link State besteht eine aktive Verbindung mit einem anderen BLE-Gerät. Es können Daten empfangen und gesendet werden. Da das Gerät, wie schon in oberen Paragraphen erwähnt, im Auto Operation Mode nur als Peripheral dient, können die Parameter der Verbindung nicht komplett angepasst werden. Es können aber bestimmte Limitationen vom verbundenen Gerät angefragt werden. Der Link State ist in zwei Phasen unterteilt, in der ersten Phase ist ein Gerät verbunden, diese wird in dieser Arbeit als „Bluetooth Low Energy-Connected Mode“ bezeichnet und in der zweiten Phase ist eine aktive Datenübertragung mit Hilfe des „Microchip Transparent UART Service“ möglich, diese wird in der Arbeit als „Transparent Service Ready Mode“ bezeichnet [17].

2.5.2.3 Der Shutdown State

Im Shutdown State befindet sich das BM70-Modul in einem Deep-Sleep Power Mode und es kann keine aktive Kommunikation stattfinden. Dieser Modus verbraucht am wenigsten Strom. Um aus diesem State zurückzukehren, muss vom Hostprozessor aktiv ein Signal über den WakeUp-Pin gesendet werden [17].

2.5.2.4 Die GPIO-Pins

Um die Limitationen des Auto Operation Mode zu mindern, wurden von Microchip in diesem Modus zusätzliche Pins definiert, welche spezielle Funktionalitäten umsetzen. Im Folgenden soll auf die für die Bachelorarbeit relevanten Pins eingegangen werden. Es wurden zwei Pins für die Status Indication zur Verfügung gestellt, denen die einzelnen Status zugeordnet sind, siehe Tabelle 2.1. Daraus ist zu erkennen, dass sich der Shutdown Mode zum Shutdown State des Paragraphen 2.5.2.3 und der Standby Mode zum Standby State 2.5.2.1 zuordnen lassen. Der Link State ist in die beiden Unterzustände „Bluetooth Low Energy-Connected Mode“ und „Transparent Service Ready Mode“ unterteilt, wie in Paragraph 2.5.2.2 erläutert [17].

Status Pin 1	Status Pin 2	Status
H	H	Shutdown Mode
H	L	Standby Mode
L	L	Bluetooth Low Energy-Connected Mode
L	H	Transparent Service Ready Mode

Tabelle 2.1: Zuordnung der Pin-Zustände zum Status nach [17]

Diese Pins sind in der Standardkonfiguration des Boards auf die Pins P1_0 und P1_1 des Boards gelegt [15].

Des Weiteren ist der WakeUp-Pin von Relevanz, denn dieser ermöglicht aus dem Shutdown Mode in den Standby Mode zurückzukehren. Dieser Pin ist low-aktiv, er reagiert also auf den logischen Wert „0“ als aktiv. Nach Aktivierung des Pins dauert es 43ms bis das BM70-Modul in den Standby Mode zurückkehrt. Dieser ist fest dem Pin P23 zugeordnet. Er ist auch im Manual Mode verfügbar [17].

2.5.3 Der Microchip Transparent UART Service

Bei Transparent UART handelt es sich um einen GATT-Service, der von „Microchip“ entwickelt wurde, und dem Bluetooth-Modul BM70 einen bidirektionalen Datenaustausch ermöglicht. Hierbei werden die folgenden beiden „Characteristics“ verwendet, der UART TX und der UART RX. Der Service ermöglicht den beiden verbundenen BLE-Geräten eine serielle Datenübertragung, sie emuliert eine herkömmliche UART-Schnittstelle. Somit kann auf dem verbundenen Prozessor wie mit einer herkömmlichen UART-Schnittstelle gearbeitet werden [18].

2.5.4 Das BM70 Pictail Evaluation Board

Das BM70 Pictail Evaluation Board dient dazu, die Funktionen des BM70-Moduls zu evaluieren. Dazu verfügt es über mehrere Jumper Konnektoren, über welche unter anderem gesteuert werden kann, welche Quelle zur Stromversorgung des BM70 genutzt wird. Des Weiteren verfügt es über einige Schalter und Taster, mit denen Funktionen des Boards überprüft werden können, sowie über das sogenannte Pictail Interface, welches die Pins des BM70-Moduls mit einem speziellen Pinstecker verbindet [14].

3 Stand der Technik

3.1 Glukosemessimplantate für Menschen

Bei menschlichen Patienten sind Glukosemessimplantate schon weitverbreitet, so tragen zwei von drei Patienten mit Typ-1-Diabetes und jeder 10. Patient mit Typ-2-Diabetes ein System zur kontinuierlichen Glukosemessung. Ein Großteil dieser Implantate funktioniert indem ein Sensor unter die Haut eingesetzt wird, der Sender bleibt an der Hautoberfläche und wird auf ihr angebracht. Die Funktionalität der Systeme kann sich hierbei stark unterscheiden, so gibt es zum einen die Methode der kontinuierlichen Glukosemessung, zum anderen die der Flash-Glukosemessung. Bei der Flash-Glukosemessung speichert der Sensor die Werte der Messung bis zu einer Stunde und diese können mit einem Smartphone abgefragt werden. Dies ist auch das verbreitetere System der beiden. Bei der kontinuierlichen Glukosemessung werden die Werte des Sensors automatisch an einen Empfänger gesendet. Bei vielen dieser Systeme können auch Schwellwerte eingestellt werden, welche bei einer Überschreitung oder Unterschreitung zu einem Alarm führen. Die verschiedenen Sensoren haben hierbei eine maximale Lebensdauer von bis zu sechs Monaten [13].

Die eingesetzten Sensoren unterscheiden sich hierbei auch in der Art, wie der Blutzuckerspiegel gemessen wird. Drei verschieden Messtechniken sind dabei verbreitet, die enzymbasierte Messung, die enzymbasierte, elektrochemische Messung und die Messung über Fluoreszenz [11].

3.2 Glukosemessimplantate für Tiere

Diabetes tritt bei rund 0,3 bis 1,33 % aller Hunde auf. Wie auch beim Mensch ist eine Überwachung des Blutzuckerspiegels sinnvoll, wobei diese entweder mittels eines Blutzuckermessgeräts oder mittels eines Messimplantats für den Menschen erfolgt [8].

4 Konzept und Analyse

In diesem Kapitel wird das Konzept für einen ersten Demonstrator dargestellt. Es wird zudem auf die benötigten Schnittstellen, sowie weitere Komponenten eingegangen.

Auf der einen Seite ist die Auswahl und Anbindung eines Bluetooth-Moduls erforderlich, um die Daten an den Endanwender weiterzugeben, auf der anderen Seite muss der Sensor über eine Schnittstelle ausgelesen werden können.

4.1 Auswahl eines Bluetooth-Moduls

Um den ParaNut-Prozessor Bluetooth-fähig zu machen musste ein passendes Bluetooth-Modul ausgewählt werden. Die Entscheidung für Bluetooth-Modul BM70 von Microchip wurde aufgrund der folgenden Eigenschaften getroffen:

Das Modul verfügt über zwei Betriebsmodi: dem Auto Pattern Mode und dem Manual Mode. Diese zwei Modi ermöglichen es gerade bei den neu entwickelten Schnittstellen am ParaNut Komplexität aus dem Gesamtsystem für einen ersten Demonstrator zu nehmen, da das Modul im Auto Pattern Mode einen eingeschränkten Funktionsumfang aufweist. So übernimmt hier das Modul an sich die Steuerung der Verbindung.

Zudem stellt es auch einen GATT-Service zur Verfügung, welcher sich Transparent UART nennt, dieser Service emuliert eine direkte UART-Verbindung zwischen dem Prozessor und dem mobilen Endgerät.

Durch diese Reduktion der Komplexität ist ein schnelles Testen des Prototyps möglich.

Trotzdem kann zur Nutzung des vollen Funktionsumfanges in den Manual Mode geschaltet werden, in dem der Entwickler vollen Zugriff auf alle Teile des Bluetooth-Stacks hat, in diesem kann das Bluetooth-Modul noch deutlich weiter optimiert werden und ein Maximum an Energieeffizienz erreicht werden. Auch ist im Manual Mode der Zugriff auf den ADC des Moduls möglich, der für die spätere Anbindung eines analogen Sensors genutzt werden kann.

Jedoch ist das BM-70-Modul ist im Vergleich zu einigen anderen Bluetooth-Modulen nicht das energieeffizienteste oder platzsparendste.

4.2 Konzeptentwurf

Wie in Kapitel 1 bereits aufgeführt, sollte ein erster Demonstrator für ein intelligentes Tierimplantat entworfen werden. Hierbei wurde in der Vorarbeit der Studierenden des Studiengangs Interaktiven Mediensystemen schon ein Grobkonzept ausgearbeitet, welches daraus bestand an den ParaNut ein Bluetooth Modul anzubinden und einen Sensor abzufragen. Dieses System sollte dann komplett unter die Haut eines Tiers implantiert werden [4]. An dieses Konzept wurde sich grundlegend gehalten, auch wenn technisch durchaus andere Konzepte denkbar wären.

Aspekte, die in dieser Arbeit nicht berücksichtigt sind, sind zum einen die Größe des finalen Implantats als auch die Auswahl des Glukosesensors selbst.

In einem ersten Demonstrator sollte der ParaNut um die benötigten Schnittstellen UART und GPIO erweitert werden. Mit Hilfe dieser Schnittstellen kann das BM70-Modul und ein Sensor angebunden werden. Der ParaNut sowie die Schnittstellen werden dazu auf ein FPGA-Board der Firma Digilent, das Zybo Z7, geflasht. Dieses verfügt über externe Pinverbindungen, den PMOD-Ports, über welche die Schnittstellen verfügbar gemacht werden. An diese Ports wird dann das BM70-Modul angebunden. Um hierbei mögliche Fehlerquellen zu reduzieren wird für den ersten Demonstrator als Demoboard das BM70 Pictail Board verwendet. Zur Vereinfachung der Schnittstellenimplementierung wird zunächst als „Senssor“ nur ein digitaler Reedschalter eingelesen. In nachfolgenden Arbeiten kann dieser dann durch einen analogen Sensor ersetzt werden. Dieser analoge Sensor wird dann über das ADC-Modul des BM70-Moduls angebunden. Der Reedschalter dient als Ersatz für den Sensor und soll demonstrieren, dass Sensorwerte ausgelesen werden können und auf Änderungen des Zustands reagiert werden kann.

4.3 Konzept der Kommunikation mit dem BM70

Durch die Verwendung des Auto Pattern Mode für den ersten Demonstrator regelt das Bluetooth-Modul an sich das Senden von Advertisement Packages, sowie den Verbindungsaufbau. Auch der GATT-Service, der zur Verfügung steht, ist mit dem in 2.5.3 beschriebenen Transparent UART Service vordefiniert. Bei der emulierten UART-Verbindung kann also ein eigenes Protokoll für die Abfrage von Sensordaten entworfen werden.

Das folgende Protokoll wurde für den Demonstrator entworfen: Zuerst wird intern am ParaNut die Initialisierung der Schnittstellen vorgenommen. Tritt eine falsche Konfiguration der GPIO-Pins auf, wird der Fehlercode `BLUETOOTH_GPIO_ERROR` zurückgegeben.

Nun wird zyklisch der Verbindungsstatus des BM70 Moduls überprüft. Tritt dieses in den Transparent Service Ready Mode, wird die Kommunikation mit dem Board gestartet. Hierbei wurden die folgenden Fehlercodes definiert:

- Der `BLUETOOTH_GPIO_ERROR` tritt ein, wenn die Konfiguration der GPIO-Pins nicht korrekt ist und ein nicht vorhandener Input- oder Output-Pin angesteuert wird.
- Der `BLUETOOTH_STATUS_ERROR` signalisiert einen nicht definierten Status des BM70-Boards.
- Der Fehlercode `BLUETOOTH_TIME_OUT` ist, wie der Name vermuten lässt, der Fehlercode, falls nach einem definierbaren Timeout-Fenster nicht der gewünschte Status erreicht wird.

Sollte jetzt eine aktive Tranparent UART Verbindung auftreten, wird zyklisch zuerst überprüft, ob sich der Sensorwert verändert hat. Ist dies der Fall, wird eine Nachricht „Sensor value updated: WERT“ an das verbundene Gerät gesendet. Später wäre hier eine Schwellwertdefinition denkbar, die bewirkt, dass eine Nachricht mit dem Wert des Sensors an das Smartphone geschickt wird, falls der eingestellte Schwellwert überschritten wird. Nach der Überprüfung des Sensorwerts wird für eine definierte Anzahl von Versuchen auf das Erhalten eines Kommandos gewartet. Die Länge des Kommandos wurde für den ersten Demonstrator auf drei Zeichen festgelegt. Sollte die maximale Anzahl an Versuchen erreicht werden, wird die Anzahl der bereits empfangenen Zeichen zurückgeliefert. Kommt es schon vor dem Einlesen des Kommandos zu einer Trennung der Bluetooth-Verbindung, wird eine Zeichenlänge von 0 Zeichen zurückgegeben.

Für den ersten Demonstrator wurden die beiden Kommandos „$S1“ und „$PI“ eingeführt. Bei den Kommandos wurde sich an folgende Syntax gehalten:

- „$“ signalisiert den Beginn eines Kommandos
- Das zweite Zeichen legt die Kategorie des Kommandos fest, hier „P“ für den ParaNut und „S“ für den Sensor
- Das letzte Zeichen legt die Aktion des Kommandos fest, hier „I“ für Informationen und „1“ für das Auslesen des ersten Sensors

Wird im Demonstrator ein nicht valides Kommando empfangen, wird ein „Please send a valid command“ an den Sender zurückgeschickt. Sollte eins der beiden erlaubten Kommandos ankommen, wird die angeforderte Information an den Sender zurückgeschickt. Dank dieser Kommunikation ist ein zyklisches Abfragen der Sensorwerte von einem verbundenen Gerät realisierbar. Dies kann in Form einer selbst entwickelten Applikation umgesetzt werden, welche in bestimmten Zeitabständen den Sensorstatus des Implantats abfragt.

5 Schnittstellenentwicklung am ParaNut

Um das Bluetooth-Modul an den ParaNut anzubinden, mussten sowohl ein UART-Modul als auch ein GPIO-Modul Teil des ParaNut-Systems werden. Diese Module sowie die implementierten APIs werden in den folgenden Abschnitten erläutert.

5.1 UART-Modul

Für die Kommunikation zwischen dem ParaNut und dem BM70-Modul muss eine UART-Schnittstelle zur Verfügung stehen. Bisher wurde am ParaNut für das Schreiben von Programmen auf die UART-Schnittstelle des Zybo Z7 Boards gesetzt, welche über den auf dem Board vorhandenen ARM-Prozessor arbeitet. Eine Nutzung dieser UART-Schnittstelle wäre auch für den Demonstrator denkbar, da jedoch in der finalen Version des Tierimplantats dieser ARM-Prozessor mit der gegebenen UART-Schnittstelle nicht zur Verfügung steht, musste am ParaNut selbst eine UART-Schnittstelle implementiert werden.

Diese wird auch für die Arbeit von Lukas Bauer benötigt, in dieser sollte unter anderem ein eigener Bootloader für das Übertragen von Programmen erstellt werden, sodass die Übertragung von Programmen nicht mehr über die UART-Schnittstelle des Zybo Boards stattfindet. Damit wird der ParaNut unabhängiger von dem Zybo Z7 Board. Aus diesem Grund fand die Arbeit am UART-Modul in Zusammenarbeit mit Lukas Bauer statt. Nähere Details hierzu befinden sich in seiner Bachelorarbeit [3].

5.1.1 Auswahl des UART-Moduls

Für die Auswahl eines UART-Moduls waren bestimmte Voraussetzungen gegeben: Das Modul sollte über eine Schnittstelle zum ParaNut, also eine Ansteuerung über den Whisbone-Bus verfügen. Außerdem sollte das Modul kompatibel mit Linux sein, da derzeit in einem Projekt am ParaNut weiter an der Umsetzung von Linux auf dem ParaNut gearbeitet wird.

In der Recherche ergab sich, dass es schon eine Vielzahl an Modulen gibt, die eine UART-Schnittstelle umsetzen, weswegen von einer Eigenentwicklung abzusehen war. Stattdessen wurde eine Anpassung eines ausgewählten Moduls durchgeführt. Zwei Module konnten gefunden werden, welche über eine Whisbone-Bus Schnittstelle verfügen: zum einen der „wbuart32“ und zum anderen der „wb_uart“.

Der „wbuart32“ steht auf GitHub zur Verfügung und hat eine Anbindung an den Whisbone-Bus, es ist das weniger komplexe Modul der beiden und ist in Verilog implementiert. Leider verfügt das Modul jedoch nicht nativ über einen Treiber auf Linux [9].

Daher fiel die Entscheidung zugunsten des komplexeren Moduls „wb_uart“. Dieses steht auf opencores zur Verfügung und ist in VHDL implementiert. Dieses Modul bildet das Verhalten des UART-Chips in Kapitel 2.3 nach. Er verfügt somit über eine Vielzahl an Einstellungsmöglichkeiten und auch die Möglichkeit eines FIFO-Speichers. Außerdem verfügt der TL16C750-Chip über Treiber unter Linux, weswegen für die Umsetzung dieses Modul als Basis der UART-Schnittstelle am ParaNut gewählt wurde. Das „wb_uart“-Projekt stellt die Weiterentwicklung des Projekts UART_16750 von Sebastian Witt dar und erweitert dieses um einen 8-Bit Whisbone Bus [1].

5.1.2 Aufbau des UART-Moduls

Das UART-Modul besteht in der SystemC-Umsetzung aus folgenden Modulen:

Das Hauptmodul `wb8_uart_16750` beinhaltet alle anderen Module. Es stellt die Hauptlogik des Moduls dar und ist für die Steuerung aller Submodule, das Einlesen der Signale, sowie die Registeroperationen zuständig. In Abbildung 5.1 sind die enthaltenen Untermodule aufgezeigt, die Module in gelber Farbe werden im Modul mehrfach eingesetzt.

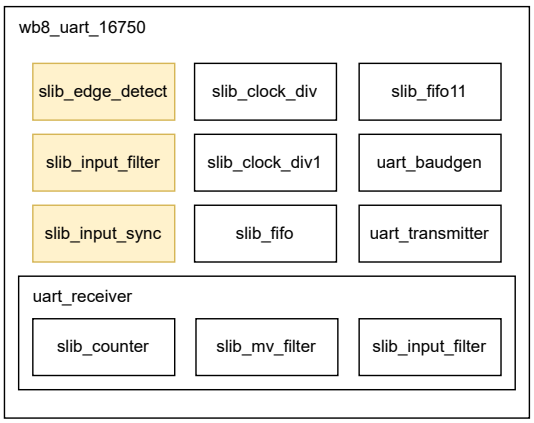

Abbildung 5.1: Zusammensetzung des Hauptmoduls

Im Modul `uart_receiver` wird der Receiverteil des UART-Moduls modelliert, hier finden unter anderem das Einlesen einer eingehenden UART-Nachricht statt, sowie die Überprüfung der Parität. Es ist aus folgenden Submodulen gemäß Abbildung 5.1 aufgebaut.

Der `uart_transmitter` ist das Gegenstück zum `uart_receiver`, stellt den Transmitterteil des UART-Moduls dar und ist für das Ausgeben einer UART-Nachricht mit Paritätsberechnung zuständig.

Die Steuerung der Interrupts erfolgt mit dem Modul `uart_interrupt`.

Für die Baudgenerierung wird das Submodul `uart_baudgen` verwendet.

Der `slib_mv_filter` stellt einen Mehrheitsfilter dar.

Für die Synchronisation der Eingangssignale mit dem UART-Modul wird das Modul `slib_input_sync` verwendet.

Ein Inputfilter wurde in der `slib_input_filter` und der `slib_input_filter2` umgesetzt. Diese Module unterscheiden sich nur in der Konfiguration, mehr dazu in der Arbeit von Lukas Bauer [3].

Auch ein FIFO-Speicher ist in der Form der Module `slib_fifo` und `slib_fifo_11` eingesetzt. Wie schon in der Umsetzung des Inputfilters unterscheiden sich diese Module in der Konfiguration [3]. Sie dienen dem Zwischenspeichern der Receive- und Transmitsignale.

Um auf Signalflanken zu reagieren, wird der `slib_edge_detect` eingesetzt.

Der `slib_counter` ist ein Zähler, welcher für die Generierung von einzelnen Bauds im Receiver verwendet wird.

Die `slib_clock_div` und `slib_clock_div1` stellen einen Taktteiler dar, dieser wird in zwei Ausführungen im Gesamtsystem verwendet und dient zum einen der Erzeugung eines internen Taktsignals und ist zum anderen Teil der Erzeugung der internen Baudgenerierung.

5.1.3 Schnittstellen des UART-Moduls

Im Folgenden wird auf die Schnittstellen und die nutzbaren Register des UART-Moduls eingegangen, diese sind in Abbildung 5.2 dargestellt.

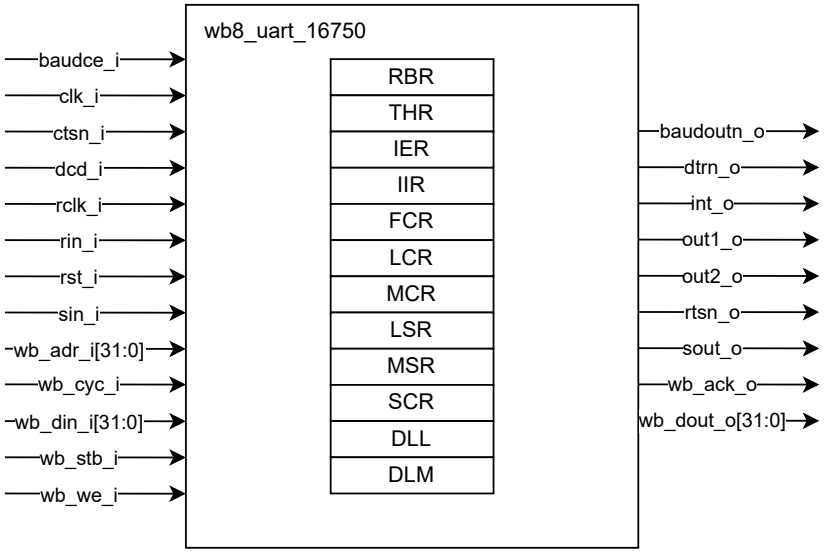

Abbildung 5.2: Schnittstellen des UART-Moduls

Die Signale, die mit **wb_** beginnen, stellen die Schnittstelle zum Whisbone-Bus dar. Die Signale **sin_i** und **sout_o** sind die Datenleitungen des UART-Moduls, sie bilden das RX- und TX-Signal ab. Die Signale **ctsn_i**, **dcd_i**, **rin_i**, **dtrn_o**, **int_o**, **out1_o**, **out2_o** und **rtsn_o** sind Modem-Status-Signale und wurden in der Anbindung an den ParaNut nicht verbunden. Genaueres zu diesen kann in der Dokumentation des TL16C750 nachgeschlagen werden [21]. Das Signal **baudce_i** aktiviert die interne Baudgenerierung des Systems. In der Umsetzung am ParaNut ist es dauerhaft aktiviert, indem es auf den logischen Wert „1“ geschaltet ist. Das Signal **clk_i** ist das Taktsignal für das Modul, es ist mit dem Taktsignal des ParaNut verbunden. Das **baudoutn_o**-Signal stellt die Baudrate in 16-facher Geschwindigkeit bereit, so wie sie am Transmitterteil des Moduls verwendet wird. **rclk_i** ist das Signal für die Taktrate des Receivers, es sollte der 16-fachen Geschwindigkeit der Baudrate entsprechen und wurde deshalb in der Anbindung am ParaNut mit dem Signal **baudoutn_o** verbunden.

Die Register, die in der Mitte des Moduls dargestellt wurden, sind in der Funktionalität deren des TL16C750-Chip identisch. Genaueres zu ihrer Funktion kann in der Dokumentation des Chips gefunden werden [21].

5.1.4 Übersetzung des Moduls in SystemC

Um das Modul mit dem ParaNut konform zu machen, musste es in SystemC übersetzt werden. Hierfür wurden anhand der Vorlage in VHDL passende Dateien in SystemC erstellt. Aus Gründen der Umsetzbarkeit in SystemC mussten einige Modifikationen durchgeführt werden.

Für die korrekte Ansteuerung der internen Register wird das eingehende Whisbone-Adresssignal intern angepasst.

Außerdem wurde eine Adressabfrage des Whisbone- Adresssignals hinzugefügt, wobei überprüft wird, ob das Modul von einem Whisbone-Master adressiert ist. Nur wenn das Modul auch adressiert ist, findet eine Verarbeitung der Bussignale statt.

Um unerwünschten Latches vorzubeugen, die durch Warnungen des ICSC-Tools identifiziert werden konnten, wurden mehrere Schaltwerke des Originalprojekts in zwei Prozesse aufgeteilt. Einen taktsynchronen Prozess, in dem der Zustandsübergang umgesetzt ist, und einen Schaltnetzprozess, in welchem die Ausgabe sowie die Übergangsfunktion des Schaltwerks beschrieben wird.

Weitere Unterschiede können in der Arbeit von Lukas Bauer gefunden werden [3].

5.1.5 Entwicklung von Testbenches

Um die korrekte Funktionalität des Moduls zu überprüfen, wurden für die einzelnen Submodule, aus denen das Modul aufgebaut ist, jeweils Testbenches erstellt. Zudem wurde auch das Hauptmodul getestet, um eine möglichst große Codecoverage aller Module zu gewährleisten. Zur Erstellung der Testbenches wurde die Funktionalität der in VHDL umgesetzten Module analysiert und passende Testszenarien umgesetzt, um zu überprüfen, ob sich die übersetzten Module in SystemC identisch verhalten. Das ursprüngliche VHDL-Projekt von Sebastian Witt enthielt eine Testbench für das Hauptmodul des UART-Moduls. Diese nutzt ein durch ein Perlskript erzeugtes Stimulus-File, in welchem die auszuführenden Tests des Hauptmoduls enthalten sind. In dieser Testbench werden die Initialisierung des Moduls, die Registerzustände nach Reset, die Interrupt, sowie die FIFO in verschiedenen Konfigurationen getestet [24]. Um diese Datei zu nutzen wurde in SystemC eine Funktion geschrieben, welche die Befehle des Stimulus-Files einliest und, je nach Befehl, die korrelierende

Aktion ausführt. Dazu wurden kleine Änderungen am Stimulus-File vorgenommen, da die Testbench im ParaNut zum Testen unter Realbedingungen die Frequenz des ParaNut in Standardkonfiguration mit 25 MHz nutzt. Da die Original-Testbench jedoch für eine andere Frequenz konzipiert war, führt dies zu Timingfehler. Deswegen wurde das taktspezifische Warten im Stimulus-File an die verwendete Frequenz unserer Testbench angepasst. Zusätzlich zu den Tests des Stimulus-Files wurden noch eigenständige Tests für das Hauptmodul geschrieben. Hierfür wird zum einen über die Datenports, also **sin** und **sout**, welche die UART RX- und TX-Leitung darstellt, geschrieben und getestet, ob sich das Modul korrekt verhält. Zum anderen werden alle Register vollständig beschrieben und gelesen.

5.1.6 High-Level-Synthese des UART-Moduls

Nachdem die Übersetzung nach dem neuen Standard für SystemC des ICSC-Tools durchgeführt wurde, die Implementierung folgte einer Vorlage von Marco Milenkovic [19]. Da diese jedoch nicht mit dem Standard des HLS Tool von Vivado übereinstimmt, traten folgende Probleme auf:

- Die Signal- und Portnamen, die im ICSC-Tool per geschweifter Klammer angegeben werden, sind in der Synthese mit dem HLS Tool von Vivado nicht unterstützt. Somit wurden Sie entfernt.
- Das Vivado HLS Tool unterstützt keine Prozesse des Typen `SC_CTHREAD`, diese verbinden das Reset-Signal nicht korrekt. `SC_CTHREAD` wurde daher durch `SC_METHOD` ersetzt, die nur sensitiv auf das Clock-Signal und das Reset-Signal ist. Dies erforderte auch eine Restrukturierung des Prozesses an sich. So wurde aus dem beispielhaften Prozesskörper in Listing 5.1 der Prozesskörper in Listing 5.2

Listing 5.1: Beispiel Funktionskörper bei der ICSC HLS

```
void Wb8Uart16750::IcscExample(){
    \\Reset-Teil
    .
    .
    .
    wait();
    while(true){
      \\Clocked-Teil
        .
        .
        .
        wait();
    }
}
```

Listing 5.2: Beispiel Funktionskörper bei der Vivado HLS

```
void Wb8Uart16750::VivadoExample(){
    if(reset){
    \\Reset-Teil
    .
    .
    .
    }else{
      \\Clocked-Teil
      .
      .
      .
    }
}
```

Es musste hier eine Abfrage des Reset-Signals eingefügt werden und die while-Schleife entfernt werden, so dass der Funktionsaufbau mit der einer `SC_METHOD` übereinstimmt.

Zudem kam es zu Problemen mit der Initialisierung von Submodulen, da diese im Standard des ICSC auch anders als in der HLS nach Vivado umgesetzt werden. Dieses Problem wurde nicht mehr weiterverfolgt, da eine komplette Umsetzung in der HLS des ICSC-Tool beschlossen wurden, welches Fortschritte im ersten Einsatz im ParaNut zeigte. In diesem sind zum einen die Ausgaben der Fehlermeldungen deutlich aussagekräftiger als die des Vivado HLS Tool, zum anderen ist die Synthese mittels ICSC auch schneller. Insgesamt ist ein Umstieg des ParaNut zum ICSC-Tool in Arbeit, was einen weiteren Grund für die Verwendung des Tools darstellt. Weitere Details zur Synthese mit Hilfe des ICSC-Tools können der Arbeit von Lukas Bauer entnommen werden [3].

5.1.7 Test mit realer Hardware

Um das in der Simulation getestete UART-Modul auch auf Hardware zu testen, wurde ein Testprogramm erstellt, welches eine Konfiguration der Register des UART-Moduls vornimmt. Hier wird eine Baudrate von 115200 eingestellt, die Zeichenlänge auf 8-Bit festgelegt und den 64-Byte-Mode aktiviert. Dies soll eine korrekte Funktionalität der Konfigurationsregister „LCR“, „FCR“, „DLM“ und „DLL“ sicherstellen. Zudem wird das Senden und Empfangen per UART getestet.

Für den Test wurden hierbei verschiedene Systeme gebaut:

Ein System, in welchem das UART-Modul über den ARM-Prozessor des Zybo Z7 Boards angesteuert wird, um das Modul unabhängig vom ParaNut zu testen. Hierfür wurde ein UART IP-Core erstellt, welcher aus dem UART-Modul und einem Buskonvertierer, dem `swb2maxi`, besteht. Dies ist nötig, da die Ansteuerung vom ARM-Prozessor nur über das „Advanced eXtensible Interface“ (AXI) des Zybo Z7 Boards möglich ist. Das angefertigte System ist in Abbildung 5.3 dargestellt.

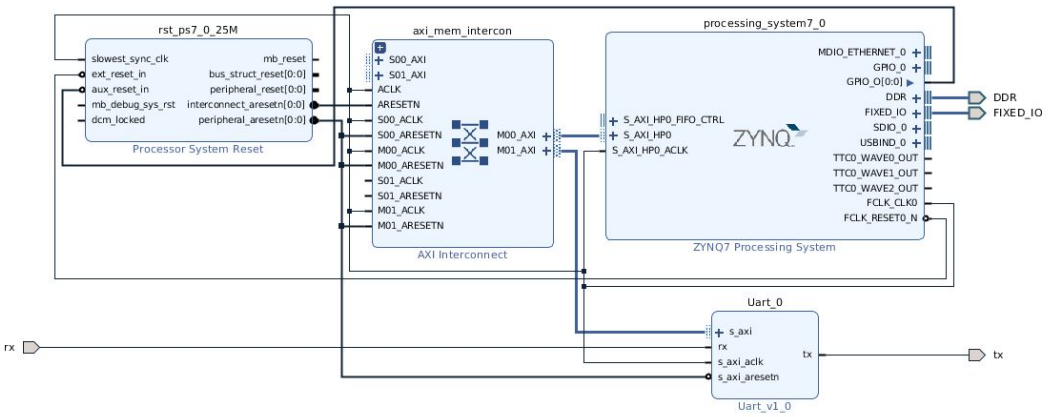

Abbildung 5.3: Blockdiagramm des Systems UART IP-Core verbunden an das Processing System in Vivado

In diesem findet man unter dem Namen „processing_system7_0“ (PS7) das Processing System des Zybo Boards, welches unter anderem den ARM-Prozessor enthält. Dieses ist über den „AXI Interconnect“ mit dem Modul Uart_0, dem UART IP-Core, verbunden. Mit diesem Aufbau konnte die korrekte Funktionsweise der Ansteuerung des UART getestet werden.

Des Weiteren erfolgte ein Test des UART-Moduls als IP-Core angebunden per AXI-Schnittstelle zum ParaNut. Dieser wird durchgeführt, um erste Rückschlüsse zur Anbindung des Moduls an den ParaNut zu ziehen. Genaueres zum Einbau des Moduls über die AXI-Schnittstelle ist in der Arbeit von Lukas Bauer zu finden [3]. Das

System ist hierbei ähnlich zu dem aus Abbildung 5.3 aufgebaut. Es wurde der „AXI Interconnect“ erweitert und an diesen der ParaNut angeschlossen. In Abbildung 5.4 ist das Gesamtsystem dargestellt.

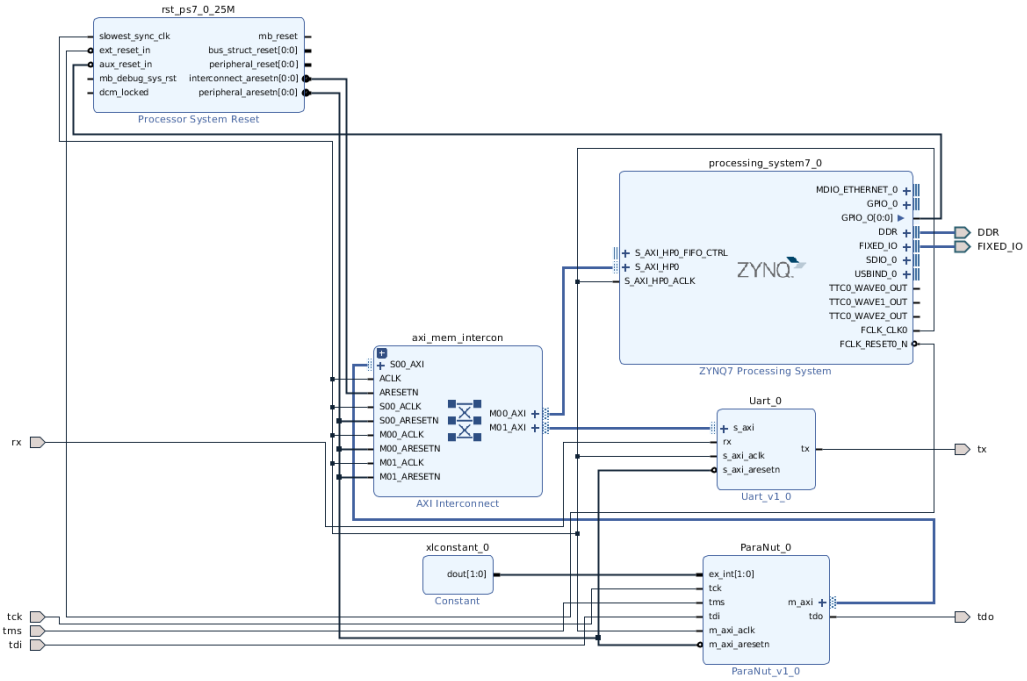

Abbildung 5.4: Blockdiagramm des Systems UART IP-Core verbunden an den ParaNut in Vivado

Über diesen „AXI Interconnect“ kann der ParaNut nun den UART IP-Core ansteuern. Über diese Ansteuerung war leider kein erfolgreicher Test möglich, so kam es zum Stillstand des Programms auf dem ParaNut sobald dieser Ausgaben über die printf-Funktion tätigte. Dabei werden diese Probleme jedoch nicht durch das UART-Modul ausgelöst. Sie ließen sich auch bei einer Erweiterung des „AXI Interconnect“ ohne ein verbundenes UART-Modul feststellen. Da es sich bei diesem Einbau nur um einen Test handelte, der UART IP-Core nicht der Auslöser war und der finale Einbau über den Whisbone-Bus stattfinden sollte, wurde das Problem nicht weiterverfolgt. Es besteht jedoch die Vermutung, dass das Problem durch die Memory Unit des ParaNut verursacht wird.

Der finale Test fand auf einem System statt, in dem das UART-Modul über den Whisbone-Bus direkt mit dem ParaNut verbunden ist. Der Einbau des Moduls per Whisbone-Bus ist hierbei in der Arbeit von Lukas Bauer geschildert [3]. Hierbei ist das UART-Modul direkt im IP-Core des ParaNut verbaut und wird direkt an das Bus-Interface der Memory Unit angeschlossen, wobei hier intern eine Adressüberprüfung eingefügt wurde. Die Tests des Systems verliefen erfolgreich. Dennoch

ließen sich Probleme mit der Konfiguration der Basisadresse feststellen, welche in der Arbeit von Lukas Bauer dokumentiert sind [3].

5.1.8 Peer-Test des UART-Moduls

Um weitere Fehler des UART-Moduls aufzufinden, wurde in Zusammenarbeit mit Lukas Bauer ein Peer-Test erstellt, welcher vom ParaNut-Team getestet wurde [3].

5.1.9 UART-API

Für die Erleichterung der Arbeit mit dem UART-Modul wurde eine API erstellt, sodass Zugriffe nicht mehr über ein direktes Schreiben der Register nötig sind.

Funktionen der UART-API

Listing 5.3: Auszug aus der lib_uart.h

```
1 void uart_init(struct uart_setup initSetup)
2 int8_t uart_data_ready();
3 int8_t uart_receive_char(int8_t enableBlocking, char* out, uint32_t
    timeOutAttempts);
4 void uart_send_char(char toSent);
5 void uart_write_register(uint32_t address, uint32_t data);
6 uint32_t uart_read_register(uint32_t address);
7 void uart_clear_rx_fifo();
8 void uart_clear_tx_fifo();
9 void uart_done();
```

In der `uart_init` Funktion wird das UART-Modul initialisiert, es werden die nötigen Register beschrieben, und der Entwickler kann mit Hilfe des Übergabeparameters `initSetup` einige Parameter der Konfiguration anpassen. Dazu wurden passende **Defines** erstellt, mit welchen die Anpassungen erleichtert werden.

Die `uart_data_ready` Funktion gibt zurück, ob mindestens ein Bit in dem FIFO-Speicher des UART-Moduls vorhanden ist.

Mit Hilfe der `uart_receive_char` Funktion ist es möglich, ein Zeichen aus dem FIFO-Speicher des UART-Moduls auszulesen. Diese Funktion kann entweder als blockierender Aufruf ausgeführt werden oder nicht blockierend. In der blockierenden Variante ist es zudem möglich, über den Parameter `timeOutAttempts` festzulegen, wie viele Versuche durchgeführt werden, bis die Funktion zurückkehrt.

Die Funktion `uart_send_char` schreibt einen Character in den Transmitter FIFO-Speicher des Moduls, welcher vom Modul selbst an die externen Leitungen weitergeschickt wird.

Die beiden `read_register` und `write_register` Funktionen ermöglichen ein Lesen sowie Schreiben der einzelnen Register des UART-Moduls. Für die Adressen der Register wurden ebenfalls `Defines` angelegt.

Die `clear_rx_fifo` und `clear_tx_fifo`-Funktionen löschen den jeweiligen FIFO-Speicher.
Die `uart_done` Funktion ist das Gegenstück zur `uart_init` und stellt quasi einen Destruktor dar.

Zusätzlich zur API wurde das erstellte Testprogramm für das UART-Modul kopiert und angepasst, sodass es auf Basis der UART API funktioniert.

Zu dieser API wurde auch eine Doxygen-Dokumentation erstellt, sie steht über ein Makefile Target zur Verfügung und kann in Form eines PDF-Dokuments als auch als HTML betrachtet werden.

5.2 GPIO-Modul

In diesem Kapitel wird die Entwicklung sowie der Einbau eines GPIO-Moduls für den ParaNut beschrieben. Dieses GPIO-Modul wird benötigt, um die Sensorwerte auszulesen sowie für die Ansteuerung des BM70 Bluetooth-Moduls im Auto Pattern Mode. So werden 2 GPIO-Pins für das Einlesen des Status des Bluetooth-Moduls benötigt sowie ein Output Pin für ein WakeUp-Signal.

5.2.1 Entwurf eines GPIO-Moduls

Die GPIO-Pins sollen über den Whisbone-Bus des ParaNut ansteuerbar sein, dieser soll die Output-Pins ansteuern und einzeln setzen können. Die Input Pins sollen vom ParaNut auslesbar sein und das anliegende logische Level zurückgeben.

Hierfür wurde der Whisbone-Bus als Schnittstelle vom GPIO-Modul zum ParaNut gewählt. Als Schnittstelle für die GPIO-Pins wurden die Signale **gpio_input_port** und **gpio_output_port** verwendet, wobei die Breite der Signale, also die Anzahl der Input- und Output-Pins, vor der Synthese konfigurierbar sein sollte, was in Abbildung 5.5 dargestellt ist.

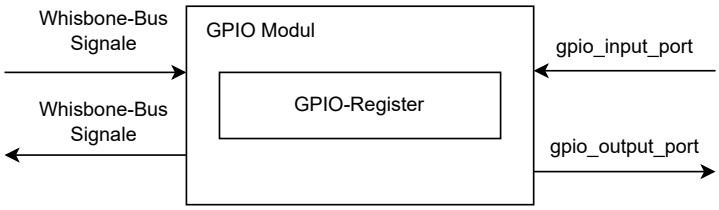

Abbildung 5.5: Aufbau des GPIO-Moduls

Sowohl die Schnittstelle des Whisbone-Bussystems als auch die GPIO-Pins lesen und schreiben auf ein internes Register, das GPIO-Register in Abbildung 5.2, die Input-Pins werden hierfür über einen eigenen Prozess und ein Zwischenregister eingelesen. Auf dieses Register wird im Kapitel 5.2.2 noch genauer eingegangen.

5.2.2 Implementierung eines GPIO-Moduls

Zur Implementierung wurde der für das EDS2-Praktikum vorhandene **example_-wb_slave** zur Hilfe genommen. Bei diesem handelt es sich um eine Vorlage, welche im ParaNut schon erfolgreich an den Whisbone-Bus angeschlossen wurde und zur Entwicklung eigener Submodule für den ParaNut-Prozessor dienen soll. Da diese Vorlage noch unter der HLS-Synthese mit dem Vivado HLS Tool entworfen wurde, derzeit aber die Bemühung besteht, dieses High Level Synthese Tool zu ersetzen und mit dem ICSC Tool auszutauschen, musste die Vorlage angepasst werden. So musste wie schon in Kapitel 5.1.6 die vorherige `SC_METHOD`, die nur sensitiv auf das Taktsignal war, umgewandelt werden und zu einem `SC_CTHREAD` abgeändert werden, was weitere Änderungen im Funktionkörper mit sich zieht, wie in Kapitel 5.1.6 beschrieben.

Des Weiteren wurden die Funktion einiger Signale aus dem **example_wb_slave** und die Logik für den Whisbone Select entfernt, da sie nicht benötigt werden. Außerdem wurde auch die Variable zur Steuerung, wie viele interne Register erzeugt werden, entfernt, da vorerst ein einziges 32-Bit Register ausreichend ist.

Nun zum Aufbau und der Funktionalität des Moduls. Das Modul besteht intern aus einem 32-Bit Register, welches sowohl für die Input-Pins als auch für die Output-Pins zuständig ist. Hierfür wird das Register zweigeteilt, die einzelnen Bereiche werden über festgelegte Werte aufgeteilt. So legt die `CFG_GPIO_AMOUNT` die Gesamtanzahl der Pins fest. Von dieser Gesamtanzahl abwärts bis zu `CFG_GPIO_OUTAMOUNT` befindet sich der Registerbereich, welcher für die Input-Pins zuständig ist, siehe Tabelle 5.1. Diese Variablen werden in Kapitel 5.2.4 noch genauer erläutert. Der Wert der Input-Pins wird in einem eigenen `SC_CTHREAD` von den externen Pins eingelesen

31	CFG_GPIO_AMOUNT CFG_GPIO_AMOUNT-1	CFG_GPIO_OUTAMOUNT CFG_GPIO_OUTAMOUNT-1	0
No functionality	Input-Pins	Output-Pins	

Tabelle 5.1: Register des GPIO-Moduls

und über eine Zwischenvariable gespeichert. Diese wird in der Hauptfunktion nun in das Register übertragen. Von `CFG_GPIO_OUTAMOUNT` -1 bis 0 liegt der Registerbereich, welcher für die Output-Pins zuständig ist. Diese werden im Hauptprozess auf die am Whisbone anliegenden Datenwerte gesetzt und in einem eigenen Prozess auf die externen Pins ausgegeben.

5.2.3 Testbench für die Anbindung

Um sicherzustellen, dass das entworfene Modul und die Implementierung korrekt funktionieren, wurde eine Testbench geschrieben, welche die Funktionalität des Moduls testet. Dazu wurden mehrere Hilfsfunktionen implementiert, die korrekte Whisbone Read- und Write-Transaktion nachstellen. Mit Hilfe dieser Funktionen wird als Erstes ein Schreiben des Registers über den Whisbone getestet. Hierfür wird einmal der komplette Bereich für den Output beschrieben und überprüft, ob die passenden Output-Pins aktiviert werden. Es wird auch versucht, in den Bereich zu schreiben, welcher nur für die Inputs zuständig ist und nicht beschreibbar sein sollte. Anschließend wird ein Reset durchgeführt und es wird der Input-Bereich des Registers getestet. Dabei werden an die einzelnen Input-Pins logische Werte angelegt und es wird überprüft, ob sich das Register korrekt verhält.

5.2.4 Einbau in den ParaNut per Whisbone-Bussystem

Für den Einbau des GPIO-Moduls in den ParaNut mussten mehrere Konfigurationsoptionen in die Konfigurationsdatei **config.mk** des ParaNut hinzugefügt werden. Es wurden die Optionen `CFG_GPIO_ENABLE`, `CFG_GPIO_BASE_ADDRESS`, `CFG_GPIO_AMOUNT` und `CFG_GPIO_OUTAMOUNT` hinzugefügt, die ermöglichen die Hardwaresynthese des Moduls an- und abzuschalten, die Ansteuerungsadresse des Moduls am Whisbone-Bus, sowie die Anzahl der aktiven Teile des Registers festzulegen, siehe Kapitel 5.2.2. Diese Optionen werden dann mit Hilfe der Makefiles des

ParaNut in die nötigen Dateien übertragen, zum Beispiel die **paranut-config.h**. Mit Hilfe der Makefiles wird auch `CFG_GPIO_INAMOUNT` berechnet, welche die Breite des Inputteils des GPIO-Registers beschreibt.

Außerdem mussten in der Hardware des ParaNut Anpassungen vorgenommen werden. So musste ein VHDL-Wrapper für die Verilog-Datei geschrieben werden, zu diesem Wrapper musste eine weitere Datei, die den Wrapper mit dem ParaNut-System verbindet, erstellt werden. Darüber hinaus musste das GPIO-Modul im ParaNut-System hinzugefügt werden, hier findet eine Überprüfung statt, ob das GPIO-Modul angesteuert wird. Wenn dies der Fall ist, werden die Whisbone-Signale des ParaNut nach außen abgeschaltet, so dass kein falscher Zugriff auf den AXI-Bus möglich ist. Im Toplevel-Modul des ParaNut wurden dann die Input- und Output-Leitungen des GPIO-Moduls zu einem `std_logic_vector` vereint, damit später mit diesem im Constraint-File gearbeitet werden kann.

Nachdem nun das GPIO-Modul im ParaNut eingebaut wurde, mussten für die Synthese auf die Hardware Anpassungen vorgenommen werden. So wurde ein neues Board-File für Vivado erzeugt, in welchem die GPIO-Pins als externe Signale erstellt wurden und mit Hilfe eines Constraint-File auf die PMOD Verbindungen des Zybos verbunden wurden. Des Weiteren wird in diesem Constraint-File festgelegt, dass es sich bei den Pins um 3,3 V Pins handelt, welche mit einem internen Pulldown-Widerstand ausgestattet sind. Zu beachten ist, dass die HLS mit dem ICSC-Tool neu ausgeführt werden muss, da die erzeugte Verilog Datei auch die Änderungen der Konfigurationsdatei erfahren muss.

5.2.5 GPIO-API

Für den erleichterten Zugriff auf das GPIO-Modul wurde eine API entworfen.

Die API ist in zwei Pin-Nummerierungen aufgeteilt: Die erste Pin-Nummerierung ist die für die Input-Pins, die andere ist für die Output-Pins zuständig. Dies verringert die Gefahr einer Verwechslung von Input- und Output-Pins. So greifen die Funktionen, die sich auf die Input-Pins beziehen, nur auf den Bereich des Registers zu, welcher für die Inputs zuständig ist, und starten hierbei bei Input-Pin 1. Selbiges gilt für die Funktionen, die sich auf die Output-Pins beziehen, nur dass diese mit dem Output-Pin 1 starten. Dies soll nun anhand einer Beispielkonfiguration veranschaulicht werden. In Tabelle 5.2 sind die einzelnen Pins eines PMOD Steckers des Zybo Z7 Boards dargestellt.

VCC	GND	4	3	2	1
VCC	GND	10	9	8	7

Tabelle 5.2: Pinverteilung eines PMOD-Steckers

Legt man nun in der **config.mk** einen Wert `CFG_GPIO_OUTAMOUNT` von 5 fest, sind die ersten 3 Pins der Tabelle 5.2 Input-Pins und mit der Nummer 1 bis 3 versehen. Ab Pin 4 der Tabelle 5.2 starten die Output-Pins, welche mit Output-Pin 1 starten.

Um die derzeitige Konfiguration der Pins auch bereitzustellen, wurden **Defines** erstellt, welche aus der **paranut-config.h** die Werte der **config.mk** bekommen. Dieses Schreiben der **paranut-config.h** erfolgt durch ein Makefile-Target des ParaNut.

Da die grundsätzliche Funktionalität der Pins und die Zugänglichkeit der Konfiguration erläutert wurde, folgt nun die Erklärung zu den Funktionen der API.

Funktionen der API

Listing 5.4: Auszug aus der lib_gpio.h

```
1 void gpio_init()
2 int8_t gpio_toogle_output(uint8_t outputpin);
3 int8_t gpio_set_output(uint8_t outputpin, uint8_t value);
4 int8_t gpio_get_output(uint8_t outputpin, uint8_t* out);
5 void gpio_write_register(uint32_t address, uint32_t data);
6 uint32_t gpio_read_register(uint32_t address);
7 void gpio_done();
```

Bei der `gpio_init` sowie der `gpio_done` handelt es sich quasi um einen Konstruktor und Destruktor. Bei der aktuellen Implementierung haben sie aber noch keine Funktionalität. Falls jedoch später ein Initialisieren oder Aufräumen nötig wird, können die beiden Funktionen angepasst werden.

Die `gpio_toogle_output` Funktion nimmt einen Output-Pin entgegen, und schaltet ihn auf den Zustand um, in dem er sich derzeit nicht befindet.

Die `gpio_set_output` Funktion ändert auch den Zustand eines Output-Pins, legt jedoch einen spezifischen Wert, der über die Funktionsparameter mitgegeben wird, fest.

Die `gpio_get_output` Funktion gibt den aktuellen Stand des Output-Pins zurück.

Weiterhin gibt es noch die `gpio_get_input`, welche für einen angegebenen Input-Pin den logischen Wert, der an den externen Ports anliegt, zurückliefert.

Zum Schreiben und Lesen des gesamten Registers gibt es noch die Methoden `gpio_read_register` und `gpio_write_register`.

Zum Test der API wurde ein Programm geschrieben, das die GPIO-API initialisiert, dann die Input-Pins einliest und die Werte dieser auf der Konsole ausgibt. Anschließend werden die `gpio_toogle_output` und die `gpio_set_output` aufgerufen, um die Output-Pins nacheinander ein- und wieder auszuschalten. Zuletzt wird das Register komplett ausgelesen.

Genauso wie für die UART-API wurde auch für die GPIO-API eine umfangreiche Dokumentation mit Hilfe von Doxygen erstellt.

6 Bluetooth-Anbindung

Nach der Auswahl des Bluetooth-Moduls (siehe Kapitel 4) wurde für die Anbindung an den ParaNut zur Vereinfachung künftiger Entwicklungen ebenfalls eine API erstellt. Diese wird im folgenden beschrieben.

6.1 Bluetooth-API

Da die Ansteuerung des BM70-Moduls im Auto Operation Mode sowohl Funktionen des GPIO-Moduls als auch des UART-Moduls benötigt, wäre auch hier die Implementierung einzelner Programme aufwändig. Zur Minimierung dieses Aufwands sollte auch hier eine API entwickelt werden. Diese inkludiert die GPIO-API, wie auch die UART-API, um einen vereinfachten Zugriff auf die beiden benötigten Schnittstellen zu haben. Zudem wird die libparanut eingebunden, da in ihr die Implementierung der `pn_usleep` umgesetzt wurde.

Für die API ist ein aktives Warten auf einen Verbindungszustand nötig. Da dafür keine Lösung vorhanden war, wurde die libparanut um eine aktive Wartefunktion erweitert. Diese Funktion wurde `pn_usleep` genannt, sie nimmt einen Parameter entgegen, der festlegt, wie viele Mikrosekunden aktiv gewartet wird und verwendet für die Zeitmessung die bereits vorhandene Funktion `pn_time_ns` der libparanut. Aus dieser Zeitmessung wird mit Hilfe des Funktionsparameters also ein Sollwert berechnet. Es trat jedoch folgendes Problem auf: In der `pn_time_ns`-Funktion wird zur Berechnung der Zeit die Tickzahl des Prozessors zurückgeben. Dies wird mit Hilfe eines Assemblerbefehls umgesetzt, der zwei 32-Bit Register ausliest und diese als Rückgabewert zurückgibt. Als Datentyp für diese Rückgabe wurde aber ein long int verwendet, was in der Kompilerversion einem 32-Bit Integer entspricht. Deswegen kommt es bei der Funktion schon nach 32-Bit zu einem Überlauf. Da aber der Sollwert ein long long int ist und somit 64-Bit enthalten kann, kommt es zu einem Sollwert, der nie von der `pn_time_ns` erreicht werden kann, und somit zu einem endlosen Warten. Das Problem ließ sich beheben, indem der Datentyp für den Assemblercode auf 64-Bit erweitert wurde. Im Zuge dieser Modifikation wurden auch die Datentypen des Assemblerbefehls, der `pn_time_ns` und der `pn_usleep` auf

fixed-width Integer-Typen umgestellt, sodass die Anzahl der Bits sofort ersichtlich ist und sie nicht kompilerabhängig ist.

Funktionen der lib_bluetooth_api

Listing 6.1: Auszug aus der lib_bluetooth.h

```
1  int8_t bluetooth_init(uint8_t statusPin1, uint8_t statusPin2,
       uint8_t wakeUpPin);
2  int8_t bluetooth_get_inputPinNumberStatusPin1();
3  int8_t bluetooth_get_inputPinNumberStatusPin2();
4  int8_t bluetooth_get_outputPinNumberWakeUpPin();
5  int8_t bluetooth_get_status();
6  int8_t bluetooth_wait_status(int8_t statusToReach, int64_t
       uSecIntervall, uint32_t timeOutAttempts);
7  int8_t bluetooth_send_data(int dataLength, char* transmitString);
8  int8_t bluetooth_data_ready();
9  int8_t bluetooth_receive_data(int dataLength,char* receivedString,
       uint8_t blockingEnabled, uint32_t timeOutAttempts);
10 int8_t bluetooth_wake_up();
11 int8_t bluetooth_done();
```

Die Funktion `bluetooth_init` und die `bluetooth_done` stellen den Konstruktor und den Destruktor dar. In der Init-Funktion werden die Init-Funktionen der GPIO-API und der UART-API aufgerufen, dabei wird das UART-Modul mit einer Baudrate von 115200, 8-Bit Zeichenlänge und 64-Byte FIFO-Speicher initialisiert. Zudem wird mit Hilfe der Funktionsparameter festgelegt, mit welchen Input- und Output-Pins die Status-Pins und der WakeUp-Pin des BM70-Board verbunden sind. Der Output-Pin wird zuletzt noch auf eine logische „1“ gesetzt, da der WakeUp-Pin low-aktiv ist. Parallel hierzu funktioniert die Done-Funktion. Sie ruft die Done-Funktionen der GPIO-API sowie der UART-API auf und der Output-Pin wird auf eine logische „0“ zurückgesetzt.

Die Funktionen `bluetooth_get_inputPinNumberStatusPin1`, `bluetooth_get_inputPinNumberStatusPin2` und `bluetooth_get_outputPinNumberWakeUpPin` geben die in der Init-Funktion definierten Input- und Output-Pins zurück.

Mit Hilfe der `bluetooth_get_status`-Funktion wird der Status des BM70-Moduls ermittelt. Hierbei werden die beiden Status-Pins eingelesen und der korrespondierende Status wie in Kapitel 2.5.2 zurückgegeben.

Die `bluetooth_wait_status`-Funktion ist eine Erweiterung der `bluetooth_get_status`-Funktion und wartet aktiv auf das Eintreten eines angegebenen Status. Hierbei wird das BM70-Modul aktiv gehalten und bei einem Eintreten des Shutdown-Mode wird das Modul über den WakeUp-Pin wieder in den

Standby-Mode gebracht. Weiterhin ist das Zeitintervall zwischen den Überprüfungen des Status einstellbar sowie die Anzahl an Versuchen nach der ein Timeout erfolgt.

Die Funktion `bluetooth_send_data` wird verwendet, um Daten an das BM70-Modul zu schicken und damit auch an das verbundene mobile Endgerät. Sie legt das übergebene Char-Array der Länge dataLength Zeichen für Zeichen in den FIFO-Speicher des UART-Moduls und sendet diese über den Ausgang des UART-Moduls ab. Dies ist nur möglich, wenn sich das BM70-Modul im Status „Transparent Service Ready Mode“ befindet.

Mit Hilfe der `bluetooth_data_ready`-Funktion lässt sich überprüfen, ob mindestens ein Zeichen in dem Receiver-FIFO-Speicher des UART-Moduls vorliegt.

Die Funktion `bluetooth_receive_data` wird verwendet, um Daten vom BM70-Modul zu empfangen. Hierbei gibt es ein blockierendes Empfangen, in dem so lange auf Zeichen gewartet wird, bis eine bestimmte Anzahl an Zeichen empfangen wurde. Zusätzlich kann hier noch festgelegt werden, ob nach einer bestimmten Anzahl an Versuchen das Warten abgebrochen wird. Zudem gibt es die Möglichkeit eines nicht blockierenden Empfangens, bei der überprüft wird, ob mindestens ein Zeichen im FIFO-Speicher des UART-Moduls liegt. Sollte dies der Fall sein, werden Zeichen empfangen, bis keine mehr anliegen. Um einen Abbruch des Auslesens während des Empfanges von Daten vorzubeugen, wird nach dem Auslesen eines Zeichens kurz gewartet, sodass Zeichen, die sich in der Übertragung befinden, ebenfalls eingelesen werden. Als Rückgabe erhält man von der Funktion die wirkliche empfangene Anzahl an Zeichen.

Um das BM70-Modul aus einem Shutdown Mode wieder aufzuwecken, steht die `bluetooth_wake_up`-Funktion zur Verfügung. Diese setzt den WakeUp-Pin für 10 ms auf den logischen Wert „0“ und deaktiviert ihn wieder, um nach insgesamt 43 ms zurückzukehren.

Wie auch für die anderen APIs wurde ebenfalls für die Bluetooth-API eine Doxygen-Dokumentation erstellt.

6.2 Programme für das Bluetooth-Modul

6.2.1 Das Demoprogramm der Bluetooth-API

Als Vorlage für ein Programm der Bluetooth-API wurde ein Demoprogramm für das BM70-Modul geschrieben. Dieses wurde für die Pinverbindung mit dem BM70, wie sie in Abbildung 6.1 dargestellt ist, entworfen. In dieser sind die Pins 1 und 2 der PMOD Connector JC als Input-Pins konfiguriert und als Status-Pins initialisiert. Der Pin 10 des PMOD Connectors JC ist als Output-Pin 5 konfiguriert und als WakeUp-Pin initialisiert.

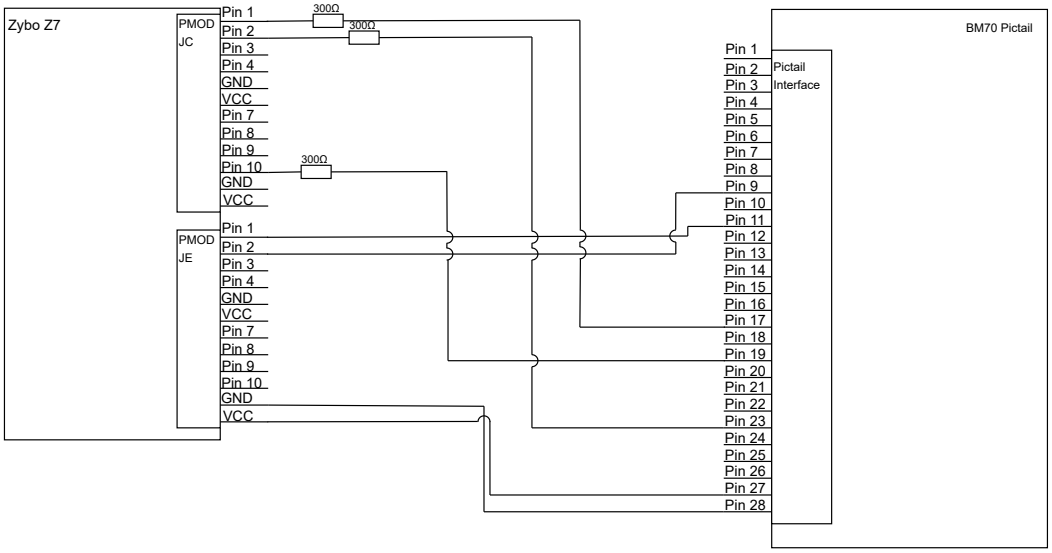

Abbildung 6.1: Verbindungen für das Demoprogramm

Mit Hilfe der `bluetooth_wait_status`-Funktion wird zyklisch überprüft, ob sich das BM70 in einer aktiven Transparent-UART-Ready-Verbindung befindet. Es wurde ein Zeitintervall von 20 µs zwischen den Prüfungen festgelegt. Falls dies der Fall ist, wird durch die Funktion `bluetooth_receive_data` auf eine Nachricht vom mobilen Endgerät gewartet, wobei dieses Warten nach 20.000.000 fehlgeschlagenen Versuchen terminiert. Wird eine Nachricht erhalten, wird sie eingelesen und überprüft, ob es sich um ein „Hallo“ handelt oder nicht. Bei einem erhaltenen „Hallo“ wird ein „Hallo Handy“ an den Sender zurückgeschickt und eine Ausgabe auf der Konsole des ParaNut getätigt. Sollte kein „Hallo“ erhalten worden sein, wird dem Sender ein „Wer bist du“ zurückgegeben und die erhaltene Nachricht auf der Konsole des ParaNut ausgeben. Hierbei ist zu beachten, dass die zu erhaltene Nachricht nur fünf Zeichen akzeptiert. Zwischen den einzelnen Überprüfungen werden auch auf der

Konsole des ParaNut Ausgaben bezüglich des Zustandes des Programms ausgegeben.

6.2.2 Das Demoprogramm des Demonstrators

Das Demoprogramm ist ähnlich zum in Kapitel 6.2.1 erstellten Demoprogramm der Bluetooth-API aufgebaut. Es soll eine erste Demonstration des Tierimplantats darstellen und wurde im Vergleich zum Demoprogramm um einen digitalen „Sensor“, einen Reedschalter, erweitert. Das Programm ist für die in Abbildung 6.2 erstellte Verbindung mit dem BM70 Pictail Board ausgelegt. Diese ist ähnlich wie der des Demoprogramms aufgebaut und nur um den Reedschalter an Pin 3 des PMOD Connectors JC erweitert, welcher hier als Input-Pin 3 konfiguriert wurde.

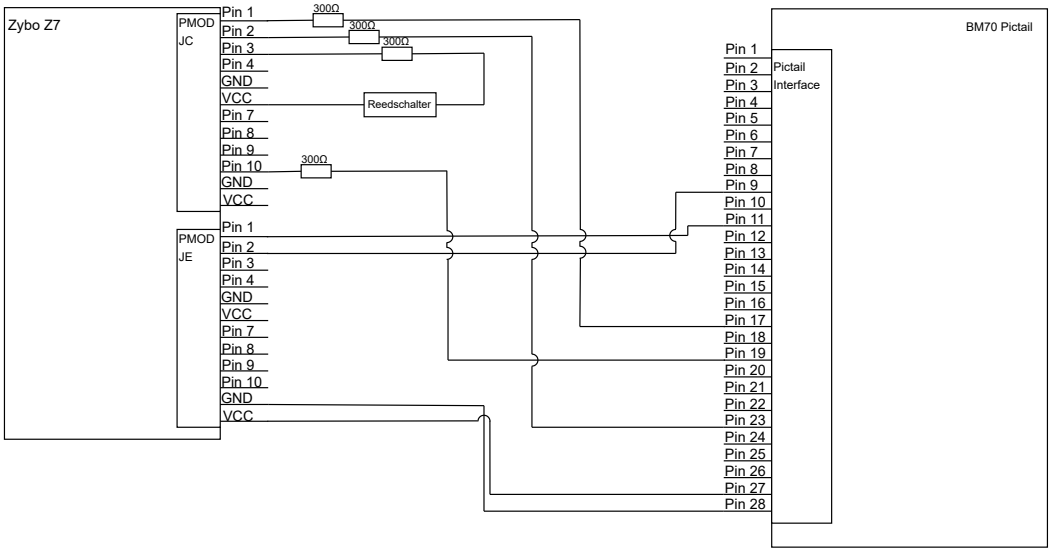

Abbildung 6.2: Verbindungen für die Tierimplantat Demo

In diesem Programm wird ebenfalls mit Hilfe der `bluetooth_wait_status`-Funktion zyklisch mit einem Zeitintervall von 20 µs überprüft, ob sich das BM70 im „Transparent Service Ready Mode“ befindet. Falls eine aktive Verbindung aufgebaut wurde, wird erst überprüft, ob sich der aktuelle Sensorzustand des Reedschalters verändert hat und wenn dies der Fall ist, wird eine Ausgabe über Bluetooth getätigt. Nach der Sensorüberprüfung wird nun aktiv mit der Funktion `bluetooth_receive_data` auf eine eingehende Nachricht gewartet, mit 20.000.000 Versuchen als Timeout-Grenze. Wird ein Kommando empfangen, wird dieses nach dem Konzept in Kapitel 4.3 verarbeitet. So wird auf ein „$S1“ mit dem aktuellen Sensorzustand und auf ein „$PI“ mit der Information, dass es sich um die Bluetooth-Implantatdemo des ParaNut handelt, geantwortet. Sollte das Kommando nicht bekannt sein, wird ein „Please send

a valid command“ an den Sender zurückgegeben. Wie auch im Demoprogramm der Bluetooth-API werden auch in diesem Programm Konsolenausgaben zum Zustand des Programms ausgegeben.

7 Ergebnisse

In diesem Kapitel werden die Ergebnisse der einzelnen durchgeführten Tests aufgeführt. Eine wichtige Frage für die Nutzung als Implantat ist der Energieverbrauch des Gesamtsystems, das zum einen die Wärmeabgabe bestimmt, zum anderen aber auch die Batterielebensdauer. Um einen Aufschluss darüber zu bekommen, wie viel Energie die einzelnen Komponenten benötigen, wurde der Energieverbrauch in mehrenen Konstellationen ermittelt.

7.1 Energieverbrauch des Bluetooth-Moduls

7.1.1 Aufbau

Komponenten

- Computer des Labors
- BM70 Pictail Board
- Voltcraft VC175 (Multimeter)
- USB-to-UART-Adapter

Zum Ausmessen des Bluetooth-Moduls wurde der Testaufbau wie in Abbildung 7.1 verwendet.

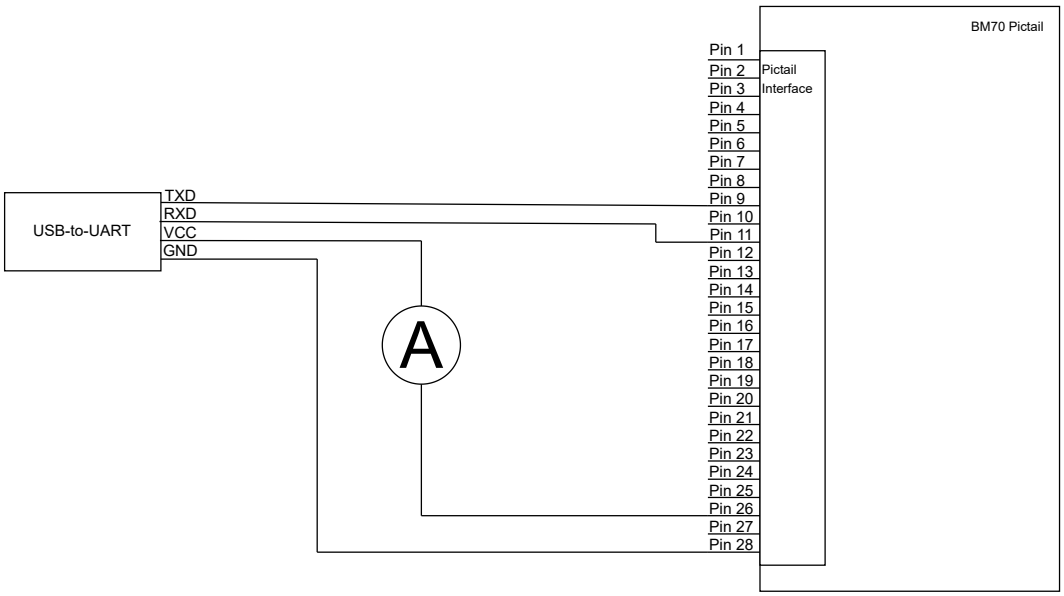

Abbildung 7.1: Verbindungen für die Energieverbrauchsmessung des Bluetooth-Moduls

Das Bluetooth-Modul wurde über den Pin 28 mit der Ground-Verbindung an einen USB-to-UART-Adapter angeschlossen. Zwischen die VCC-Leitung des BM70-Moduls und dem USB-to-UART-Adapters wurde ein Multimeter in Reihe geschalten, über welches somit der Stromverbrauch des Bluetooth-Moduls gemessen werden konnte. Die Spannung beträgt hierbei 3,3 V.

7.1.2 Durchführung

Nun wurden die verschiedenen Modi des Bluetooth-Moduls der Reihe nach gemessen. Das Bluetooth-Modul startet, wie im Zustandsdiagramms zu sehen ist, im Standby Modus. Dabei befindet sich das BM70 zuerst für ein vordefiniertes Zeitintervall in einer Fast Advertising Phase, in welchem das Modul einen Stromverbrauch von 1200 µA hat. Nach diesem Zeitintervall fällt das Modul in den Slow Advertisement Phase, in welchem es in deutliche größeren Intervallen Advertisement Packages schickt. Dadurch fällt der Stromverbrauch deutlich ab und es wird ein Stromverbrauch von 60 µA gemessen. Bei einer aktiven Bluetooth-Verbindung betrug der Verbrauch 1500 µA. Dabei unterschieden sich die Messungen zwischen dem Connected Mode und dem aktiven Transparent UART Ready Mode nicht, auch das Senden von Daten machte sich nicht in den Messergebnissen erkennbar. Somit ergibt sich eine Energieaufnahme zwischen ca. 0,2 mW und 5,0 mW.

7.1.3 Ergebnis

Durch diese Messungen konnte festgestellt werden, dass der Slow Advertisement Mode der energieeffizienteste ist. Die einzelnen Parameter waren dabei auf Werkszustand des BM70 Moduls eingestellt. Durch eine Anpassung dieser kann der Connected Mode effizienter gestaltet werden.

7.2 Energieverbrauch des ParaNut

7.2.1 Aufbau

Komponenten

- Computer des Labors
- Digilent Zybo Z7 Board
- BM70 Pictail Board
- **uart_test**-Programm des ParaNut
- **example_gpio**-Programm des ParaNut
- **example_bluetooth**-Programm des ParaNut
- USB 3.0 Color Display Tester Model: AT35

Um den Verbrauch des ParaNut-Prozessors auf Hardware grob abschätzen zu können, wurde mit Hilfe eines USB-Messgeräts die Leistung-, Spannungs- und Stromaufnahme bei verschiedenen Zuständen eines Zybo Z7 Boards erfasst. Das Digilent Zybo Z7 Board wurde an einen „USB 3.0 Color Display Tester Model: AT35“ der Firma „Rui Deng“ angeschlossen. Später wurde das Zybo Z7 auch mit einem BM70 Pictail verbunden, siehe Abbildung 6.1.

7.2.2 Durchführung

Als Referenz wurde die Energieaufnahme bei angeschaltetem Zybo Board ohne ParaNut gemessen. Dabei ergaben sich die Werte, die in der Tabelle 7.1 in der Zeile „Ohne ParaNut“ zu finden sind.
Anschließend wurde ein ParaNut mit den Konfigurationsoptionen der **config.mk** des Anhangs A.1 gebaut, sowie folgenden Konfigurationsoptionen der **config.mk** für die Synthese auf Hardware in Anhang A.2. Dieser gebaute ParaNut wurde nun

zusammen mit verschiedenen Programmen geflasht, wobei die Werte sowie die Programme der Zeile „Mit ParaNut und hello_newlib-Programm“ bis zur Zeile „Mit ParaNut und UART-Programm“ entnommen werden können. Zuletzt wurde ein Test durchgeführt, bei dem an den PMOD-Steckern des ParaNut, wie in Abbildung 6.1, das BM70-Modul angesteckt wurde und das Demoprogramm für die Bluetooth-API ausgeführt. Hierbei befand sich das BM70-Modul in der Phase des Fast Advertising. Die erhaltene Messung kann aus der letzten Zeile der Tabelle 7.1 entnommen werden.

	Spannung in V	Strom in A	Leistung in W
Ohne ParaNut	5,05	0,2380	1,2
Mit ParaNut und hello_newlib-Programm	5,12	0,3021	1,550
Mit ParaNut und GPIO-Programm	5,14	0,3038	1,55
Mit ParaNut und UART-Programm	5,17	0,2995	1,551
Mit ParaNut und Bluetooth-Programm + BM70 Modul	5,16	0,3025	1.56

Tabelle 7.1: Messergebnisse der Messung am Zybo Z7 Board

7.2.3 Ergebnis

In den Messungen konnte ermittelt werden, dass das Zybo Board durch einen geflashten ParaNut mindestens eine zusätzliche Leistung von 0,35 W benötigt, was grob die Leistungsaufnahme des ParaNuts darstellt. Hierbei handelt es sich trotzdem um eine Worst-Case-Betrachtung, da wahrscheinlich auf dem Zybo Z7 ohne geflashte Firmware einige Logikelemente abgeschalten sind und den Verbrauch weiter senken. Weiterhin kann beobachtet werden, dass das ausgeführte Programm nur einen minimalen Anstieg der Leistung verursacht und selbst mit angeschlossenem Bluetooth-Modul in einer Fast Advertising Phase nur eine Leistungsaufnahme von 1,56 W benötigt.

7.3 Test des Bluetooth-Moduls mit dem Demoprogramm der Bluetooth-API

7.3.1 Versuchsaufbau

Komponenten

- Computer des Labors
- Digilent Zybo Z7 Board
- BM70 Pictail Board

- **example_bluetooth**-Programm des ParaNut
- 300 Ω Widerstände
- App: Microchip Bluetooth Data auf Android
- Handy mit dem Betriebssystem Android

Wie in Abbildung 6.1 wurde das BM70-Modul mit dem Zybo Z7 Board verbunden. Hierbei ist zu beachten, dass der Jumper J1 des Pictail Boards auf die Option Pictail eingesteckt werden muss. Dies sorgt für eine Stromversorgung des BM70-Moduls über das Pictail Interface.

7.3.2 Durchführung

Zur Durchführung des Tests wurde ein ParaNut mit den Konfigurationsoptionen in den beiden Konfigurationsfiles in Kapitel A.1 und A.2 gebaut. Es muss darauf geachtet werden, dass das Referenzdesign des ParaNut dabei mit der Makefile Option **UART_EXT=1** gebaut wird. Darauffolgend wird das **example_bluetooth**-Programm des ParaNuts geflasht. Nach dem erfolgreichen Flashen werden die Statusmeldungen des Programms auf der Konsole ausgegeben und mit Hilfe der Microchip Bluetooth Data App kann eine Verbindung aufgebaut werden. In dieser wird nun der Microchip Transparent UART Service gestartet und mit Hilfe der Konsole können Nachrichten an den ParaNut übertragen werden. In der App sollte die Option „Append CR/LF Flag“ angeschaltet werden. Nun wurden mehrere Nachrichten an den ParaNut gesendet. Die erste Nachricht ist ein „Hallo“, auf welches als Antwort ein „Hallo Handy“ zurückkommt. Als Nächstes wurde die Nachricht „test“ gesendet, auf welche die Nachricht „Wer bist du?“ erfolgt. Als letzter Test wurde als Nachricht „testtest“ gesendet. Die Antwort lautete „Wer bist du? Wer bist du?“. Nun wurde die Verbindung des Smartphones über Bluetooth beendet und für 5 min gewartet, um zu überprüfen, ob das BM70-Modul in den Shutdown Mode verfällt. Nach diesen 5 min wurde das Mobiltelefon wieder mit dem BM-70 verbunden.

7.3.3 Ergebnis

Durch den Test ergab sich, dass die Kommunikation vom ParaNut zum BM70-Modul mit Hilfe der Bluetooth-API möglich ist und wie ausgelegt funktioniert. So fällt das BM70 auch nach längerem Warten nicht in den Shutdown Mode und die Kommunikation mit diesem funktioniert auch nach dem Warten noch wie erwartet.

7.4 Test des Bluetooth-Moduls mit dem Demoprogramm des Demonstrators

7.4.1 Versuchsaufbau

Komponenten

- Computer des Labors
- Digilent Zybo Z7 Board
- BM70 Pictail Board
- **bluetooth_implant**-Programm des ParaNut
- 300 Ω Widerstände
- App: Microchip Bluetooth Data auf Android
- Handy mit dem Betriebssystem Android
- Reedschalter

Wie in Abbildung 6.2 wurde das BM70-Modul mit dem Zybo Z7 Board verbunden. Hierbei ist zu beachten, dass der Jumper J1 des Pictail Boards auf die Option Pictail eingesteckt werden muss. Dies sorgt für eine Stromversorgung des BM70-Moduls über das Pictail Interface. Der Reedschalter war zu Beginn in einer offenen Position. Der Aufbau des Tests in Hardware ist in Abbildung 7.2 gezeigt.

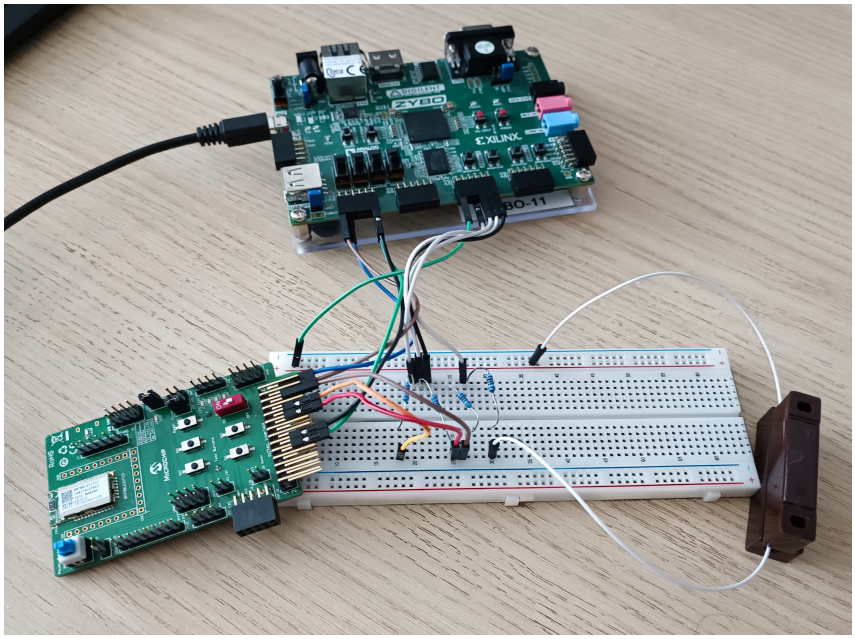

Abbildung 7.2: Aufbau des Demonstrators in Hardware

7.4.2 Durchführung

Für diesen Test wurde ein ParaNut gebaut, wie es in Kapitel 7.3.2 beschrieben wurde. Anschließend wurde das Programm **bluetooth_implant** des ParaNut auf den ParaNut geflasht und eine Verbindung per Smartphone aufgebaut, wie in Kapitel 7.3.2 beschrieben. Nun wurden folgende Kommandos über die App geschickt: „$S1“ „$PI“ und ein „$test“. Auf das Kommando „$S1“ kam die Nachricht „Sensor Value: 0“ zurück. Bei dem Befehl „$PI“ wurde als Antwort „This is the ParaNut Implant Demo“ erhalten. Als Antwort auf die Nachricht „$test“ wurde die Nachricht „Please send a valid command“ empfangen. Nun wurde der Reedschalter geschlossen, worauf nach einer Verzögerung „Sensor value updated: 1“ erhalten wurde. Das erneute Ausführen des „$S1“-Befehls lieferte ein „Sensor Value: 1“ zurück.

7.4.3 Ergebnis

In diesem Test wurde das vollständige Konzept der Anbindung an den ParaNut nach Kapitel 4 getestet. Auf die gesendeten Kommandos wurden die zu erwartenden Antworten erhalten. Auch die Reaktion auf eine Änderung des Sensorwerts findet statt.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

In der vorliegenden Arbeit wurde ein Konzept für einen ersten Demonstrator für intelligente Tierimplantate erarbeitet. Die benötigten Schnittstellenmodule wurden entwickelt und an den ParaNut per Whisbone-Bus angeschlossen, so dass der ParaNut-Prozessor nun sowohl über eine konfigurierbare UART-Schnittstelle als auch über eine konfigurierbare GPIO-Schnittstelle verfügt.

Für die beiden Schnittstellen wurden jeweils APIs entwickelt, die es den künftigen Entwicklern erleichtern die Schnittstellen zu verwenden. Durch diese Schnittstellen konnte erfolgreich eine Bluetooth-API entwickelt werden, die es ermöglicht mit Hilfe eines BM70-Bluetooth-Moduls eine Kommunikation vom ParaNut mit mobilen Endgeräten aufzunehmen.

Auch ein erster Prototyp für das Auslesen eines Sensors und die Übermittlung zu einem mobilem Endgerät per Bluetooth wurde implementiert. Dieser erlaubt über den Transparent UART Service Werte eines Sensors auszulesen oder über Änderungen informiert zu werden. Die Funktionsfähigkeit wurde im ersten Prototypen durch einen Reedschalter gezeigt.

8.2 Ausblick

Im Folgenden soll nun ein Ausblick vermittelt und mögliche nächste Schritte skizziert werden.

Zunächst ist die Anbindung von analogen Sensoren erforderlich. Dazu kann möglicherweise der ADC des Bluetooth-Moduls BM70 bereits eingesetzt werden. Dafür, aber auch zur Reduzierung der Energieaufnahme ist es erforderlich, den BM70 im Manual Mode zu betreiben. Die Bluetooth-API sollte dafür um die Funktionalitäten des Manual Mode erweitert werden. Alternativ könnte auch ein externer ADC über die GPIO-Schnittstelle dieser Arbeit angesprochen werden. Zusammen mit dem Einsatz des ADC ist damit eine Anbindung beliebiger analoger Sensoren an den ParaNut möglich (z.B. Temperatur-, Druck- oder auch Glukosesensoren).

Sinnvoll für den Einsatzzweck als Implantat erscheint auch ein Low-Energy-Mode des ParaNut selbst. Dieser könnte zum Beispiel die Hardwarefunktionalität des ParaNut-Prozessors reduzieren und damit in Phasen, in denen derzeit aktiv gewartet wird, die Stromaufnahme minimieren.

Alternativ ist für eine einsatzfähige Version eines Tierimplantats eine Umsetzung des ParaNut als Anwendungsspezifische integrierte Schaltung (ASIC) denkbar, im Gegensatz zur derzeitigen Implementierung auf einem FPGA-Board, wie dem Zybo Z7. Dies erscheint insbesondere zur Reduzierung der Größe, aber auch zur Reduzierung der Betriebsspannung und damit der Energieaufnahme sinnvoll.

Literaturverzeichnis

[1] Aglietti, Federico: *wb_uart*. https://opencores.org/projects/wb_uart. Version: Februar 2010. – Abgerufen am 09.03.2023 5.1.1

[2] Bahle, Alexander ; Kiefer, Gundolf ; Pfützner, Anna-Kerstin ; Vollbracht, Lutz: *The ParaNut/RISC-V Processor - An Open, Parallel,and Highly Scalable Processor Architecture for FPGA-based Systems*. Embedded World(2020). https://ees.hs-augsburg.de/paranut/paranut-paper-ew2020.pdf. – Abgerufen am 03.03.2023 2.1, 2.1

[3] Bauer, Lukas: *Weiterentwicklung der Debug-Infrastruktur des ParaNut-Prozessors*. Februar 2023. – Bachelorarbeit an der Hochschule Augsburg 5.1, 5.1.2, 5.1.4, 5.1.6, 5.1.7, 5.1.7, 5.1.8

[4] Begus, Romina ; Uebel, Lukas: *Präsentation: Das Geschäftsmodell*. 2022. – Projektarbeit an der Hochschule Augsburg, IMS, unveröffentliche Quelle 4.2

[5] Bluetooth SIG: *Bluetooth Specification Version 5.0 | Vol 0*. https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043. Version: Dezember 2016. – S. 253 Abgerufen am 06.03 2.4, 2.4.1

[6] Bundesministerium für Gesundheit: *Diabetes mellitus Typ 1 und Typ 2*. https://www.bundesgesundheitsministerium.de/themen/praevention/gesundheitsgefahren/diabetes.html. Version: März 2023. – Abgerufen am 09.03.2023 1.1

[7] Catchpole, B. ; Kennedy, L. J. ; Davison, L. J. ; Ollier, W. E. R.: Canine diabetes mellitus: from phenotype to genotype. In: *Journal of Small Animal Practice* (2007), jul. http://dx.doi.org/10.1111/j.1748-5827.2007.00398.x. – DOI 10.1111/j.1748–5827.2007.00398.x 1.1

[8] Corradini, Sara ; Fracassi, Federico: Caniner Diabetes mellitus. In: *Veterinary Focus* 27 (2017), Juni, Nr. 2, 27-34. https://vetfocus.royalcanin.com/de/-/media/vet-focus/german-pdfs/veterinary-focus--2017--272de.pdf?rev=4ce34f39f77541be84d73a852eab0626. – ISSN 2430–7904. – aus Veterinary Focus – Vol no.-2017 abgerufen am 13.03.2023 3.2

[9] Gisselquist, Dan: *Another Wishbone (or even AXI-lite) Controlled UART*. https://github.com/ZipCPU/wbuart32. Version: November 2022. – Abgerufen am 09.03.2023 5.1.1

[10] Gomez, Carles ; Oller, Joaquim ; Paradells, Josep: Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology. In: *Sensors* 12 (2012), aug, Nr. 9. http://dx.doi.org/10.3390/s120911734. – DOI 10.3390/s120911734 2.4

[11] Kersting, Thomas ; Huster, Stefan ; Hoffmann, Anja: *Erstattung von Real-Time-Messgeräten (rtCGM) für Patienten mit insulinpflichtigem Diabetes mellitus*. https://www.iges.com/e6/e1621/e10211/e22175/e22908/e22914/e22916/attr_objs22918/IGES_Eversense_Gutachten_20180513_ger.pdf. Version: Mai 2018. – Abgerufen am 13.03.2023 3.1

[12] Kiefer, Gundolf ; Bahle, Alexander ; Meyer, Christian H. ; Wagner, Felix ; Borgsmüller, Nico: *The ParaNut Processor Architecture Description and Reference Manual*. https://ti-build.informatik.hs-augsburg.de:8443/paranut_developers/paranut/-/blob/develop/doc/paranut-manual.pdf. Version: Februar 2023. – v1.0-114-g23393c6b* Abgerufen am 15.03.2023 2.1

[13] Kröger, Jens ; Kulzer, Bernhard: Neue Formen des Glukosemonitorings und die Auswirkungen auf Therapie und Schulung in Deutschland. In: *Deutscher Gesundheitsbericht Diabetes 2021* (2020), November, 173-183. https://www.ddg.info/fileadmin/user_upload/06_Gesundheitspolitik/03_Veroeffentlichungen/05_Gesundheitsbericht/20201107_Gesundheitsbericht2021.pdf. – ISSN 1614–824X. – Abgerufen am 13.03.2023 3.1

[14] Microchip Technology Inc.: *BM70 PICtail/PICtail Plus Evaluation Board (EVB) User's Guide*, Januar 2018. https://ww1.microchip.com/downloads/en/DeviceDoc/70005235D.pdf. – ISBN: 978-1-5224-2605-9 Abgerufen am 07.03.2023 2.5.4

[15] Microchip Technology Inc.: *BM70 MCU Interface*. https://microchipdeveloper.com/ble:bm70-mcu-interface. Version: 2021. – Abgerufen am 01.03.2023 2.5.2.4

[16] Microchip Technology Inc.: *BM70/71 Bluetooth Low Energy (BLE) Module*. DS60001372K. Microchip, 2021. https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/Bluetooth-Low-Energy-BLE-Module-DS60001372K.pdf. – ISBN: 978-1-5224-6823-3 2.5.1

[**17**] Microchip Technology Inc.: *BM70/71 Bluetooth Low Energy Module User's Guide*. DS50002542B. Microchip, 2021. https://ww1.microchip.com/downloads/en/DeviceDoc/BM70-71-Bluetooth-Low-Energy-Module-Users-Guide-DS50002542.pdf. – ISBN: 978-1-5224-6824-0 2.5.1, 2.5.2, 2.2, 2.5.2.1, 2.5.2.2, 2.5.2.3, 2.5.2.4, 2.1, 2.5.2.4

[**18**] Microchip Technology Inc.: *Microchip Transparent UART Service for BM70/RN4870*. https://microchipdeveloper.com/wireless:ble-mchp-transparent-uart-service. Version: 2021. – Abgerufen am 03.03.2023 2.5.3

[**19**] Milencovic, Marco: *Synthese von SystemC Code mit Open-Source-Tools*. Februar 2023. – Bachelorarbeit an der Hochschule Augsburg 5.1.6

[**20**] Rohde&Schwarz: *UART verstehen*. https://www.rohde-schwarz.com/de/produkte/messtechnik/essentials-test-equipment/digital-oscilloscopes/uart-verstehen_254524.html. Version: 2023. – Abgerufen am 20.02.2023 2.2

[**21**] Texas Instruments: *Asynchronous Communications Element With 64-Byte FIFOs And AutoFlow Control datasheet (Rev. C)*. https://www.ti.com/lit/ds/symlink/tl16c750.pdf?ts=1677765101633. Version: Dezember 1997. – Aberufen am 03.03.2023 2.3, 5.1.3

[**22**] Tosi, Jacopo ; Taffoni, Fabrizio ; Santacatterina, Marco ; Sannino, Roberto ; Formica, Domenico: Performance Evaluation of Bluetooth Low Energy: A Systematic Review. In: *Sensors* 17 (2017), dec, Nr. 12, S. 2898. http://dx.doi.org/10.3390/s17122898. – DOI 10.3390/s17122898 2.4.1, 2.4.2, 2.4.3

[**23**] Webseite des ParaNut-Projekts: *Der ParaNut-Prozessor*. https://ees.hs-augsburg.de/paranut/index.html. Version: Januar 2023. – Abgerufen am 03.01.2023 2.1

[**24**] Witt, Sebastian: *Stimulus-File der Testbenches*. https://raw.githubusercontent.com/freecores/uart16750/master/sim/rtl_sim/src/uart_stim.dat. Version: August 2018. – Abgerufen am 09.03.2023 5.1.5

Ich, Elias Schuler, Matrikel-Nr. 2030805, versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

> *Entwurf eines Demonstrators für intelligente Tier-Implantate auf Basis eines ParaNut/RISC-V-Prozessors*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Dasing, den 19. März 2023

Elias Schuler

A Anhang

A.1 Verwendete config.mk des ParaNut

Listing A.1: Verwendete config.mk des ParaNut

```
1  #  ------------------------------------------------------------------------------
2  #
3  #  This file is part of the ParaNut project.
4  #
5  #  Copyright (C) 2022-2023 Elias Schuler <elias.schuler@hs-augsburg.de>
6  #                2022-2023 Lukas Bauer <lukas.bauer@hs-ausgsburg.de>
7  #            2019-2021 Alexander Bahle <alexander.bahle@hs-augsburg.de>
8  #                              Michael Schaeferling <michael.
       schaeferling@hs-augsburg.de>
9  #                              Christian Meyer <christian.meyer@hs-augsburg
       .de>
10 #       Hochschule Augsburg, University of Applied Sciences
11 #
12 #  Description:
13 #    This file contains the global configuration options for the
       ParaNut.
14 #
15 #  ---------------------- LICENSE
       ---------------------------------------------
16 #  Redistribution and use in source and binary forms, with or without
       modification,
17 #  are permitted provided that the following conditions are met:
18 #
19 #  1. Redistributions of source code must retain the above copyright
       notice, this
20 #     list of conditions and the following disclaimer.
21 #
22 #  2. Redistributions in binary form must reproduce the above copyright
        notice,
23 #     this list of conditions and the following disclaimer in the
       documentation and/or
24 #     other materials provided with the distribution.
25 #
```

```
26 #  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
       "AS IS" AND
27 #  ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
       THE IMPLIED
28 #  WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
       ARE
29 #  DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS
       BE LIABLE FOR
30 #  ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
       CONSEQUENTIAL DAMAGES
31 #  (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
       SERVICES;
32 #  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
       CAUSED AND ON
33 #  ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
       TORT
34 #  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
       USE OF THIS
35 #  SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
36 #
       ------------------------------------------------------------------------

37
38
39 # Unit factors (do not edit!) ...
40 KB = 1024
41 MB = 1024*1024
42
43
44
45 # timebase in us.
46 # Defines the mtime timebase in us
47 CFG_NUT_MTIMER_TIMEBASE_US  ?=  1000
48
49 # mtimer base address.
50 # Defines the address at which the mtimer will be added to the system
       interconnect
51 CFG_NUT_MTIMER_ADDR      ?=  0x80000000
52
53 # Simulation memory address.
54 # Defines the start address (reset address) of the ParaNut during
       simulation and
55 # the address at which the main memory will be added to the system
       interconnect.
56 # Also used to determine the cacheable memory addresses.
57 CFG_NUT_RESET_ADDR    ?=  0x10000000
58
59
```

```
60 # ********************** General options
       ************************************
61
62 # Number of cores overall as log2.
63 # Defines the log2 of the overall number of ParaNut cores (ExUs).
64 CFG_NUT_CPU_CORES_LD ?= 2
65
66 # Number of cores (ExUs) with mode capability = 1 (linked).
67 # Defines the number of ParaNut cores (ExUs) that have a mode
       capability of 1 and thus can only
68 # operate in mode 1 (linked mode).
69 CFG_NUT_CPU_CAP1_CORES ?= 0
70
71 # System memory size.
72 # Defines the system memory size in Bytes. 8MB is recommended for
       simulation, otherwise
73 # needs to match the memory size of the used board.
74 # Also used to determine the cacheable memory addresses.
75 CFG_NUT_MEM_SIZE ?= (256 * MB)
76
77 # Number of external interrupt lines.
78 # Defines the number of external interrupt lines. Needs to be at least
       equal to 1.
79 CFG_NUT_EX_INT ?= 2
80
81
82 # ********************** SystemC simulation options
       ***************************
83
84 # Simulation clock speed in Hz.
85 # Defines the simulation clock speed in Hz. The configured value can be
        read from the
86 # pnclockinfo CSR.
87 CFG_NUT_SIM_CLK_SPEED ?= 25000000
88
89 # Simulation maximum peripherals number.
90 # Defines the maximum number of peripherals that can be connected to
       the systems Wishbone
91 # interconnect.
92 CFG_NUT_SIM_MAX_PERIPHERY ?= 5
93
94
95
96 # ********************** EXU (Execution Unit)
       *********************************
97
98 # RISC-V ISA Extensions
99 # Defines if the RISC-V Extension hardware is enabled/disabled.
100 # 0 - extension disabled
```

```
101 # 1 - extension enabled
102 CFG_EXU_M_EXTENSION ?= 1
103 CFG_EXU_A_EXTENSION ?= 1
104
105 # Privilege Mode options.
106 # Defines the RISC-V privilege mode capability of the ParaNut. Do NOT
      use any other value!
107 # 1 - only M-Mode available
108 # 2 - M- and U-Mode available
109 # 3 - M-, S- and U-Mode available, enable the Memory Management Unit (
      MMU)
110 CFG_PRIV_LEVELS ?= 1
111
112 # Performance counter enable.
113 # Defines if the hardware performance counters are enabled/disabled.
114 # 0 - no performance counters
115 # 1 - performance counter and 64bit cycle counter is added
116 CFG_EXU_PERFCOUNT_ENABLE ?= 1
117
118 # Performance counter register width.
119 # Defines the number of bits for the hardware performance counters.
120 # Warning: Has to be between 33 and 64.
121 CFG_EXU_PERFCOUNTER_BITS ?= 40
122
123 # Performance counter number of registers as log2.
124 # Defines the log2 of the number of hardware performance counters.
125 # Warning: 3 is the only supported value for now
126 CFG_EXU_PERFCOUNTERS_LD ?= 3
127
128
129
130 # ********************** MemU (Memory Unit)
      **********************************
131
132 # Number of cache banks as log2.
133 # Defines the log2 of the number of cache banks.
134 # A cache line has a size of 2^CFG_MEMU_CACHE_BANKS_LD words. A good
      starting point is 2 (4 banks).
135 CFG_MEMU_CACHE_BANKS_LD ?= 2
136
137 # Number of cache sets as log2.
138 # Defines the log2 of the number of cache sets.
139 # A bank has 2^CFG_MEMU_CACHE_SETS_LD * 2^CFG_MEMU_CACHE_WAYS_LD words.
140 CFG_MEMU_CACHE_SETS_LD ?= 9
141
142 # Number of cache ways as log2.
143 # Defines the log2 of the number of cache ways (cache associativity).
144 # A bank has 2^CFG_MEMU_CACHE_SETS_LD * 2^CFG_MEMU_CACHE_WAYS_LD words.
145 # 0 - 1-way set-associativity
```

```
146 # 1 - 2-way set-associativity
147 # 2 - 4-way set-associativity
148 CFG_MEMU_CACHE_WAYS_LD ?= 2
149
150 # Cache replacement method.
151 # Defines the cache replacement method/strategy, either pseudo-random
      or least recently used (LRU).
152 # 0 - random replacement
153 # 1 - LRU replacement
154 CFG_MEMU_CACHE_REPLACE_LRU ?= 1
155
156 # Arbiter Method.
157 # Defines the MemU arbitration method/strategy, either a round-robin or
       pseudo-random arbitration.
158 # >0 - round-robin arbitration, switches every (1 <<
      CFG_MEMU_ARBITER_METHOD) clocks
159 # <0 - pseudo-random arbitration (LFSR-based)
160 CFG_MEMU_ARBITER_METHOD ?= 7
161
162 # Busif Data Width.
163 # Defines the width of the master Wishbone data bus.
164 # 32 - 32 Bit data width
165 # 64 - 64 Bit data width
166 CFG_MEMU_BUSIF_WIDTH ?= 32
167
168
169
170 # ******************** MMU (Memory Management Unit)
      ****************************
171
172 # Enable or disable the Translation Lookaside Buffer (TLB)
173 # 1 - TLB enabled
174 CFG_MMU_TLB_ENABLE ?= 0
175
176 # Number of TLB entries as log2.
177 # Defines the log2 of the number of TLB entries.
178 # Must be at least 1
179 CFG_MMU_TLB_ENTRIES_LD ?= 2
180
181
182
183 # ********************** IFU (Instruction Fetch Unit)
      ************************
184
185 # Instruction buffer size as log2.
186 # Defines the log2 of the number of instruction buffer elements (max.
      number of prefetched
187 # instructions).
188 CFG_IFU_IBUF_SIZE_LD ?= 2
```

```
189
190
191
192 # ********************** LSU (Load-Store Unit)
        *******************************
193
194 # LSU write buffer size as log2.
195 # Defines the log2 of the number of write buffer elements in the LSU.
196 CFG_LSU_WBUF_SIZE_LD ?= 2
197
198
199 # ***************************** UART Module
        ***********************************
200 # The UART module is only avalabail in a synthesiesed ParaNut System
201
202
203 # Enables the Uart hardware module to be synthesized
204 CFG_UART_ENABLE ?= true
205
206 # Sets the base address of the UART module on the Wishbone bus (Default
        0x60000000)
207 # Where the UART module is connected to the systems interconnect.
208 # To update this setting correctly the update_uart traget needs to be
        executed
209 # This requires the ICSC HLS tool and sv2v
210 # IMPORTANT: At the moment only even addresses are supported like (0
        x60000000, 0x40000000, 0xA0000000)
211 # please use one of the addresses listed above to enshure that the UART
        is working
212 # NOT SUPPORTED: Are uneven addresses like (0x70000000, 0x50000000),
        addresses like 0x61000000 are also not supported
213 CFG_UART_BASE_ADDRESS ?= 0x60000000
214
215
216 # ***************************** GPIO Module
        ***********************************
217 # Enables the GPIO hardware module to be synthesized
218 CFG_GPIO_ENABLE ?= true
219
220 # Sets the base address of the GPIO module on the Wishbone bus (Default
        0x62000000)
221 # Where the GPIO module is connected to the systems interconnect.
222 # To update this setting correctly the update_gpio traget needs to be
        executed
223 # This requires the ICSC HLS tool and sv2v
224 # IMPORTANT: At the moment only even addresses are supported like (0
        x60000000, 0x40000000, 0xA0000000)
225 # please use one of the addresses listed above to enshure that the UART
        is working
```

```
226 # NOT SUPPORTED: Are uneven addresses like (0x70000000, 0x50000000),
      addresses like 0x61000000 are also not supported
227 CFG_GPIO_BASE_ADDRESS ?= 0x62000000
228
229 # Sets up the Amount of GPIO Pins beeing enabled
230 # IMPORTANT: Currently only 8 is supported
231 # A amount of 32 can be enabled, but the constraint file for the
      hardware needs to be set up manually
232 CFG_GPIO_AMOUNT ?= 8
233
234 # Sets up the amount of GPIO Output pins a amount of 8 is currently
      supported
235 # The rest of the 8 GPIO pins are Inputs.
236 CFG_GPIO_OUTAMOUNT ?= 5
```

A.2 Verwendete config.mk für das Zybo Z7 Board

Listing A.2: Verwendete config.mk für das Zybo Z7 Board

```
1 # ------------------------------------------------------------------------------
2 #
3 #  This file is part of the ParaNut project.
4 #
5 #  Copyright (C)  2022-2023 Elias Schuler <elias.schuler@hs-augsburg.de
      >
6 #                 2022-2023 Lukas Bauer <lukas.bauer@hs-ausgsburg.de>
7 #         2019-2021 Alexander Bahle <alexander.bahle@hs-augsburg.de>
8 #                           Michael Schaeferling <michael.
      schaeferling@hs-augsburg.de>
9 #      Hochschule Augsburg, University of Applied Sciences
10 #
11 #  Description:
12 #    This file contains the global configuration options for the
      ParaNut.
13 #
14 #  ---------------------- LICENSE
      ----------------------------------------------
15 #  Redistribution and use in source and binary forms, with or without
      modification,
16 #  are permitted provided that the following conditions are met:
17 #
18 #  1. Redistributions of source code must retain the above copyright
      notice, this
19 #     list of conditions and the following disclaimer.
20 #
```

```
21 #   2. Redistributions in binary form must reproduce the above copyright
       notice,
22 #      this list of conditions and the following disclaimer in the
       documentation and/or
23 #      other materials provided with the distribution.
24 #
25 #   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
       "AS IS" AND
26 #   ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
       THE IMPLIED
27 #   WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
       ARE
28 #   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS
       BE LIABLE FOR
29 #   ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
       CONSEQUENTIAL DAMAGES
30 #   (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
       SERVICES;
31 #   LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
       CAUSED AND ON
32 #   ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
       TORT
33 #   (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
       USE OF THIS
34 #   SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
35 #
       ----------------------------------------------------------------------


36
37
38 ## Do not edit
39 KB = 1024
40 MB = 1024*1024
41
42
43 # ********** SystemC Simulation options **********
44 # Simulation clock speed in Hz.
45 # Defines the simulation clock speed in Hz. The configured value can be
       read from the
46 # pnclockinfo CSR.
47 CFG_NUT_SIM_CLK_SPEED ?= 25000000
48
49 # timebase in us.
50 # Defines the mtime timebase in us
51 CFG_NUT_MTIMER_TIMEBASE_US  ?=  1000
52
53 # mtimer base address.
54 # Defines the address at which the mtimer will be added to the system
       interconnect
```

```
55 CFG_NUT_MTIMER_ADDR      ?=  0x80000000
56
57 # Simulation memory address.
58 # Defines the start address (reset address) of the ParaNut during
       simulation and
59 # the address at which the main memory will be added to the system
       interconnect.
60 # Also used to determine the cacheable memory addresses.
61 CFG_NUT_RESET_ADDR ?= 0x10000000
62
63 # Simulation maximum peripherals number.
64 # Defines the maximum number of peripherals that can be connected to
       the systems Wishbone
65 # interconnect.
66 CFG_NUT_SIM_MAX_PERIPHERY ?= 5
67
68
69 # ********** General options **********
70 # Number of cores overall as log2.
71 # Defines the log2 of the overall number of ParaNut cores (ExUs).
72 CFG_NUT_CPU_CORES_LD ?= 1
73
74 # Number of cores (ExUs) with mode capability = 1 (linked).
75 # Defines the number of ParaNut cores (ExUs) that have a mode
       capability of 1 and thus can only
76 # operate in mode 1 (linked mode).
77 CFG_NUT_CPU_CAP1_CORES ?= 0
78
79 # System memory size.
80 # Defines the system memory size in Bytes. 8MB is recommended for
       simulation, otherwise
81 # needs to match the memory size of the used board.
82 # Also used to determine the cacheable memory addresses.
83 CFG_NUT_MEM_SIZE ?= (256 * MB)
84
85 # Number of external interrupt lines.
86 # Defines the number of external interrupt lines. Needs to be at least
       equal to 1.
87 CFG_NUT_EX_INT ?= 2
88
89
90 # ********** EXU **********
91 # RISC-V ISA Extensions
92 # Defines if the RISC-V Extension hardware is enabled/disabled.
93 # 0 - extension disabled
94 # 1 - extension enabled
95 CFG_EXU_M_EXTENSION ?= 1
96 CFG_EXU_A_EXTENSION ?= 1
97
```

```
98 # Privilege Mode options.
99 # Defines the RISC-V privilege mode capability of the ParaNut. Do NOT
       use any other value!
100 # 1 - only M-Mode is available
101 # 2 - M- and U-Mode are available
102 # 3 - M-, S- and U-Mode available, enable the Memory Management Unit (
       MMU)
103 CFG_PRIV_LEVELS ?= 1
104
105 # Performance counter enable.
106 # Defines if the hardware performance counters are enabled/disabled.
107 # 0 - no performance counters
108 # 1 - performance counter and 64bit cycle counter is added
109 CFG_EXU_PERFCOUNT_ENABLE ?= 1
110
111 # Performance counter register width.
112 # Defines the number of bits for the hardware performance counters.
113 # Warning: Has to be between 33 and 64.
114 CFG_EXU_PERFCOUNTER_BITS ?= 40
115
116 # Performance counter number of registers as log2.
117 # Defines the log2 of the number of hardware performance counters.
118 # Warning: 3 is the only supported value for now
119 CFG_EXU_PERFCOUNTERS_LD ?= 3
120
121
122 # ********************* MemU ****************************
123 # Number of cache banks as log2.
124 # Defines the log2 of the number of cache banks.
125 # A cache line has a size of 2^CFG_MEMU_CACHE_BANKS_LD words. A good
       starting point is 2 (4 banks).
126 CFG_MEMU_CACHE_BANKS_LD ?= 2
127
128 # Number of cache sets as log2.
129 # Defines the log2 of the number of cache sets.
130 # A bank has 2^CFG_MEMU_CACHE_SETS_LD * 2^CFG_MEMU_CACHE_WAYS_LD words.
131 CFG_MEMU_CACHE_SETS_LD ?= 9
132
133 # Number of cache ways as log2.
134 # Defines the log2 of the number of cache ways (cache associativity).
135 # A bank has 2^CFG_MEMU_CACHE_SETS_LD * 2^CFG_MEMU_CACHE_WAYS_LD words.
136 # 0 - 1-way set-associativity
137 # 1 - 2-way set-associativity
138 # 2 - 4-way set-associativity
139 CFG_MEMU_CACHE_WAYS_LD ?= 2
140
141 # Cache replacement method.
142 # Defines the cache replacement method/strategy, either pseudo-random
       or least recently used (LRU).
```

```
143 # 0 - random replacement
144 # 1 - LRU replacement
145 CFG_MEMU_CACHE_REPLACE_LRU ?= 1
146
147 # Arbiter Method.
148 # Defines the MemU arbitration method/strategy, either a round-robin or
      pseudo-random arbitration.
149 # >0 - round-robin arbitration, switches every (1 <<
    CFG_MEMU_ARBITER_METHOD) clocks
150 # <0 - pseudo-random arbitration (LFSR-based)
151 CFG_MEMU_ARBITER_METHOD ?= 7
152
153 # Busif Data Width.
154 # Defines the width of the master Wishbone data bus.
155 # 32 - 32 Bit data width
156 # 64 - 64 Bit data width
157 CFG_MEMU_BUSIF_WIDTH ?= 32
158
159
160 # ********************* MMU ****************************
161
162 # Enable or disable the Translation Lookaside Buffer (TLB)
163 # 1 - TLB enabled
164 CFG_MMU_TLB_ENABLE ?= 0
165
166 # Number of TLB entries as log2.
167 # Defines the log2 of the number of TLB entries.
168 # Must be at least 1
169 CFG_MMU_TLB_ENTRIES_LD ?= 2
170
171
172 # ********************* Ifu ****************************
173 # Instruction buffer size as log2.
174 # Defines the log2 of the number of instruction buffer elements (max.
      number of prefetched
175 # instructions).
176 CFG_IFU_IBUF_SIZE_LD ?= 2
177
178
179 # ********************* Lsu ****************************
180 # LSU write buffer size as log2.
181 # Defines the log2 of the number of write buffer elements in the LSU.
182 CFG_LSU_WBUF_SIZE_LD ?= 2
183
184
185 # ***************************** UART Module
      ***********************************
186 # The UART module is only avalabail in a synthesiesed ParaNut System
187
```

```
188
189 # Enables the Uart hardware module to be synthesized
190 CFG_UART_ENABLE ?= true
191
192 # Sets the base address of the UART module on the Wishbone bus (Default
        0x60000000)
193 # Where the UART module is connected to the systems interconnect.
194 # To update this setting correctly the update_uart traget needs to be
      executed
195 # This requires the ICSC HLS tool and sv2v
196 # IMPORTANT: At the moment only even addresses are supported like (0
      x60000000, 0x40000000, 0xA0000000)
197 # please use one of the addresses listed above to enshure that the UART
       is working
198 # NOT SUPPORTED: Are uneven addresses like (0x70000000, 0x50000000),
      addresses like 0x61000000 are also not supported
199 CFG_UART_BASE_ADDRESS ?= 0x60000000
200
201
202 # ****************************** GPIO Module
      **********************************
203 # Enables the GPIO hardware module to be synthesized
204 CFG_GPIO_ENABLE ?= true
205
206 # Sets the base address of the GPIO module on the Wishbone bus (Default
        0x62000000)
207 # Where the GPIO module is connected to the systems interconnect.
208 # To update this setting correctly the update_gpio traget needs to be
      executed
209 # This requires the ICSC HLS tool and sv2v
210 # IMPORTANT: At the moment only even addresses are supported like (0
      x60000000, 0x40000000, 0xA0000000)
211 # please use one of the addresses listed above to enshure that the UART
       is working
212 # NOT SUPPORTED: Are uneven addresses like (0x70000000, 0x50000000),
      addresses like 0x61000000 are also not supported
213 CFG_GPIO_BASE_ADDRESS ?= 0x62000000
214
215 # Sets up the Amount of GPIO Pins beeing enabled
216 # IMPORTANT: Currently only 8 is supported
217 # A amount of 32 can be enabled, but the constraint file for the
      hardware needs to be set up manually
218 CFG_GPIO_AMOUNT ?= 8
219
220 # Sets up the amount of GPIO Output pins a amount of 8 is currently
      supported
221 # The rest of the 8 GPIO pins are Inputs.
222 CFG_GPIO_OUTAMOUNT ?= 5
```