Integration des Imote2 in das SDL Environment Framework (SEnF)

Fächerübergreifende Projektarbeit

9. Oktober 2007

Arbeitsgruppe Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Thorsten Schmelzer

betreut durch Prof. Dr. Reinhard Gotzhein
und Dipl. Inf. Marc Krämer

Inhaltsverzeichnis

Kapitel 1

Einleitung

Drahtlose Sensornetzwerke gewinnen in der heutigen Zeit immer mehr an Bedeutung. Diese Netzwerke bestehen aus kleinen Sensorknoten, die untereinander drahtlos kommunizieren können. Jeder Knoten besitzt Sensoren, die er selbstständig auswerten und die Ergebnisse an andere Knoten oder einen Zentralrechner weitergeben kann. Die Hardwareplattform Imote2 [13] ist ein leistungsfähiger drahtloser Sensorknoten, der in dieser Arbeit programmiert werden soll. Die Programmierung von Sensorknoten erfolgt häufig in Low-Level Programmiersprachen wie Assembler oder C. Mit einer Spezifikationssprache sollen Anwendungen für Sensorknoten auf einem höheren Abstraktionsniveau beschrieben werden. Aus der mit der Spezifikationssprache erstellten Spezifikation kann mit gewissen Einschränkungen Kode generiert werden, der dann für den Sensorknoten compiliert und auf diesem zur Ausführung gebracht werden kann. Die Vorteile der Benutzung einer Spezifikationssprache sind, dass von der Hardware des Sensorknotens abstrahiert wird und dass der Benutzer meist die Möglichkeit erhält, das System in einer graphischen Repräsentation zu spezifizieren. SDL [16] ist eine Spezifikationssprache, die mit ihrer Werkzeugunterstützung und Compilern hervorragend für diese Aufgabe geeignet ist. SDL wird in der AG Vernetzte Systeme und in dieser Arbeit dazu verwendet Systeme auf Modellebene zu spezifizieren. Damit eine Abstraktion von der Hardware des Sensorknotens möglich ist, werden Treiber benötigt, die der Spezifikationssprache eine Schnittstelle anbieten, mit der die Hardware des Sensorknotens genutzt werden kann. Ein Framework enthält die Treiber für unterschiedliche Hardwareplattformen und bildet so die Hardwareabstraktion für die Spezifikationssprache. Ziel der Arbeit ist die Implementierung einiger Treiber von Kommunikationsschnittstellen der Imote2 Hardwareplattform. So soll vor allem die Nutzung der drahtlosen und einer drahtgebundenen Kommunikationsschnittstelle möglich sein. Die entwickelten Treiber werden in das in der AG Vernetzte Systeme vorhandene SDL Environment Framework (SEnF) [6] eingebunden, so dass sie eine Schnittstelle für mit SDL spezifizierte Systeme bilden.

Die Arbeit gliedert sich in folgende Kapitel: In Kapitel 2 wird die verwendete Hardwareplattform Imote2 vorgestellt. Die einzelnen Komponenten, sowie die Schnittstellen des Imote2 werden dort genau beschrieben. Kapitel 3 stellt die für die Programmierung des Imote2 notwendige Software vor und erläutert Probleme, die während der Programmierung des Imote2 aufgetreten sind. Die Lösungen dieser Probleme werden ebenfalls angesprochen. Kapitel 4 handelt von der Implementierung der einzelnen Treiber. Es werden Treiber für die Leuchtdioden, die serielle Schnittstelle und die drahtlose Schnittstelle des Imote2 vorgestellt. Es wird beschrieben, wie diese Treiber in das Framework integriert werden. In Kapitel 5 wird eine Anwendung vorgestellt, die dieses Framework und die neuen Treiber nutzt. Bei der Anwendung handelt es sich um eine Modem-Anwendung, die Daten von einem PC zu einem anderen überträgt und dafür die drahtlose Schnittstelle des Imote2 nutzt. Im letzten Kapitel wird die Arbeit kurz zusammengefasst und ein kurzer Ausblick gegeben, welche Arbeiten für eine vollständige Integration der Hardwareplattform noch offen sind.

Kapitel 2

Hardwareplattform Imote2

Die für diese Arbeit verwendete Hardwareplattform ist der von Crossbow/Intel hergestellte Imote2 (IntelMote2). Er ist der Nachfolger des Imote und gehört zur Familie der drahtlosen Sensorknoten. Abbildung 2.1 zeigt ein Bild des Imote2. Die wesentlichen Bauteile des Imote2 sind:

- *Intel PXA271*: der Mikrocontroller des Imote2 [12]
- *Chipcon CC2420*: der Transceiver Chip zum drahtlosen Senden und Empfangen [20]
- *Dialog DA9030*: die Power Management Einheit, die den Prozessor und die übrige Hardware mit der korrekten Spannung versorgt [5]

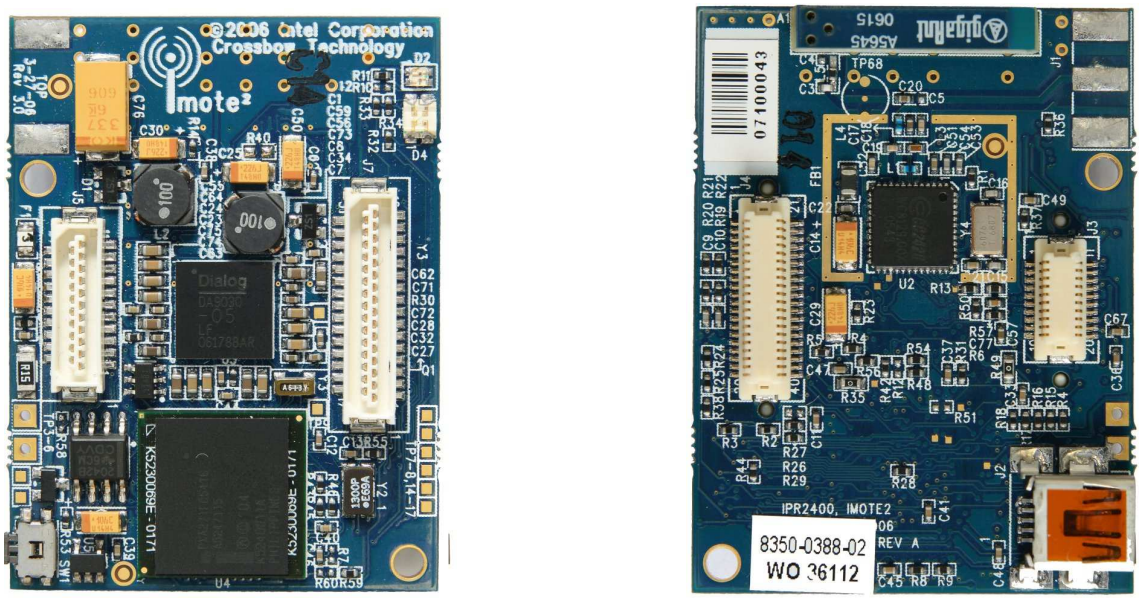

Abbildung 2.1: Die Oberseite (links) und Unterseite (rechts) der Imote2-Hardwareplattform

Daneben verfügt der Imote2 noch über drei farbige Leuchtdioden (LEDs), die vom Benutzer angesteuert werden können. Zwei mehrpolige Stecker auf der Ober- und Untersteite des Imote2 dienen als Schnittstellen. Zum einen kann der Imote2 hierüber mit Strom versorgt werden. Zum anderen werden die verschiedenen Schnittstellen des Imote2, wie serielle Schnittstellen und GPIO, nach außen geführt und ermöglichen das Anschließen von Peripheriegeräten. Es existiert beispielsweise eine Sensorplattform von Crossbow für den Imote2, es können auch eigene Sensorplatten entwickelt und angeschlossen werden. Zwei weitere Möglichkeiten, den Imote2 mit Strom zu versorgen, sind eine externe Stromquelle, die direkt an den Imote2 festgelötet wird und der MiniUSB-Anschluss,

der ebenfalls auf der Imote2-Platine integriert ist. Die Programmierung des Imote2 erfolgt entweder über den USB-Anschluss oder über ein JTAG-Kabel (siehe Abschnitt 3.1.3). Einen Überblick über die wichtigsten Bestandteile des Imote2 zeigt Abbildung 2.2. Die darauf befindlichen Komponenten werden im Folgenden genauer vorgestellt.

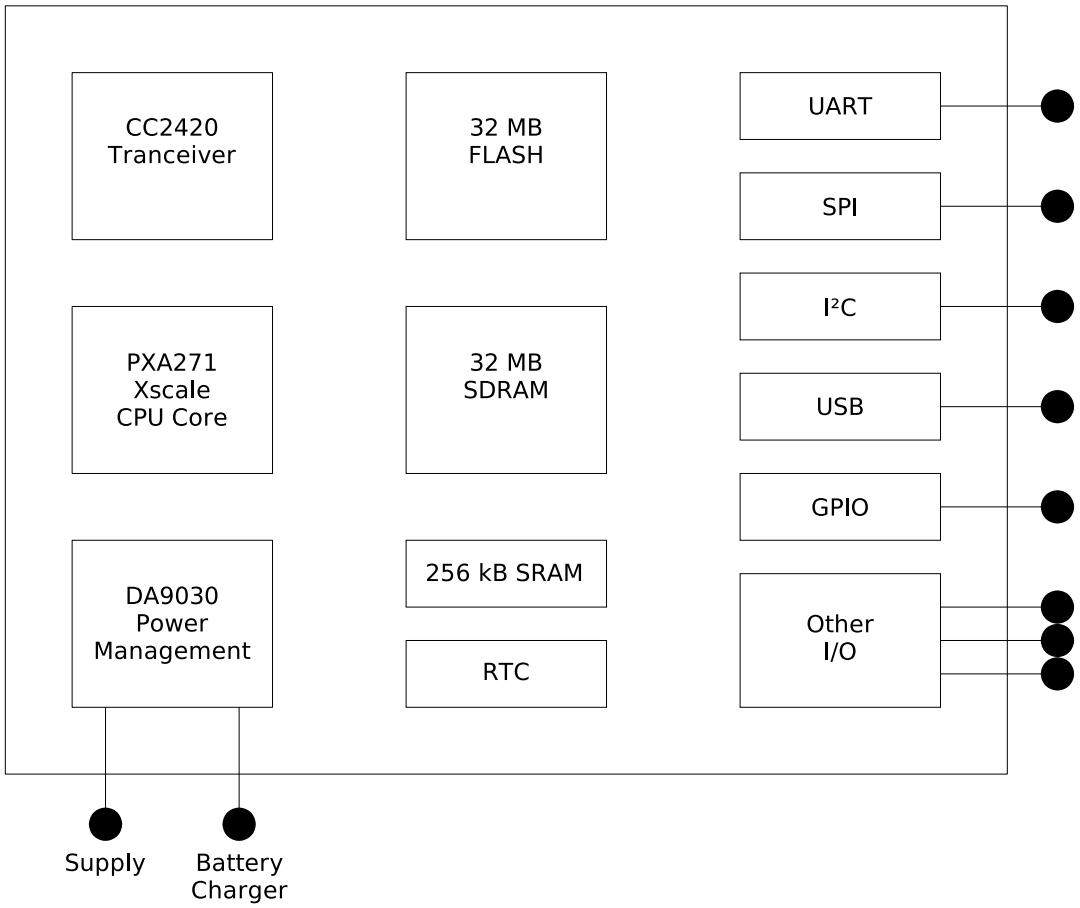

Abbildung 2.2: Die Hauptkomponenten des Imote2

2.1 Der Intel PXA271

Der PXA271 ist der Prozessor der Imote2 Hardwareplattform. Die folgenden Informationen wurden aus dem Datenblatt des Imote2 [13] und dem PXA27x Developer's Manual [12] entnommen. Der PXA271 gehört zu der Intel XScale Prozessorfamilie. XScale ist Intels Name für ihre Implementierung der ARM-Architektur. ARM steht für Advanced RISC Machine und ist eine 32 Bit RISC Architektur, die von dem englischen Computerhersteller Acorn Computers Ltd entwickelt wurde [1]. Der erste kommerzielle ARM-Prozessor wurde 1985 fertig gestellt. Intel ist einer der Lizenznehmer der ARM-Architektur und brachte im Frühling 2004 die PXA27x Prozessorreihe auf den Markt. Der PXA271 wurde für die Verwendung in mobilen Geräten, wie beispielsweise PDAs, entworfen und besitzt einige PDA typische Schnittstellen. Eine besondere Eigenschaft der PXA27x Prozessorfamilie ist, dass die Taktfrequenz des Prozessors einstellbar ist. Neben einem energiesparenden 13 MHz Modus verfügt der Prozessor noch über die Taktfrequenzen von 104 MHz, 208 MHz, 312 MHz und 416 MHz. Eine Änderung der Taktfrequenz ist während des Betriebes möglich. Daneben existieren noch verschiedene Energiesparmodi, wie Sleep und DeepSleep. Der PXA271 ist direkt mit dem RAM verbunden und besitzt auf der Imote2 Hardwareplattform folgende Speicherkapazitäten:

- 256 KByte SRAM (Static Random Access Memory), der um ein vielfaches schneller ist als der normale SDRAM des PXA271
- 32 MByte SDRAM (Synchronous Dynamic Random Access Memory)
- 32 MByte EEPROM (Electrically Erasable Programmable Read-Only Memory), der FLASH-Speicher des Imote2
- 32 KByte internen Befehlscache und 32 KByte internen Datencache [10]

Der PXA271 verfügt insgesamt über 9 Timer, sowie eine Echtzeituhr (RTC). Der erste Timer läuft immer mit einer Frequenz von 3,25 MHz und besitzt vier Matchregister. Bei jedem Inkrement des Timers wird der Wert des Zählregisters mit dem Wert des Matchregisters verglichen. Stimmen beide Werte überein, ist die eingestellte Zeit abgelaufen und es wird ein Interrupt ausgelöst. Die restlichen Timer besitzen nur je ein Matchregister und können auf unterschiedliche Zeitinkremente eingestellt werden [12].

Alle XScale-Prozessoren nutzen den ARM-v5TE Befehlssatz [10]. Dieser Befehlssatz wurde von ARM Ldt für die ARM9-Prozessorfamilie eingeführt und wird auch von der ARM10-Prozessorfamilie genutzt [2]. Der Befehlssatz hat neben den üblichen RISC Eigenschaften, wie beispielsweise die immer gleiche Wortbreite von Befehlen, noch ein paar Besonderheiten: Jeder ARM Assemblerbefehl kann mit einer Bedingung versehen werden; der Befehl wird nur dann ausgeführt, wenn die Bedingung erfüllt ist. Mit dieser bedingten Befehlsausführung kann oftmals ein Sprung vermieden werden, was die Ausführung effizienter machen soll. Außerdem besitzt die XScale-Architektur neben dem normalen Befehlssatz noch einen weiteren Befehlssatz, den THUMB-Befehlssatz. Dieser besteht im Gegensatz zur normalen Befehlsbreite von 32 Bit nur aus 16 Bit Befehlen. Somit kann gerade im eingebetteten Bereich durch diesen speziellen Befehlssatz die Kodegröße reduziert werden. Der Nachteil des Befehlssatzes ist, dass außer bedingten Sprüngen keine bedingte Ausführung von Befehlen möglich ist. Dadurch wird der Befehlsatz langsamer als der normale Befehlssatz.

2.2 Die Schnittstellen des PXA271

Zur Kommunikation mit seiner Umgebung oder anderen Mikrocontrollern besitzt der PXA271 eine große Anzahl an Schnittstellen, die wichtigsten sind [12]:

- *UART (Universal Asynchronous Receiver Transmitter)*: Diese Schnittstelle implementiert eine asynchrone serielle Verbindung nach RS232-Standard. Die maximale Teilnehmerzahl ist pro Schnittstelle auf zwei beschränkt, somit sind nur direkte Verbindungen zwischen zwei Teilnehmern möglich. Ohne Erweiterungen benötigt die Verbindung drei Leitungen (Senden, Empfangen, Erdung). Ein Takt wird für eine asynchrone Übertragung nicht übertragen. Gesteuert wird die Übertragung durch Start- und Stoppbits. Der PXA271 besitzt drei solcher UART-Schnittstellen, die eine maximale Baudrate von 921600 Baud pro Sekunde besitzen.
- *SPI (Serial Peripheral Interface)*: Das SPI-Interface ist ein serieller synchroner Bus, der nach dem Master-Slave Prinzip funktioniert. Es ist möglich, mehr als zwei Teilnehmer an diesen Bus anzuschließen. Das SPI-Interface benötigt fünf Leitungen: zwei Datenleitungen, eine Taktleitung, eine Chip-Select Leitung und eine Erdung. Mit der Chip-Select Leitung legt der Master fest, mit welchem Slave er kommunizieren will. Das SPI-Interface dient vor allem zur Verbindung von Microcontrollern untereinander. Beim PXA271 ist eine maximale Datenrate von 13 Mbps möglich. Der PXA271 besitzt drei solcher SPI-Interfaces, wovon ein SPI-Interface beim Imote2 mit dem Transceiverchip fest verbunden ist.
- *I^2C (Inter-Integrated Circuit)*: Die I^2C-Schnittstelle ist ein speziell für die Verbindung von Mikrocontrollern entwickelter serieller, synchroner Bus. Er benötigt nur drei Leitungen, eine bidirektionale Datenleitung und eine Taktleitung sowie eine Erdung. Standardmäßig werden Übertragungsraten bis 400 kbps erreicht. Der PXA271 besitzt zwei I^2C-Schnittstellen, eine wird ausschließlich genutzt, den Prozessor mit dem Power Manager zu verbinden.

- *JTAG (Joint Test Action Group Debug Interface)*: Diese Schnittstelle dient zum Debuggen des PXA271. Voraussetzung dafür ist ein geeigneter JTAG Hardware Debugger, sowie eine geeignete Software. Es ist außerdem möglich, Kode über die JTAG Schnittstelle auf den Imote2 zu laden.
- *USB (Universal Serial Bus)*: Standard USB-Schnittstelle, kann zur Stromversorgung und Laden von Kode verwendet werden. Der PXA271 kann sowohl als USB-Client eingesetzt werden, als auch selbst als USB-Host fungieren. In diesem Fall können weitere USB-Geräte an den Imote2 angeschlossen werden.
- *GPIO (General Purpose Input/Output)*: Fast jeder Pin des Prozessors kann neben seiner speziellen Aufgabe noch als Standard Ein-/Ausgabepin konfiguriert werden. Die Steuerung des Verhaltens dieser Pins obliegt dem Programmierer. So können eigene Schnittstellen implementiert werden. Ein Beispiel für die Verwendung von GPIO-Pins sind die Leuchtdioden des Imote2.
- *SDIO (Secure Digital Input/Output)*: Die SDIO-Schnittstelle bietet eine Anschlussmöglichkeit für SD-Speicherkarten. Aber auch andere Geräte, wie zum Beispiel Kameras unterstützen diese Art von Schnittstelle und können ebenfalls angeschlossen werden.
- *I^2S (Inter-IC Sound)*: Diese Schnittstelle ist speziell für die Übertragung von Audiodaten ausgelegt. Die maximale Abtastrate für die Audiodaten beträgt 48 kHz.
- *AC97 (Audio Codec 97)*: Dies ist die zweite Soundschnittstelle des PXA271, auch sie unterstützt eine maximale Abtastrate von 48 kHz. Es ist zu beachten, dass nur eine der beiden Soundschnittstellen des PXA271 gleichzeitig genutzt werden kann.
- *MSL (Mobile Scalable Link)*: Dies ist eine Hochgeschwindigkeitsverbindung, die dafür gedacht ist, zwei Prozessoren miteinander zu verbinden. Datenraten von bis zu 192 Mbps sind möglich. Der PXA271 besitzt mehrere unabhängige Datenkanäle, um mehrere gleichzeitige Verbindungen zu unterstützten.
- *Quick Capture Interface*: Diese Schnittstelle ist speziell für die Anbindung von optischen Sensoren, wie zum Beispiel Kameras, ausgelegt.

2.3 Der Chipcon CC2420

Der Chipcon CC2420 ist eine drahtlose Sende- und Empfangseinheit (Transceiver). Er operiert im 2,4 GHz Bereich und unterstützt den ZigBee (IEEE 802.15.4) Standard [20]. Der Transceiver besitzt eine maximale Datenrate von 250 kbps und eine maximale Reichweite von 30 Metern. Es ist derselbe Transceiver, den auch der MICAz verwendet [4]. Im Gegensatz zum MICAz, der eine Außenantenne besitzt, verwendet der Imote2 eine auf der Platine integrierte *gigaAnt* Antenne. Er verfügt über einen Sende- und Empfangspuffer von jeweils 128 Bytes. Pro Rahmen können bis zu 125 Byte Nutzdaten übertragen werden. Der CC2420 ist in der Lage, selbständig Prüfsummen (CRC) beim Senden zu bilden und diese beim Empfang zu überprüfen. Die Verbindung zwischen PXA271 und CC2420 erfolgt über das SPI-Interface. Daneben existieren noch vier weitere Steuerleitungen: Die SFD-Leitung zeigt an, ob der Transceiver gerade sendet. Die Leitung FIFO zeigt an, ob sich ein Byte im Empfangspuffer befindet. FIFOP zeigt an, wenn ein Rahmen komplett empfangen wurde. Die CCA (Clear Channel Assessment) Leitung zeigt an, ob der Kanal, den der Transceiver gerade benutzt, frei ist. Somit kann sichergestellt werde, dass der CC2420 nur sendet, wenn der Kanal auch frei ist (CSMA-CA).

2.4 Der Dialog DA9030

Der Dialog DA9030 ist der Power Manager der Imote2 Hardwareplattform. Er wurde speziell für die PXA27x Prozessorfamilie entwickelt [5]. Er bezieht seine Versorgungspannung entweder über

den USB-Anschluss, das Batteriepack des Imote2 oder über eine andere externe Stromquelle. Der DA9030 ist in der Lage an ihn angeschlossene aufladbare Akkus mithilfe einer externen Stromquelle wiederaufzuladen. Des weiteren versorgt der DA9030 alle Bauteile auf der Platine des Imote2 mit den entsprechenden Spannungen und stellt dem Benutzer an den Steckerverbindungen Spannungen von 1,8 V, 3 V und 5 V zur Verfügung. Eine seiner Hauptaufgaben ist die Versorgung des Prozessors mit der Kernspannung. Je nach eingestellter Taktfrequenz muss diese unterschiedlich hoch sein, um sicherzustellen, dass der PXA271 richtig funktioniert. Der DA9030 ist über das I^2C Power Interface mit dem PXA271 verbunden und kann von ihm programmiert werden.

Kapitel 3

Programmieren des Imote2

Das erste Ziel dieser Arbeit besteht darin, ein lauffähiges Programm für den Imote2 zu erzeugen. Im ersten Teil dieses Kapitels wird die dafür benötigte Software vorgestellt. Anschließend wird ein kleines Beispielprogramm implementiert, das auf dem Imote2 lauffähig ist, und sowohl die LEDs ansteuern, als auch die Prozessorfrequenz einstellen kann. Im letzten Teil geht es um Probleme, die während der Programmierung des Imote2 aufgetreten sind. Es werden sowohl Ursachen als auch Lösungen dieser Probleme besprochen.

3.1 Software

Zuerst benötigt man einen geeigneten Compiler, der in der Lage ist, Kode für die Imote2-Plattform zu erzeugen. Daneben benötigt man eine Möglichkeit, den erzeugten Kode auf die Plattform zu laden. Ebenfalls in diesem Abschnitt wird TinyOS vorgestellt, da es einige Dateien enthält, die für die Programmierung des Imote2 notwendig sind.

3.1.1 Wasabi Toolchain

Die Wasabi Toolchain ist ein *CrossCompiler* für XScale-Architekturen. Der Begriff *CrossCompiler* bringt zum Ausdruck, dass das Zielsystem, für den der Kode generiert wird (in diesem Fall der Imote2) eine andere Prozessorarchitektur besitzt, als das System, das den Kode compiliert. Die Wasabi Toolchain ist frei verfügbar und kann von der Intel Homepage heruntergeladen werden [23]. Sie basiert auf der GNU Compiler Collection in Version 3.3.1. Der Benutzer kann zwischen einem vorcompiliertem Paket und den Quellen, die er dann selbst compilieren muss, wählen. Die offiziell unterstützten Betriebsysteme sind Windows XP (mit Cygwin Umgebung) und Red Hat Linux 9 [24]. Es ist jedoch ohne Probleme möglich, die Toolchain auf dem in der AG Vernetzte Systeme verwendeten Mandriva Linux zu installieren, so dass davon ausgegangen werden kann, dass die Toolchain auch auf anderen Linux Distributionen lauffähig ist. Nach der Installation hat man einen vollständige Compiler für C/C++, inklusive eines Simulators, der es ermöglicht für XScale compilierten Kode unter Linux/Cygwin auszuführen. Zum Debuggen gibt es den GDB inklusive dem graphischen Frontend Insight. Ein Debuggen direkt auf dem Imote2 ist damit nicht möglich, lediglich ein Debuggen von im Simulator laufenden Programmen.

Die Wasabi Toolchain ist nicht das einzige *CrossCompiler*-Paket für die XScale-Architektur. Es besteht die Möglichkeit, sich den CrossCompiler aus den frei verfügbaren Quellen selbst zu compilieren [8]. Der Vorteil besteht darin, unterschiedliche Versionen des GCC nutzen zu können. Die Wasabi Toolchain nutzt eine relativ alte GCC Version, die nicht mehr weiterentwickelt wird(letzte Version der Wasabi Toolchain stammt aus dem Jahr 2004). Im Vorfeld dieser Arbeit wird GCC Version 4.1.1 für die XScale-Architektur compiliert. Einfache mit dieser Toolchain compilierte Programme ließen sich sowohl vom Simulator der selbstcompilierten Toolchain, als auch von dem der Wasabi Toolchain ausführen, ein Test auf realer Hardware blieb bis jetzt jedoch aus. Im weiteren

Verlauf des Projektes wird jedoch ausschließlich die Wasabi Toolchain genutzt. Zum einen wird keine neue GCC-Version benötigt, zum anderen erscheint es sicherer, auf die vorcompilierte Wasabi Toolchain zurückzugreifen, als sich noch eine weitere mögliche Fehlerquelle zu schaffen.

3.1.2 TinyOS

TinyOS [21] ist ein an der University of California, Berkeley entwickeltes Betriebsystem für eingebettete Sensornetzwerkknoten. Es ist frei verfügbar und unterstützt viele verschiedene Hardwareplattformen und Prozessoren, unter anderem den Imote2 sowie den MICAz. TinyOS verwendet als Programmiersprache NesC, eine Makrosprache für C. Vor dem Compilieren wird das geschriebene Programm vom NesC Transpiler zu normalem C umgewandelt. Ziel von TinyOS ist es, für unterschiedliche Hardware dem Benutzer eine einheitliche Schnittstelle zur Verfügung zu stellen. Zusätzlich erfüllt TinyOS noch typische Betriebssystemaufgaben wie Scheduling, Bereitstellen von Schutzmechanismen, Bearbeitung von Interrupts, etc. Im Gegensatz zu einem normalen Betriebssystem ist TinyOS nicht fest auf der Zielplattform installiert, sondern wird jedes mal mit der Applikation zusammen compiliert und dann auf die Zielplattform übertragen (siehe Abbildung 3.1). Letztendlich bilden TinyOS und Applikation zusammen eine neue Applikation, und innerhalb dieser neuen Applikation fungiert TinyOS als Hardwareabstraktionsschicht.

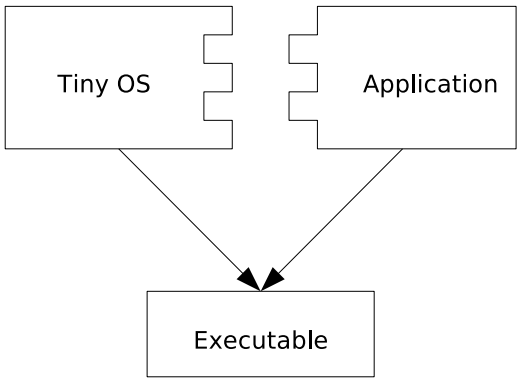

Abbildung 3.1: Nutzung von TinyOS: TinyOS und die Anwendung werden gemeinsam übersetzt und bilden eine neue Anwendung.

Für den Imote2 besitzt TinyOS viele Treiber (UART, SPI, CC2420), somit ist es möglich Programme für den Imote2 komplett in TinyOS zu schreiben. Aufgrund von Problemen bei der Integration von C-Code mittels Linker wurde bereits in der Entwicklung einer Plattform für den MICAz die Entscheidung gefällt, TinyOS nicht zu verwenden und lediglich das Wissen daraus in das schon vorhanden SDL Environment Framework [6] zu integrieren (siehe Abschnitt 4.1.4). Der große Vorteil von TinyOS sind die einheitlichen Schnittstellen, die es möglich machen, ein Programm für unterschiedliche Hardwareplattformen zu compilieren, ohne es ändern zu müssen (beide Plattformen müssen die benutzte Hardware besitzen). Dieser Vorteil kann aber auch zum Nachteil werden: da die Implemtierung auf jeder unterstützten Plattform laufen soll, wird oftmals nur der kleinste gemeinsame Nenner implementiert. Auf manchen Plattformen wird somit nicht die volle Leistungsfähigkeit ausgeschöpft. Da die Quellen von TinyOS frei verfügbar sind, können sie in dieser Arbeit als Vorlage benutzt werden. Sehr wertvoll erweisen sich die TinyOS Definitionen für Registernamen des Imote2. Sie ermöglichen einen lesbaren Kode, da Register mit ihrem Namen, statt ihrer Adresse angesprochen werden können. Die Namensgebung ist konsistent mit der Benennung in der Intel Bedienungsanleitung des PXA271 [12]. Ebenfalls übernommen werden viele in TinyOS enthaltene Assemblerdateien. Diese enthalten einfache Systemroutinen, die in Assembler programmiert sind, wie zum Beispiel das Aktivieren des Befehlscache, das zur Systeminitialisierung benötigt wird.

Zu Beginn der Arbeit wird versucht, ein einfaches TinyOS Beispielprogramm `Blink` (Blinken einer LED) zu compilieren und auf den Imote2 zu spielen. Dies schlägt fehl, aus unbekannten Gründen

kann das selbstcompilierte Programm zwar auf den Imote2 übertragen werden, aber der Knoten zeigt keinerlei Reaktion. Überspielt man ein vorcompiliertes `Blink` Programm [3], läuft der Knoten problemlos. Die von den TinyOS Entwicklern propagierte Fähigkeit, sofort nach der Installation von TinyOS einfache Beispiele compilieren und auszuführen zu können, kann in diesem Fall nicht nachgewiesen werden.

3.1.3 Bootloader und USBLoader

Die von Crossbow gelieferten Imote2 haben einen Bootloader vorinstalliert [17]. Er ermöglicht es, Daten per USB-Kabel auf den Imote2 zu übertragen. Abbildung 3.2 zeigt den FLASH-Speicher des Imote2. Der Bootloader (in der Abbildung TOS Loader) befindet sich am Beginn des Speichers, seine Address- und Partitionstabellen befinden sich im oberen Speicherbereich. Diese Speicherbereiche darf nur der Bootloader ändern, Anwendungsprogramme haben darauf keinen Zugriff. Wenn der Imote2 gestartet wird, wird immer zuerst der Bootloader aktiv. Er überprüft, ob der Imote2 über den USB-Port mit einem PC verbunden ist. Ist dies nicht der Fall, springt der Bootloader das Anwendungsprogramm, das hinter dem Bootloader im Speicher (Application and Data Section) liegt, an, und dieses wird ausgeführt. Die Programmierung des Imote2 geschieht mit Hilfe des Programms USBLoader, das ebenfalls in TinyOS enthalten ist. Wurde ein Programm erfolgreich compiliert, muss es zunächst noch ins Binärformat umgewandelt werden. Dies geschieht mit dem Befehl:

```
xscale-elf-objcopy -O binary inputfile outputfile
```

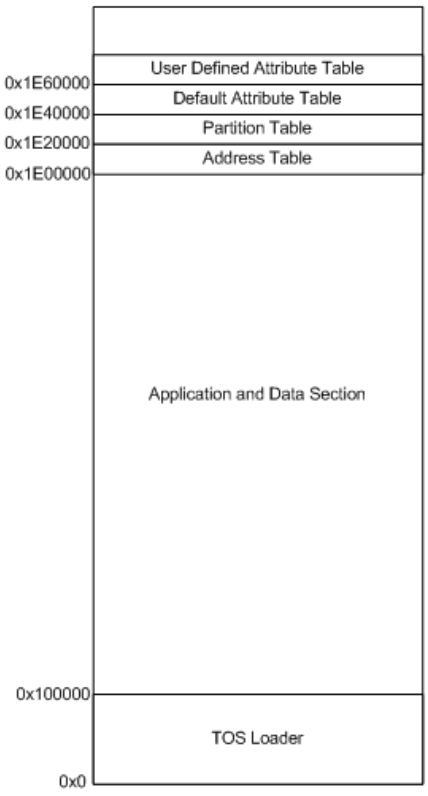

Abbildung 3.2: Der FLASH-Speicher des Imote2 mit installiertem Bootloader [17].

Danach kann die Binärdatei auf den Imote2 geladen werden. Der USBLoader wird mit dem Befehl

```
USBLoaderHost.exe -p file
```

gestartet. Hat der Bootloader des Imote2 die USB-Verbindung erkannt, startet der Uploadvorgang. Der Uploadvorgang läuft automatisch ab, eine CRC-Überprüfung des geladenen Binärprogramms findet statt. Ist der Uploadvorgang beendet, startet der Bootloader den Imote2 neu. Nach diesem Neustart wird das geladene Image verifiziert und bei erfolgreicher Verifizierung wird das Image an die Adresse 0x100000 (Beginn der Application and Data Section) kopiert, so dass es beim nächsten Neustart ausgeführt wird. Nach dem FLASH-Vorgang startet der Imote2 automatisch neu.

Der Upload über die USB-Schnittstelle ist nur mit Hilfe des Bootloaders möglich. Ist der Bootloader nicht installiert (dies kann bei anderen Herstellern des Imote2 der Fall sein) oder wird der Bootloader beschädigt, ist dies nicht möglich. Der Bootloader muss dann mit Hilfe eines JTAG-Kabels und geeigneter Flashsoftware wieder auf den Imote2 geladen werden. Eine Beschädigung des Bootloaders ist möglich, wenn das ebenfalls in TinyOS enthaltene Programm `ImoteConsole` zum Hochladen von Dateien auf den Imote2 verwendet wird [9]. Dies sollte unbedingt vermieden werden.

Ein Nachteil des USBLoaders ist, dass er momentan nur in einer Cygwin Umgebung unter Windows läuft, da er die Windows API benutzt, um auf die USB Schnittstelle zuzugreifen. Der USBLoader ist somit der einzige Grund, weswegen im Moment ausschließlich Windows (mit Cygwin) als Entwicklungsumgebung zur Verfügung steht. Die restliche benutzte Software, also die Wasabi Toolchain und TinyOS, kann auch unter Linux eingesetzt werden.

Eine Alternative zur Nutzung des Bootloaders ist die Nutzung der Flashsoftware JFlashmm [14] in Verbindung mit einem JTAG-Kabel. Damit ist es möglich, eigene Applikationen und andere Betriebssysteme, wie zum Beispiel Linux, auf den Imote2 zu laden.

3.2 Minimalbeispiel

Mit der in Abschnitt 3.1 vorgestellten Software ist es nun möglich, ein einfaches Programm für den Imote2 zu schreiben, zu compilieren und auf den Knoten zu laden. Das folgende Beispiel zeigt ein einfaches Blinken der roten LED und dient als erste Testapplikation, um die Funktionalität der Toolchain zu zeigen:

```
1  int main() {
2      initHardware();
3      long int i;
4
5      while(1) {
6          redOn();
7          for (i=0; i < 2000; i++){
8              wait();
9          }
10         redOff();
11         for (i=0; i < 2000; i++){
12             wait();
13         }
14
15     }
16     return 0;
17 }
```

Die Funktion initHardware() ist dafür verantwortlich, den Prozessortakt einzustellen und die Leuchtdioden zu initialisieren, sowie den Befehls- und den Datencache des PXA271 zu aktivieren. Wird dies nicht getan, so muss der Prozessor jeden Befehl einzeln aus dem RAM holen, was die Abarbeitung deutlich verlangsamt. Um die Blinkfrequenz der roten LED mit dem menschlichem Auge wahrnehmen zu können, muss zwischen aus- und einschalten der LED gewartet werden. Die einfachste Art zu warten, ist die Ausführung von NOP (No Operation) Befehlen. Dies tut die Funktion wait(). Mit Hilfe der LED kann man zwei Dinge feststellen. Erstens erkennt man, ob das Programm auf dem Imote2 überhaupt läuft und auch nicht nach einiger Zeit abstürzt. Zweitens kann man anhand der Blinkfrequenz feststellen, ob der PXA271 auf der eingestellten Kernfrequenz arbeitet.

3.2.1 Ansteuerung der LEDs

Die drei LEDs des Imote2 sind über GPIO-Pins mit dem Prozessor verbunden (Pin-Belegung siehe [13]). Um eine LED ein- und auszuschalten, muss man den entsprechenden Pin als GPIO-Pin und als Ausgangspin konfigurieren. Danach kann man den Pin auf logisch High bzw. Low setzen und, und so die LED ein- und ausschalten. Dies geschieht über das Schreiben von Registern. Zu beachten ist, dass die LEDs des Imote2 bei logisch High ausgeschaltet sind und bei logisch Low leuchten. Deshalb schaltet ein Setzen des entsprechenden Registers die LED aus und ein Löschen sie ein.

3.2.2 Änderung der Taktfrequenz des Prozessors

Um die Taktfrequenz des Prozessors zu ändern, müssen zwei Dinge getan werden. Erstens muss die Taktfrequenz auf den entsprechenden Wert gesetzt werden. Dazu gibt es im Prozessor ein spezielles Register. Die Taktfrequenz berechnet sich aus

$$\text{Frequenz} = 13\,\text{MHz} * \text{Multiplikator}$$

In das Register wird der entsprechende Multiplikator geschrieben [12], danach kann der Frequenzwechsel initialisiert werden. Dies geschieht durch Setzen eines Bits in einem Coprozessor-Register. Zweitens muss die Kernspannung der Frequenz angepasst werden. Höhere Frequenzen erfordern eine höhere Kernspannung, wird diese nicht geändert, kommt es zu unkontrollierten Abstürzen des Imote2. Die für die unterschiedlichen Frequenzen benötigten Kernspannungen können aus [11] entnommen werden. Um Energie zu sparen, sollte immer nur die minimal benötigte Kernspannung eingestellt werden, die für die stabile Funktion des PXA271 benötigt wird. Für die Kernspannung ist, wie schon in Abschnitt 2.4 beschrieben, nicht der PXA271 zuständig, sondern der Power Manager DA9030. Dieser ist über das Power I^2C-Interface mit dem Prozessor verbunden. Um die Kernspannung zu ändern muss ein Register des DA9030 geschrieben werden. Um die Änderung der Kernspannung bei einem Frequenzwechsel zu vereinfachen, besitzt der PXA271 einen Automatismus. Der Programmierer füllt einen Registersatz (maximal 32 Register) mit den I^2C-Befehlen, die notwendig sind, um das entsprechende Register des DA9030 zu schreiben. Der PXA271 kann so konfiguriert werden, dass diese Befehle bei einem Frequenzwechsel des Prozessorkerns automatisch an den DA9030 übertragen werden, und somit die neue Spannung eingestellt wird.

Mit Hilfe dieses Minimalbeispiels wird folgendes erreicht:

- die Funktionalität der Wasabi Toolchain kann überprüft werden
- ebenso die Funktionalität des USBLoaders
- die Funktionalität der LEDs ist sichergestellt, dies ist gerade im Hinblick auf einfache Debugmöglichkeiten von Bedeutung
- die Frequenz des Prozessors kann verändert werden, damit hat man Zugriff auf die volle Leistungsfähigkeit des PXA271

3.3 Probleme

Während der Arbeit am Imote2 traten zwei grundlegende Probleme auf, die von der verwendeten Programmiersprache C++ verursacht werden. Diese Probleme und ihre Lösung werden im folgenden Abschnitt näher erläutert.

3.3.1 Anpassung des Linkerskriptes und der Startroutine

Um dieses Problem zu verstehen, muss zunächst geklärt werden, was ein Linkerskript ist. Bei der Übersetzung von Programmen mittels des GCC, wird aus jeder Quelltextdatei in der Regel eine Objektdatei erzeugt. Diese Objektdatei benutzt als Dateiformat ELF (Executable and Linking Format). In der Objektdatei befindet sich der vom Compiler übersetzte Maschinenkode, eingeteilt in sogenannte Abschnitte (engl. Sections). Der eigentliche Kode einer Funktion befindet sich

zum Beispiel in dem .text Abschnitt, während statische, initialisierte Variablen in dem .data Abschnitt abgelegt werden. Daneben gibt es noch viele weitere Abschnittarten, für eine Übersicht siehe [18]. Ebenfalls in der Objektdatei befinden sich Einsprungpunkte, die sogenannten Symbole. Jede Funktion und jede globale Variable erhält einen Einsprungpunkt, so dass sie mit diesem Namen angesprungen werden kann. Um sich alle Symbole einer Objektdatei anzeigen zu lassen, muss man folgenden Befehl ausführen:

```
xscale-elf-nm filename
```

Möchte man sich den Inhalt einer Objektdatei anzeigen lassen, kann man den Befehl

```
oxscale-elf-objdump -Dzh filename
```

ausführen. Man bekommt dann den in Assemblerkode rückübersetzten Maschinenkode der Objektdatei, sowie die einzelnen Abschnitte und Symbole angezeigt.

Nun möchte man im Allgemeinen nicht mehrere Objektdateien als Ergebnis, sondern eine ausführbare Datei. Hierfür gibt es den Linker, er setzt die compilierten Objektdateien und wenn nötig auch externe Bibliotheksdateien zu einer ausführbaren Datei zusammen. Wie dies geschieht wird mit einer einfachen Skriptsprache im Linkerkript festgelegt [22]. Das Linkerskript legt fest, welche Abschnitte der Eingabedateien in welche Abschnitte der Ausgabedatei übernommen werden und wo im Speicher sich dieser Ausgabeabschnitt befindet. Hierzu ein kleines Beispiel. Im Linkerskript des Imote2 findet sich folgender Ausdruck:

```
.text :
{
*(.text)
} >FLASH
```

Die erste Zeile bezeichnet den Namen des Ausgabeabschnittes, also der Abschnitt in der fertig gelinkten Ausgabedatei. Die Zeile, die mit dem Stern beginnt, bedeutet, suche in allen zu linkenden Objektdateien alle .text Abschnitte und schreibe deren Inhalt in den Ausgabeabschnitt. In diesem Fall befinden sich alle einzelnen .text Abschnitte der Eingabedateien nach dem Linkvorgang in einem gemeinsamen .text Abschnitt der Ausgabedatei. Diesen Vorgang verdeutlicht Abbildung 3.3. Die letzte Zeile besagt, dass der gesamte Abschnitt in den FLASH-Speicher des Imote2 geladen wird.

Im Normalfall ist es nicht nötig, ein spezielles Linkerskript zu benutzen. Wird kein spezifisches Skript angegeben, nutzt der GCC sein Standardlinkerskript, das sich mit

```
ld -verbose
```

anzeigen lässt. Auf eingebetteten Systemen benötigt man häufig ein angepasstes Linkerskript, da bestimmte Einsprungpunkte gesetzt werden müssen, oder ein bestimmtes Speicherlayout festgelegt werden muss. In der Ausgabedatei beginnt der Kode direkt an Adresse 0, wird aber später noch vom Bootloader des Imote2 relokiert. Der Speicher des Imote2 ist unterteilt in FLASH, RAM und SRAM. Zunächst wird das Linkerskript von TinyOS für den Imote2 verwendet. Dies funktioniert für einfache Programme auch. Es zeigt sich aber, dass das Linkerskript für die Verwendung des Imote2 in dieser Arbeit angepasst werden muss. Das erste Problem besteht in der Verwendung der Programmiersprache. Für C besitzt das TinyOS Linkerskript alle erforderlichen Ausgabeabschnitte, für C++ jedoch nicht. C++ legt eine Reihe weiterer Abschnitte in seinen Objektdateien an. Für diese Abschnitte gibt es keine Ausgabeabschnitte im Linkerskript. Existiert zu einem Eingabeabschnitt kein Ausgabeabschnitt, so wird der entsprechende Eingabeabschnitt nicht verworfen, sondern der Linker versucht ihn trotzdem in die Ausgabedatei zu integrieren. Dabei kommt es im Falle der Verwendung von C++ zu Überschneidungen von Abschnitten in der Ausgabedatei und der Linkvorgang schlägt fehl. Die Lösung besteht darin, die fehlenden Abschnitte im Linkerskript zu ergänzen, um so zu vermeiden, dass es zu Überlappungen kommt. So werden Eingabeabschnitte

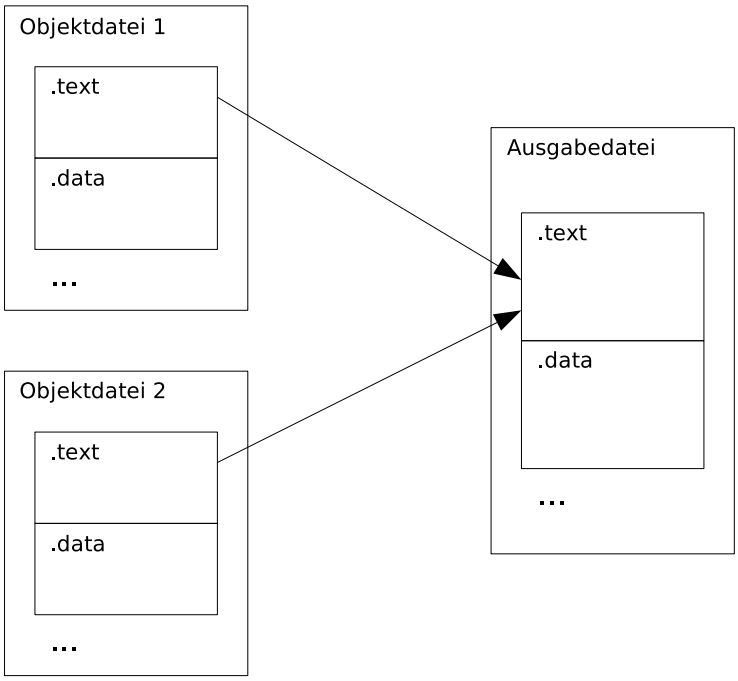

Abbildung 3.3: Linkvorgang: .text Abschnitte aus unterschiedlichen Objektdateien werden in einem .text Abschnitt zusammengelinkt.

wie beispielsweise .gnu.linkonce.r.* zum Ausgabeabschnitt .text hinzugefügt und neue Ausgabeabschnitte wie zum Beispiel .ctors und .dtors eingefügt. Nun lassen sich C++ Programme linken und auf dem Imote2 ausführen.

Das zweite Problem wird dadurch verursacht, dass keine init-Routine existiert, die bei Programmstart aufgerufen wird. Die init-Routine wird üblicherweise noch vor dem Sprung in die main-Routine aufgerufen. Sie ist unter anderem für die Initialisierung der statischen Variablen und Objekte verantwortlich. Wird sie nicht ausgeführt, werden diese nicht initialisiert, was sich in einem Nullpointer äußert, wenn auf solch eine statische Variable oder Objekt zugegriffen wird. In einer normalen Betriebssystemumgebung wird ein Nullpointerzugriff vom Betriebssystem abgefangen. Auf dem Imote2 fehlt dieser Mechanismus, da kein Betriebssystem verwendet wird. Somit greift man mit einem Nullpointer tatsächlich auf die Daten zu, die an Adresse 0 im Speicher liegen. Dies kann unvorhersehbare Folgen haben, führt meistens auch bald zum Absturz und muss somit vermieden werden.

Der Quellkode für die init-Routine befindet sich in den Bibliotheksdateien `crtbegin.o`, `crtend.o`, `crti.o` und `crtn.o`, die die Wasabi Toolchain zur Verfügung stellt. Diese Dateien enthalten unter anderem Listen mit Funktionspointern (CTOR_LIST und DTOR_LIST), die auf Konstruktoren und Destruktoren zeigen. Alle Dateien zusammen bilden die komplette init-Routine und müssen dem Linker als zusätzliche Dateien angegeben werden, damit eine init-Routine erzeugt wird. Damit die einzelnen Kodeteile der Objektdateien richtig zusammengefügt werden, muss das Linkerskript angepasst werden. Die entsprechenden Abschnitte .ctors und .dtors werden aus dem Standardlinkerskript des Wasabi GCC übernommen und der .init Abschnitt wird entsprechend angepasst. Damit die init-Routine vor der main-Routine ausgeführt wird, muss die von TinyOS übernommene Assemblerdatei `barecrt.s` angepasst werden. Die Datei enthält unter anderem die Interrupteinsprungpunkte des PXA271, sowie die start-Routine. Die start-Routine ist die erste Routine, die bei Programmstart aufgerufen wird. Sie ruft am Ende die main-Routine auf. Kurz bevor dies geschieht wird nun die init-Routine aufgerufen. Der veränderte Quellkode lautet:

```
...
bl _init
nop
bl main
```

Werden alle Veränderungen durchgeführt, so werden globale Variablen und Objekte richtig initialisiert und es kann auf sie zugegriffen werden.

3.3.2 Problem bei dynamischer Speicheranforderung in Verbindung mit Interrupts

Für die Verwendung von Objekten in C++ ist es notwendig, dynamischen Speicher anfordern zu können. Auch für die Verwendung von dynamisch angelegten Puffern ist dies nötig. Dynamischen Speicher fordert man unter C/C++ mit dem Befehl

```
malloc(size)
```

an. Das Anlegen eines neuen Objektes in C++ mit dem new-Operator wird ebenfalls auf einen malloc-Aufruf abgebildet. Der Rückgabewert von malloc ist ein Zeiger auf den neu allokierten Speicherbereich oder NULL, wenn kein Speicher allokiert werden konnte. Der Befehl

```
free(pointer)
```

gibt den allokierten Speicher, auf den der übergebene Zeiger zeigt, wieder frei. Beide Funktionen sind in den Bibliotheken der verwendeten Toolchain enthalten.

Es gibt geeignete Strategien, um dynamischen Speicher zu verwalten. Wenn diese Strategie möglichst schnell ist und den Speicher möglichst wenig fragmentiert, kann dieser optimal genutzt werden. Im GCC wird der Memory Allocator von Doug Lea verwendet [15]. Dieser organisiert Speicher in Chunks und Bins. Chunks sind Speicherbereiche mit einer bestimmten Größe. Die Minimalgröße eines Chunks beträgt 16 Byte, das bedeutet für die dynamische Speicherallokation, dass auch wenn der benötigte Speicher kleiner als 16 Byte ist, immer mindestens 16 Bytes allokiert werden. Chunks sind wiederum in Bins organisiert. Diese Bins werden dazu benutzt, freie Chunks zu verwalten. Die verwendete Speicherzuteilungsstrategie ist Best-Fit, was bedeutet, das immer derjenige Speicherbereich zugeteilt wird, der am Besten passt. Im Optimalfall ist dies ein Chunk der angeforderten Größe. Existiert kein solcher, wird ein möglichst kleiner Chunk ausgewählt und aufgeteilt.

Wenn Speicher angefordert wird, sollten die Funktionen, die Bins und Chunks manipulieren, atomar sein, ansonsten kann die Organisation der Bins und Chunks zerstört werden, wenn zum Beispiel nicht exklusiv auf Variablen zugegriffen wird. Dazu gibt es die Funktionen malloc_lock und malloc_unlock, sie werden von malloc bzw. free aufgerufen und sollen sicherstellen, dass die Funktionen nicht unterbrochen werden können. Diese Funktionen sind im Falle des Imote2 zwar in der Bibliothek vorhanden und werden auch an den richtigen Stellen aufgerufen, enthalten aber keinen Kode. Somit sind die Funktionen malloc und free unterbrechbar. Da es für den Imote2 noch keine Implementierung von Threads gibt, ist eine Unterbrechung durch Threads ausgeschlossen, nicht aber die Unterbrechung durch Interrupts. Dies führt im folgenden Fall zu Problemen: Ein malloc/free Aufruf wird durch einen Interrupt unterbrochen und innerhalb der Interruptbehandlung wird ein weiterer malloc/free Aufruf gemacht. In diesem Fall kann es zu fehlerhaften Daten oder überschriebenen Rücksprungadressen kommen kommen, wenn durch den neuen Aufruf Daten überschrieben werden, die der ursprüngliche Aufruf gerade bearbeitet. Die Folge davon ist meist ein Programmabsturz. Gerade wenn als Programmierspache C++ verwendet wird, werden häufig neue Objekte in der Interrupt-Routine angelegt und es ist wahrscheinlich, dass ein solcher Fehler auftritt, da bei jedem Anlegen eines Objekts malloc aufgerufen wird.

Die Lösung des Problems besteht darin, die Funktionen malloc_lock und malloc_unlock selbst zu implementieren und so die Standardimplementierung zu überschreiben. Die Funktion malloc_lock sperrt jetzt alle Interrupts auf dem Imote2, somit können die kritischen Abschnitte der malloc/free Funktionen nicht mehr unterbrochen werden. Die Funktion malloc_unlock gibt die Interrupts wieder frei und nun können die während der Sperrung aufgetretenen Interrupts behandelt werden. Die

hier vorgestellte Lösung führt dazu, dass Interrupts relativ häufig gesperrt werden. Werden Interrupts zu lange gesperrt, kann es sein, dass während einer Sperrung mehrere Interrupts desselben Typs auftreten. Diese mehrfachen Interrupts werden dann als ein einziger aufgetretener Interrupt erkannt. Somit gehen hier Interrupts verloren. Für diesen Fall besitzt der PXA271 für seine Kommunikationsschnittstellen große Puffer, in denen die Daten zwischengespeichert werden können (siehe Abschnitt 4.2.4, Beschreibung, was passiert wenn mehrere UART-Interrupts auftreten).

Nach der Behebung der beschrieben Probleme sind C/C++ Programme auf dem Imote2 voll lauffähig. Unter TinyOS werden diese Probleme umgangen, da TinyOS als Programmiersprache C verwendet und dynamische Speicherallokation in Interruptbehandlungen vermeidet.

Kapitel 4

Erweiterung des SEnF

Der nächste Schritt nach dem Compilieren und Ausführen von Programmen auf dem Imote2, die keine SDL-Spezifikationen enthalten, ist die Erweiterung des **S**DL **En**vironment **F**rameworks (SEnF). Im ersten Abschnitt geht es um die benutzte Software, im zweiten wird das SEnF kurz vorgestellt. Der Rest des Kapitels handelt von den Änderungen an SEnF, die gemacht werden müssen, um die Imote2 Plattform und die Treiber zu integrieren.

4.1 Software

Durch die Erweiterung des SEnF soll es möglich sein, ein System in SDL zu spezifizieren und daraus ausführbaren Kode für die Imote2 Plattform zu generieren. Dazu sind Treiber in das SEnF zu integrieren, sodass man in der Lage ist, die Kommunikationsschnittstellen des Imote2 direkt aus dem SDL-System heraus zu benutzen.

4.1.1 SDL

SDL (Specification and Descripton Language) [16] ist eine von der ITU-T (International Telecommunication Union - Telecommunication Standardization Sector) genormte Sprache, um verteilte Systeme, insbesondere Telekommunikationssysteme, zu spezifizieren. SDL ist in der International Telecommunication Union Recommendation Z.100 beschrieben und wird vor allem im Bereich der Telekommunikation eingesetzt. In SDL kommunizieren Prozesse, die durch erweiterte Zustandsautomaten spezifiziert werden, mit Hilfe von Signalen. Die Prozesse sind untereinander mit Kanälen verbunden, über die die Signale verschickt werden können. Es können ebenfalls Signale an die Umgebung geschickt werden. Als weiteren Zusatz zu normalen Zustandsautomaten gibt es in SDL Timer. Immer wenn ein Timer abläuft, schickt er ein Signal an den Prozess, zu dem er gehört, dieses kann dann weitere Aktionen auslösen. Die Gesamtheit der Prozesse und Kanäle bilden das SDL-System.

4.1.2 Telelogic Tau

Telelogic Tau [19] ist eine kommerzielles Werkzeugumgebung der Firma Telelogic, das es dem Benutzer ermöglicht, SDL-Systeme zu entwickeln. Tau stellt dem Benutzer Analysewerkzeuge zur Verfügung, mit denen es möglich ist, sein erstelltes System zu überprüfen. Die erstellte graphische Repräsentation (GR) wird dann in eine textuelle Repräsentation (PR) umgewandelt. Liegt die textuelle Repräsentation vor, kann Tau aus dieser Kode generieren. Dazu besitzt Tau zwei Transpiler: Cadvanced und Cmicro. Beide generieren aus der textuellen Beschreibung C-Kode, der dann noch für die entsprechende Zielplattform compiliert werden muss. Der Unterschied zwischen Cmicro und Cadvanced besteht darin, dass Cmicro einen geringeren Sprachumfang besitzt als Cadvanced und speziell optimierten Kode erzeugt [25]. Cmicro ist speziell für eingebettete Systeme mit sehr geringen Ressourcen gedacht, wie zum Beispiel den MICAz.

4.1.3 ConTraST und SDLRE

ConTraST (**Co**nfigurable **Tra**nspiler for **S**DL to C++ **T**ranslation) [7] ist ein in der AG Vernetzte Systeme entwickelter Transpiler, der SDL-Systeme nach C++ übersetzt. Gegenüber Cadvanced und Cmicro besitzt ConTraST den Vorteil, dass die Quellen verfügbar sind und der Transpiler somit besser anpass- und erweiterbar ist. Außerdem ist der generierte Kode einfacher lesbar, und es existiert eine gute Verfolgbarkeit zwischen SDL-System und generiertem Kode. Dies ist gerade bei der Fehlersuche von Vorteil. Außerdem unterstützt ConTraST noch einige weitere SDL-Sprachfeatures, die Cadvanced und Cmicro nicht unterstützen [25]. Damit der mit ConTraST generierte Kode zur Ausführung gebracht werden kann, benötigt man noch eine Laufzeitumgebung (**SDL R**untime **E**nvironment). Diese Laufzeitumgebung wurde ebenfalls in der AG Vernetzte System entwickelt und ist eine manuelle Transformation der formalen Semantik von SDL-2000 in C++.

Da der Imote2 genügend Leistung besitzt, um den von ConTraST generierten Kode in Verbindung mit SDLRE ausführen zu können, wird während des gesamten Projektes ConTraST eingesetzt. Deswegen ist auch eine Unterstützung der Programmiersprache C++ auf dem Imote2 unbedingt nötig.

4.1.4 SEnF

Damit das SDL-System mit der Umgebung interagieren kann, benötigt man eine geeignete Schnittstelle. Ein mit Tau bzw. ConTraST erstelltes SDL-System kann Signale an die Umgebung senden, die Implementierung dieser sogenannten Environment Functions ist aber Sache des Benutzers. Tau kann bei der Kodeerzeugung automatisch eine C-Datei generieren, die die Funktionen XInEnv und XOutEnv enthält. Diese Funktionen behandeln ein- und ausgehende Signale des SDL-Systems. Dieser generierte Kode ist lediglich ein Gerüst und die eigentliche Anbindung der Umgebung muss vom Entwickler selbst realisiert werden. Liegt keine entsprechende Implementierung vor, so werden die Signale einfach verworfen.

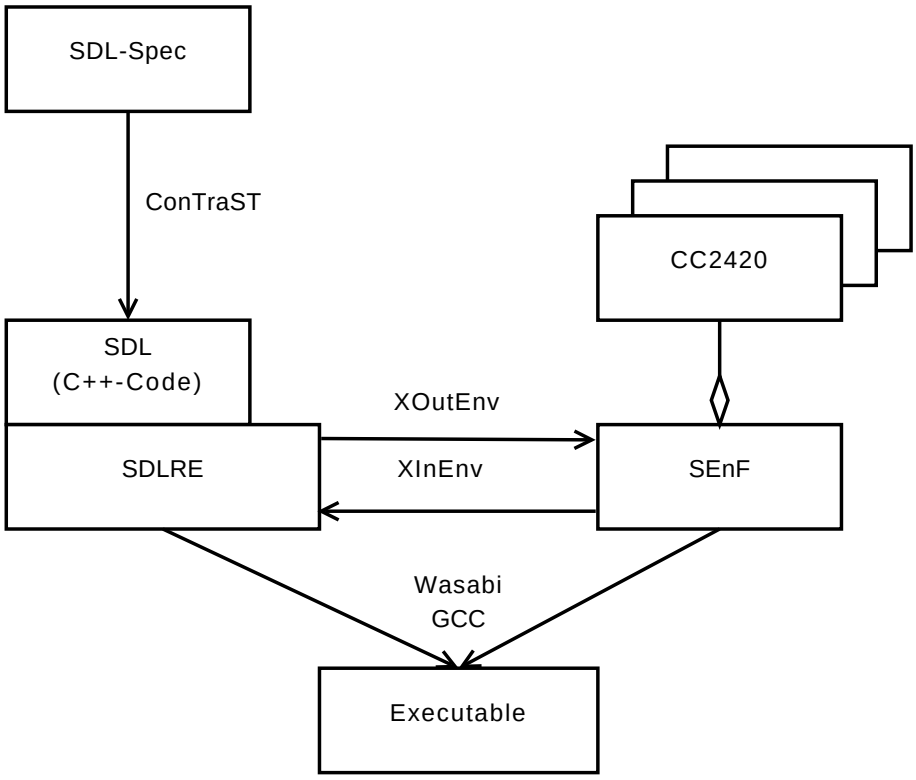

Abbildung 4.1: SDL-System mit SEnF für Imote2

Das SEnF stellt eine Sammlung von Schnittstellen zur Verfügung. SEnF wurde von der AG Vernetzte Systeme entwickelt und liegt mittlerweile in der zweiten Version vor [6]. Ziel ist es, die Umgebung für ein SDL-System automatisch und plattformunabhängig bereitzustellen. SEnF kann sowohl mit Cadvanced/Cmicro als auch mit ConTraST verwendet werden. Dazu besitzt SEnF eine Reihe von generischen Treibern, die als Module realisiert werden. Durch das Setzten der Option XENV wird SEnF automatisch beim Compiliervorgang eingebunden. Durch die Präcompilerdirektiven #define und #ifdef wird festgestellt, für welches Zielsystem compiliert werden soll. Durch die Nutzung von SDL-Packages in Tau, die für jede Kommunikationsschnittstelle einheitliche Signale definieren, ergibt sich, welche Module des SEnF geladen werden müssen. Die Einbindung des Kodes erfolgt über die Präcompilerdirektive #include. Somit wird für jedes System nur der Kode compiliert, der auch benötigt wird.

Jedes SEnF-Modul kann einen eigenständigen Thread starten. SDL-Signale, die an die Umgebung geschickt werden sollen, werden in der Funktion XOutEnv behandelt. Diese leitet das Signal an alle SEnF-Module weiter. Das SDL-Modul, für das das Signal bestimmt ist, empfängt das Signal und kann es nun verarbeiten. Zum Beispiel kann es empfangene Daten über eine Schnittstelle verschicken. Hat ein SEnF-Modul Daten empfangen und muss ein Signal an das SDL-System geschickt werden, benachrichtigt das Modul das SEnF mittels eines Notify(). Das Einfügen eines Signals darf nur dann erfolgen, wenn das SDL-System keine Transitionen ausführt, ansonsten kann es zu Konflikten mit den Warteschlangen der SDLRE kommen. Deswegen ruft die Laufzeitumgebung des SDL-Systems die XInEnv-Funktion zu einem Zeitpunkt auf, zu dem keine Transition ausgeführt wird. Werden andauernd Transitionen ausgeführt, so wird die XInEnv-Funktion nach einer bestimmten Zeit aufgerufen. Die XInEnv-Funktion überprüft dann, ob sie ein Notify() erhalten hat. Ist dies der Fall, wird das entsprechende Signal in das SDL-System eingefügt.

Abbildung 4.1 zeigt ein SDL-System für den Imote2 mit integriertem SEnF. Die SDL-Spezifikation wird mit Hilfe von Tau erstellt und anschließend mit ConTraST nach C++ übersetzt. Die Laufzeitumgebung des SDL-Systems kommuniziert mit dem SEnF über die Funktionen XInEnv und XOutEnv. Das SEnF selbst besteht aus den Modulen, die vom SDL-System benötigt werden. Die Dateien des SDL-Systems, der Laufzeitumgebung und des SEnF werden gemeinsam übersetzt und bilden eine Anwendung, die auf den Imote2 geladen wird.

4.2 Implementierungen der SEnF-Erweiterung

In diesem Abschnitt geht es um die eigentlichen Implementierungen, die während des Projektes gemacht werden. Im ersten Teil werden die Änderungen beschrieben, die an den SEnF XInEnv/ XOutEnv Funktionen vorgenommen werden, um den Imote2 zu unterstützen. Der zweite Teil handelt von der Realisierung der Plattform-Dateien für SEnF. Am Ende des Abschnittes werden alle neu implementierten SEnF-Module vorgestellt.

4.2.1 Änderungen an XInEnv/XOutEnv

Dass ein SEnF-Modul einen eigenen Thread startet, ist auf dem Imote2 momentan nicht möglich, da noch keine geeignete Thread-Implementierung vorliegt. Nun muss aber beispielsweise der Empfang eines Rahmens mit dem Transceiver jederzeit möglich sein. Dies wird beim Imote2 über Interrupts realisiert, sie können den Programmfluss jederzeit unterbrechen, auch wenn die Laufzeitumgebung gerade das SDL-System ausführt. So wird sichergestellt, dass Ereignisse in der Umgebung jederzeit behandelt werden können. Der Benachrichtigungsmechanismus wird leicht abgeändert, so läuft die Benachrichtigung nicht mehr über ein Aufruf der Funktion Notify(), sondern über eine Variable (DatenEmpfangen), die entsprechend gesetzt wird. Dazu besitzt jedes SEnF-Modul die folgenden Funktionen:

- SEnF_Module_Init: Diese Funktion wird beim Laden des Moduls aufgerufen. Sie startet, initialisiert und konfiguriert die für das Modul benötigte Hardware und initialisiert die benötigten Variablen.

- SEnF_Module_Execute: Dies ist die Verarbeitungsfunktion des Moduls, in den meisten Fällen wird sie von einem Interrupt aufgerufen. Als Parameter besitzt die Funktion die Nummer des aufgetretenen Interrupts, sodass es möglich ist, dass ein Modul an mehrere Interrupts gekoppelt ist. Die eigentliche Interruptverarbeitung findet in dieser Funktion statt, sie setzt auch die Variable DatenEmpfangen. Die Variable zeigt nur an, ob Signale an das SDL-System geschickt werden müssen und nicht wie viele. Werden beispielsweise mehrere Rahmen empfangen, so sind sie in geeigneten Datenstrukturen zwischenzuspeichern und dann zu senden (siehe Abschnitt 4.2.4).
- SEnF_Module_In: Diese Funktion wird regelmäßig von der SDL-Laufzeitumgebung in der Funktion XInEnv aufgerufen und zwar immer dann, wenn das SDL-System bereit ist Signale von der Umgebung zu empfangen. Die Funktion überprüft die Variable DatenEmpfangen, ist diese gesetzt, müssen Signale versendet werden. Die entsprechenden Signale werden dann generiert und an das SDL-System geschickt. Danach wird Variable DatenEmpfangen zurückgesetzt. Müssen keine Signale versendet werden, kehrt die Funktion sofort zurück.
- SEnF_Module_Out: Wenn ein Signal an die Umgebung gesendet werden soll, wird diese Funktion von der XOutEnv Funktion aufgerufen. Die Funktion stellt fest, um welches Signal es sich handelt, und ob das Signal für das Modul bestimmt ist. Ist dies der Fall, werden die Parameter aus dem Signal gelesen und entsprechend verarbeitet.

Für jedes neue Modul, das dem SEnF hinzugefügt wird, müssen die hier vorgestellten Funktionen implementiert werden. Daneben besitzt jedes Modul noch eine zentrale Datenstruktur SEnF_Module_Data. Sie enthält die Variable DatenEmpfangen und gegebenenfalls weitere modulspezifischen Daten. Der grundsätzliche Ablauf stellt sich folgendermaßen dar: Durch einen Interrupt empfängt das Modul Daten, die an das SDL-System weitergegeben werden sollen. Die Variable DatenEmpfangen wird gesetzt und die für das Signal notwendigen Daten in der Datenstruktur gespeichert. Wird die SEnF_Module_Out-Funktion des Moduls aufgerufen, kann das Signale mit den zwischengespeicherten Daten erstellt und versendet werden.

4.2.2 Plattformintegration

Für jede unterstützte Plattform gibt es im SEnF einen Ordner, der alle plattformspezifischen Funktionen enthält, die aber nicht zu einem bestimmten Modul gehören. Für die Imote2 Plattform werden folgende Dateien implementiert, die plattformspezifischen Kode bereitstellen:

- `inttypes.h`: Diese Datei enthält die Deklaration der Integerdatentypen für den Imote2, sie wurde aus TinyOS übernommen.
- `os_arm_bare.c/.h`: Zentrale Datei, die alle anderen Dateien in diesem Ordner einbindet. Zusätzlich enthält sie die überschriebenen Funktionen malloc_lock und malloc_unlock (siehe Abschnitt 3.3.2).
- `os_arm_bare_coio.c`: Die Datei enthält eine Implementierung des SEnF-Errors für den Imote2. Da der Imote2 keine Konsole besitzt, kann die Fehlernachricht nicht direkt ausgegeben werden. Statt dessen werden bei einem Fehler alle LEDs des Imote2 auf blinkend geschaltet. Ist das UART-Modul geladen, wird die Fehlernachricht zusätzlich über den ersten UART-Anschluss gesendet.
- `os_arm_bare_driver.c/.h`: Enthält Kode, um Module auf der Imote2 Plattform zu laden.
- `os_arm_bare_inithardware.c`: Enthält die Initialisierung der Hardware des Imote2. Die Funktion initHardware wird im Konstruktor der Laufzeitumgebung aufgerufen und aktiviert die MemoryManagementUnit, den Datencache, den Befehlscache, sowie die Möglichkeit, Interrupts auszulösen, außerdem setzt sie die initiale Prozessortaktfrequenz. Der Kode zum Ändern der Prozessortaktfrequenz befindet sich ebenfalls in der Datei.
- `os_arm_bare_inout.c`: Enthält die Funktionen XInEnv und XOutEnv für den Imote2.

- `os_arm_bare_interrupt.c/.h`: Der zentrale Interrupthandler des Imote2 befindet sich in diesen Dateien. Wenn ein Modul auf einen Interrupt reagieren soll, so registriert sich das Modul beim Laden beim SEnF-Interrupthandler. Wenn ein Interrupt auftritt, der für ein SEnF-Modul bestimmt ist, so ruft der zentrale Interrupthandler den SEnF-Interrupthandler auf, der dann wiederum die entsprechende Execute-Funktion mit der entsprechenden Interrupt-ID aufruft. Der Imote2 unterstützt zwei verschiedene Interruptarten, IRQ (Interrupt Request) und FIQ (Fast Interrupt Request). Es ist frei konfigurierbar, welche Art von Interrupt eine Interruptquelle auslöst. FIQ sind für besonders wichtige Interrupts gedacht, da sie ein höhere Priorität als normale IRQs haben [10]. In dieser Arbeit, wie auch in TinyOS, werden ausschließlich IRQs benutzt.

- `os_arm_bare_registers_def.h`: Diese aus TinyOS übernommene Datei enthält Makros für den Zugriff auf die Register des PXA271. Sie ermöglichen es, ein Register mit seinem Namen statt seiner Adresse anzusprechen.

- `os_arm_bare_time.c`: Ein SDL-System benötigt eine Zeitbasis, diese Datei enthält den Kode der die Zeitbasis bereitstellt. Wichtig für das SDL-System ist, dass die Zeit in keinem Fall rückwärts läuft, sonst kommt es zu Fehlern. Ein Hardware-Timer des PXA271 wird ausschließlich für die SDL-Zeit verwendet. Es ist wichtig zwischen dem Hardware-Timer des Prozessors und einem Timer in einem SDL-System zu unterscheiden. Der Hardware-Timer wird so konfiguriert, dass er nicht abläuft. Der Hardware-Timer wird somit als Uhr verwendet. Die Inkremente des Hardware-Timers betragen $\frac{1}{32768}$ s, also ungefähr 30 μs. Das SDL-System erhält die Zeit in Sekunden und Nanosekunden. Die 15 niederwertigsten Bits des Zählregisters des Timers geben die Sekundenbruchteile an. Diese müssen in Nanosekunden umgerechnet werden. Da die SDL-Zeit sehr häufig abgefragt wird, muss die Umrechung schnell durchgeführt werden. Eine exakte Rechnung mit Gleitkommazahlen scheidet somit aus, die Umrechnung muss mittels Integermultiplikation durchgeführt werden. Die höchste mit 15 Bit darstellbare Zahl ist 32767, so muss gelten:

  $32767 \cdot x\,\text{ns} < 1\,\text{s} : x \in \mathbb{N}$

  Die größte natürliche Zahl, für die der Ausdruck gilt, ist 30518. Rechnet man exakt, so erhält man für den Zählerstand des Timers von 32767

  $32767 \cdot \frac{1}{32768}\,\text{s} \approx 999969482\,\text{ns}$

  Der näherungsweise berechnete Wert ist

  $32767 \cdot 30518\,\text{ns} = 999983306\,\text{ns}$

  Der maximale Fehler beträgt somit

  $999983306\,\text{ns} - 999969482\,\text{ns} = 13824\,\text{ns}$

  Die verstrichenen Sekunden können exakt berechnet werden, da $2^{15} = 32768$ ist. Bei einem Inkrement von $\frac{1}{32768}$ s, ist dies genau eine Sekunde. Also geben die oberen 17 Bit des Zählregisters exakt die verstrichenen Sekunden an. Da das Zählregister 32-Bit besitzt, kommt es bei langen Laufzeiten des Knotens zu einem Überlauf des Hardware-Timers. Der Hardware-Timer wird so konfiguriert, dass nach genau 36 Stunden ein Interrupt ausgelöst und der Timer dann wieder auf 0 gesetzt wird. Durch den Interrupt wird die SDL-Laufzeitumgebung benachrichtigt, dass ein Überlauf aufgetreten ist und 36 Stunden auf die jetzt gemessene Zeit aufgeschlagen werden müssen. Der Hardware-Timer wird bei der SEnF-Plattforminitialisierung, also beim Start der Hardwareplattform, gestartet.

4.2.3 SEnF-Modul LED

Das Modul ist für die Ansteuerung der LEDs des Imote2 zuständig. Im SDL-System stehen dann folgende ausgehende Signale zur Verfügung:

- ToggleLed(Integer): die LED mit der entsprechenden Nummer wird umgeschaltet.

- SetLed(Integer,Boolean): die LED mit der entsprechenden Nummer wird eingeschaltet (TRUE) oder ausgeschaltet (FALSE).

Der Kode, der im Minimalbeispiel verwendet wird (Abschnitt 3.2), kann an dieser Stelle übernommen werden. Lediglich die Signalbehandlung wird hinzugefügt. Die entsprechenden Funktionen wie redOn() und redOff() werden aus der SEnF_LED_IN-Funktion aufgerufen.

4.2.4 SEnF-Modul UART

Mit diesem Modul ist es möglich, die drei UART-Schnittstellen des Imote2 zu nutzen. Beim Einbinden des Moduls werden alle drei Schnittstellen aktiviert. Ein gleichzeitiges Nutzen ist möglich. Jede Schnittstelle hat drei Leitungen, eine Sendeleitung, eine Empfangsleitung und eine Erdungsleitung. Es sind nur direkte Verbindungen zwischen zwei Teilnehmern möglich. Eine Übertragung läuft folgendermaßen ab: Der Sender sendet das Startbit gefolgt von fünf bis acht Datenbits und einem optionalen Paritätsbit und als letztes ein oder zwei Stoppbits. Start- und Stoppbits dienen zur Synchronisation mit dem Empfänger. Um miteinander kommunizieren zu können, müssen beide Teilnehmer mit derselben Baudrate arbeiten, ansonsten kommt es zu Übertragungsfehlern. In der vorliegenden Implementierung wird kein Paritätsbit genutzt, immer 8 Datenbits gesendet und ein Stoppbit verwendet. Es werden die Geschwindigkeiten 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600 und 115200 Baud pro Sekunde unterstützt. Diese Geschwindigkeiten reichen aus, um die in der Praxis vorkommenden Anschlüsse zu unterstützten.

Im SDL-System stehen folgende ausgehende Signale zur Verfügung:

- UART_config(Integer,Integer): Mit Hilfe dieses Signals kann der Benutzer die Geschwindigkeit des UART-Ports setzen. Der erste Parameter gibt den entsprechenden UART-Port (1 bis 3) an, der zweite die neue Geschwindigkeit. Wird die neue Geschwindigkeit nicht unterstützt, wird automatisch die Maximalgeschwindigkeit gesetzt.
- UART_send(Integer,Octet_string): Dieses Signal versendet eine Bytefolge über den UART-Port. Der erste Parameter enthält den entsprechenden Port, der zweite die Daten, die gesendet werden sollen.

Und folgende Signale können von der Umgebung empfangen werden:

- UART_recv(Integer,Octet_string): Wenn eine Bytefolge über eine der UART-Schnittstellen empfangen wurde, wird dieses Signal an das SDL-System gesendet. Der erste Parameter enthält den Port, der zweite die empfangene Bytefolge.
- UART_sendFinished(Integer): Wenn eine Bytefolge erfolgreich versendet wurde, wird dieses Signal an das SDL-System gesendet. Der Parameter gibt den Port an, auf dem die Bytefolge gesendet wurde. Es soll damit verhindert werden, dass Signale zu schnell an das Modul gesendet und somit die UART-Schnittstelle überlastet wird.

Die UART-Schnittstellen des Imote2 besitzen je einen Sendepuffer und einen Empfangspuffer mit einer Größe von 64 Byte. Beide Puffer sind als FIFO-Warteschlange realisiert. Gelesen und geschrieben werden sie über ein Register. Das Schreiben des Registers der Sendewarteschlange fügt ein Byte an das Ende der Warteschlange an, ein Lesen des Registers der Empfangswarteschlange liest das erste Element und löscht es anschließend aus der Warteschlange.

Gesteuert wird der Empfang von Bytes über Interrupts. Jedesmal, wenn ein Byte empfangen wird, wird ein Interrupt ausgelöst und die Execute-Funktion kann das empfangene Byte aus dem Empfangspuffer lesen. Bei großer Last und entsprechend geringer Taktfrequenz des Prozessors kann es vorkommen, dass die Interruptbearbeitung langsamer ist, als das Empfangen des nächsten Bytes. So kann es passieren, dass sich mehr als ein Byte im Empfangspuffer befindet. In diesem Fall liest die Interruptbehandlungsroutine alle im Empfangspuffer vorhandenen Bytes auf einmal aus.

Damit nicht jedes empfangene Datenbyte als Signal an das SDL-System geschickt werden muss, ist es nötig Bytefolgen zu erkennen. Kommen mehrere Datenbytes direkt hintereinander über dieselbe UART-Schnittstelle, so wird davon ausgegangen, dass diese zu einer Bytefolge gehören. Die Datenbytes werden dann in einem Signal an das SDL-System gesendet. Eine Grenze einer Bytefolge wird dann erkannt, wenn für eine Zeit lang keine Daten über den Port empfangen werden. Diese

Zeit entspricht der Zeit, die der UART-Port benötigt, um drei Bytes zu versenden, sie ist also abhängig von der Geschwindigkeit der UART-Schnittstelle. Um festzustellen, ob die spezifizierte Zeit abgelaufen ist und eine Bytefolgegrenze vorliegt, benötigt man einen Timer. Einer der Timer des PXA271 wird ausschließlich für die UART-Schnittstellen benutzt. Er läuft mit einer Auflösung von einer Mikrosekunde. Immer wenn ein Byte über den UART empfangen wird, registriert sich die Schnittstelle mit einem entsprechenden Event bei dem Timer. Läuft der Timer ab, wird eine Grenze einer Bytefolge erkannt. Kommt in der Zeit, in der der Timer läuft, ein weiteres Datenbyte an, wird der Timer neu gestartet und keine Grenze erkannt. Die empfangenen Bytes werden in einem Array zwischengespeichert, bis eine Bytefolgegrenze erkannt wird oder die maximale Länge der Bytefolge (100 Byte) erreicht ist. Dieses Verfahren funktioniert als Rahmenerkennung nicht immer zuverlässig. Es kann passieren, dass zwei oder mehr Rahmen zu einer Bytefolge zusammengefasst werden. Notwendige Protokolle für eine zuverlässige Rahmenerkennung sind Aufgabe des SDL-Systems. Im SDL-System können dann beispielsweise Längenfelder oder Rahmengrenzen spezifiziert werden. Ist ein Rahmen länger als die maximale Länge der Bytefolge, so muss das SDL-System ebenfalls dafür sorgen, dass der Rahmen wieder richtig zusammengesetzt wird.

In der ersten Version des Moduls werden die erkannten Bytefolgen in mehreren Puffer fester Größe zwischengespeichert. Diese Zwischenspeicherung ist notwendig, da das Versenden von Signalen zu dem SDL-System asynchron geschieht. Man kann nicht vorhersagen, wann genau das Signal gesendet wird. Es können mehrere Bytefolgen empfangen werden, bevor das System bereit ist, Signale an das SDL-System zu schicken. Folglich müssen auch mehrere Puffer bereitstehen. Dies führt aber zu einem erheblichen Verwaltungsaufwand, da beim Erkennen der Bytefolgegrenze zwischen den einzelnen Puffer umgeschaltet werden muss, damit keine Daten verloren gehen. Auch muss der Zustand der Puffer gespeichert werden. Der Zustand eines Puffers enthält, ob der Puffer gerade benutzt wird, ob er frei ist und wenn der Puffer besetzt ist, wie viele Bytes in ihm gespeichert sind. Der daraus resultierende Kode ist fehleranfällig, weshalb eine andere Lösung genutzt wird.

Statt mehrere Puffer zu benutzen, werden die erkannten Bytefolgen in einer einfach verketteten Liste gespeichert. Sie ist einfacher zu verwalten als mehrere Puffer. Wird eine Bytefolge in die Liste aufgenommen, wird gleichzeitig die DatenEmpfangen Variable gesetzt,um zu signalisieren, dass ein Signal nach SDL gesendet werden muss. Wenn jetzt mehrere Bytefolgen empfangen werden, bevor das System bereit ist Signale an das SDL-System zu senden, werden diese Bytefolgen an die Liste angehängt. Wenn das System bereit zum versenden von Signalen ist, wird für jedes Element der Liste ein Signal generiert und gesendet und der entsprechende Listeneintrag gelöscht.

Das Versenden von Bytefolgen über die UART-Schnittstelle läuft folgendermaßen ab: Wird das Signal zum Datenversenden empfangen und ist die Länge der zu versendenden Bytefolge kleiner gleich 64 Byte, dann kann die Bytefolge vollständig in den Sendepuffer geschrieben werden und wird dann automatisch gesendet. Ist die Bytefolge größer als 64 Byte, muss beim Füllen des Sendepuffers eine Pause gemacht werden, damit es zu keinem Überlauf des Puffers kommt. Hierfür wird der oben erwähnte Timer des Moduls verwendet. Die Sendefunktion registriert ein Event beim Timer. Die Wartezeit entspricht der Zeit, die der UART benötigt, um 48 Bytes zu versenden. Nach dieser Zeit hat sich der Sendepuffer geleert und weitere zu sendende Bytes können in den Puffer geschrieben werden. Dieser Vorgang wiederholt sich solange, bis die gesamte Nachricht gesendet wurde. Um ein zu schnelles Senden zu verhindern, muss dem SDL-System signalisiert werden, wann es eine neue Bytefolge senden darf. Dafür gibt es das oben erwähnte Signal UART_sendFinished. Jedesmal wenn der Sendepuffer nach einem Sendevorgang leer wird, wird ein Interrupt ausgelöst, der ein Signal an das SDL-System sendet. Es ist wichtig, dass der Puffer erst leer ist, nachdem die komplette Bytefolge gesendet wurde, ansonsten würde das UART_sendFinished Signal zu früh gesendet.

4.2.5 SEnF-Modul CC2420

Dieses Modul ermöglicht die Nutzung des CC2420 Transceivers. Gegenüber den meisten anderen Schnittstellen kann der CC2420 nur im Halbduplexmodus arbeiten: Es ist nur möglich, entweder zu Empfangen oder zu Senden. Ob der CC2420 gerade sendet oder empfängt, wird durch interne Zustände angezeigt [20]. Standardmäßig befindet sich der CC2420 im Empfangsmodus, der Chip wartet nun auf eine Präambel, die ihm anzeigt, dass jetzt etwas auf seinem Kanal gesendet wird. Um etwas zu senden, muss der CC2420 in den Sendemodus wechseln. Neben diesen beiden Modi

gibt es noch den Idle-Modus. Dieser dient vor allem zum Energiesparen, da bei drahtlosen Systemen auch die Empfangsbereitschaft Energie benötigt. Schließlich gibt es noch den Modus, in dem der CC2420 heruntergefahren ist.

Da der CC2420 auch auf der Hardwareplattform MICAz verwendet wird, existiert schon eine Implementierung des Treibers in SEnF. Große Teile des Kodes können mit geringen Änderungen übernommen werden. Neu implementiert wird die Kommunikation zwischen CC2420 und PXA271, da das SPI-Interface sich auf dem Imote2 anders verhält als auf dem MICAz. Auch die Signalverarbeitung und Interruptbehandlung laufen auf dem MICAz anders, als auf dem Imote2, so müssen diese Funktionen ebenfalls neu implementiert werden.

Dem SDL-System stellt dieses Modul folgende ausgehende Signale zur Verfügung:

- CC2420_SEND(Octet_string): Dieses Signal sendet den im Parameter übergebenen Datenrahmen über den CC2420. Es wird nicht darauf geachtet, ob der Kanal frei ist.
- CC2420_SEND_CCA(Octet_string): Hier wird der Rahmen nur gesendet, wenn der Kanal frei ist. Der Kanal ist frei, wenn das CCA-Bit (Clear Channel Assessment) gesetzt ist. Kann der Rahmen nicht gesendet werden, wird nicht noch einmal versucht den Rahmen zu senden. Für einen neuen Versuch ist das SDL-System zuständig.
- CC2420_SETUP(Integer,Integer): Mit Hilfe dieses Signals kann der Kanal und die Sendeleistung des CC2420 gesetzt werden. Der erste Parameter gibt den gewünschten Kanal an (Kanäle 11 bis 26 sind erlaubt), der zweite die gewünschte Sendeleistung (Werte zwischen 0 und 32 erlaubt).
- CC2420_MODE(Integer): Mit diesem Signal lassen sich alle oben erwähnten Modi des CC2420 manuell einstellen, mit einer Ausnahme. In den Sendemodus gelangt man ausschließlich über die beiden Send-Signale. Der Parameter gibt an, welcher Modus eingestellt werden soll (Werte von 0 bis 3 erlaubt).
- CC2420_RECEIVE(Boolean): Ist der Parameter des Signals TRUE, wird der Transceiver in Empfangsmodus gesetzt, ansonsten in den Idle-Modus.

Das Modul kann folgende Signale an das SDL-System senden:

- CC2420_RECV(Octet_string, Boolean, Integer): Wenn ein Rahmen empfangen wurde, wird dieses Signal an das SDL-System gesendet. Der erste Parameter enthält den empfangenen Rahmen, der zweite gibt an, ob die CRC-Checksumme erfolgreich gebildet werden konnte und der dritte Parameter enthält die Signalstärke, mit der der Rahmen empfangen wurde.
- CC2420_SFD(Boolean): Dieses Signal zeigt dem SDL-System an, dass ein Rahmen erfolgreich gesendet wurde. Der übergebene Parameter ist immer TRUE.
- CC2420_SENDING(Boolean): Diese Signal ist die Antwort auf die Signale CC2420_SEND und CC2420_SEND_CCA des SDL-Systems. Wurde mit dem Senden begonnen, so ist der Parameter TRUE, wurde nichts gesendet, etwa weil der Kanal besetzt ist, so ist er FALSE.
- CC2420_CCA(Boolean): Mit diesem Signal wird das SDL-System über den Kanalstatus informiert. Ist der Kanal frei, ist der übergebene Parameter TRUE, wenn nicht FALSE. Es wird nur dann ein Signal gesendet, wenn sich der Kanalstatus ändert.

Die Kommunikation zwischen Transceiver und Prozessor läuft über SPI-Interface. Die Geschwindigkeit dieser Verbindung beträgt 6,5 Mbps. Der CC2420 unterstützt zwar Taktraten bis 10 Mhz, aber die Taktrate des SPI-Interfaces des Prozessors, der als Master fungiert, kann nur ein ganzzahliger Teil von 13 MHz sein. So wird die maximal mögliche Geschwindigkeit genutzt. Die übertragenen Datenbits pro SPI-Frame kann beim PXA721 frei von 4 bis 32 gewählt werden. Der CC2420 unterstützt nur 8 Bits pro Frame. Die Chip-Select Leitung wird nicht von dem SPI-Interface des PXA271 gesteuert, sondern vom Programmierer. Dies ist nötig, da die Spezifikation des CC2420 verlangt, dass die Chip-Select Leitung bei Befehlen, die sich über mehrere Frames erstrecken, dauerhaft gesetzt sein muss. Bei der automatischen Steuerung wird die ChipSelect-Leitung zwischen

den einzelnen Frames zurückgesetzt und der CC2420 erkennt die an ihn gesendeten Befehle nicht. Daten können beim SPI-Interface nur dann von einem Slave empfangen werden, wenn der Master ihm Daten sendet. Der PXA271 besitzt einen Sende- und Empfangspuffer für je 16 Frames. Es können also mehrere Frames zum CC2420 gesendet werden, die Antwortframes stehen dann im Empfangspuffer des SPI-Interface. Es gibt drei unterschiedliche Möglichkeiten, mit dem CC2420 zu kommunizieren. Die erste Möglichkeit sind die *Command Strobes*. Dies sind kurze Befehle, die in einem Frame zum CC2420 gesendet werden. Mit ihnen wird zum Beispiel der CC2420 in den Sendemodus gesetzt oder der Empfangspuffer geleert. Für eine vollständige Auflistung aller *Command Strobes* siehe [20]. Die Antwort des CC2420 auf solch einen Befehl ist immer ein Byte mit Statusinformationen. Die zweite Möglichkeit ist das Schreiben oder Lesen eines Registers des CC2420. Das erste gesendete Byte enthält die Adresse des Registers, das man Lesen oder Schreiben möchte. Dann folgen im Falle vom Schreiben die in das Register zu schreibenden Daten. Möchte man aus dem Register lesen, schickt man dem CC2420 beliebige Bytes, die Antwort enthält dann den Inhalt des Registers. Die dritte Möglichkeit ist das Schreiben in den Sendepuffer bzw. das Lesen des Empfangspuffers. Es funktioniert genauso wie das Lesen/Schreiben eines Registers, nur dass entsprechend mehr Frames ausgetauscht werden können. Der CC2420 besitzt einen Sendepuffer und einen Empfangspuffer für je 128 Bytes. Das erste Byte ist immer das Längenbyte, das angibt, wie lange der Rahmen ist. Wird ein Rahmen empfangen, enthalten die letzten beiden Bytes das Ergebnis der CRC-Prüfsummenberechnung und die Signalstärke, mit der der Rahmen empfangen wurde. Somit ist eine maximale Nutzlast von 125 Byte pro Rahmen möglich. Muss ein Rahmen übertragen werden, das größer als 16 Bytes ist, reicht der Sende- bzw. Empfangspuffer des SPI-Interface nicht mehr aus. In diesem Fall wird aktiv gewartet bis der Sende- bzw. Empfangspuffer wieder geleert wird, dann wird das Senden fortgesetzt.

Hat der CC2420 einen Rahmen vollständig empfangen, so setzt er die FIFOP-Leitung, diese ist mit einem GPIO-Pin des PXA271 verbunden. Der GPIO-Pin ist so konfiguriert, dass eine steigende Flanke einen Interrupt auslöst. Die Interruptbehandlung holt dann den empfangenen Rahmen über das SPI-Interface. Es kann vorkommen, dass das SPI-Interface gerade einen Rahmen zum CC2420 überträgt, dann ist eine Übertragung zunächst nicht möglich. In diesem Fall wird auf die Beendigung des Transfers gewartet, und dann der Rahmen ausgelesen. Beim Versenden eines Rahmens wird der Rahmen zunächst zum CC2420 übertragen, danach wird gesendet (mit oder ohne CCA). Mit Hilfe des Statusbytes kann festgestellt werden, ob sich der CC2420 im Sendemodus befindet und das entsprechende SFD-Signal kann an das SDL-System geschickt werden. Die SFD-Leitung ist ebenfalls an einen GPIO-Pin des PXA271 angeschlossen. Die SFD-Leitung ist gesetzt, wenn ein Rahmen gesendet wird. Wird mit dem Senden begonnen, so wird ein Interrupt für diesen GPIO-Pin aktiviert, der auf eine fallende Flanke achtet. Somit kann festgestellt werden, wann das Senden abgeschlossen ist. Der dritte Interrupt kann von der CCA-Leitung des CC2420 kommen. Jedesmal wenn eine fallende oder steigende Flanke auftritt, wird ein Interrupt ausgelöst und der CCA-Status aktualisiert.

Kapitel 5

Anwendung

Die integrierten SEnF-Module werden nun in einer Anwendung benutzt. Die Anwendung realisiert ein Modemszenario (Abbildung 5.1): Es gibt zwei PCs, die jeweils per UART-Schnittstelle mit einem Imote2 verbunden sind. Die beiden Imote2 können über die CC2420-Schnittstelle kommunizieren. Über diese Verbindung werden nun Daten von einem PC zum Anderen übertragen. Außerdem ist es möglich, die blaue Leuchtdiode jedes Imote2 anzusteuern.

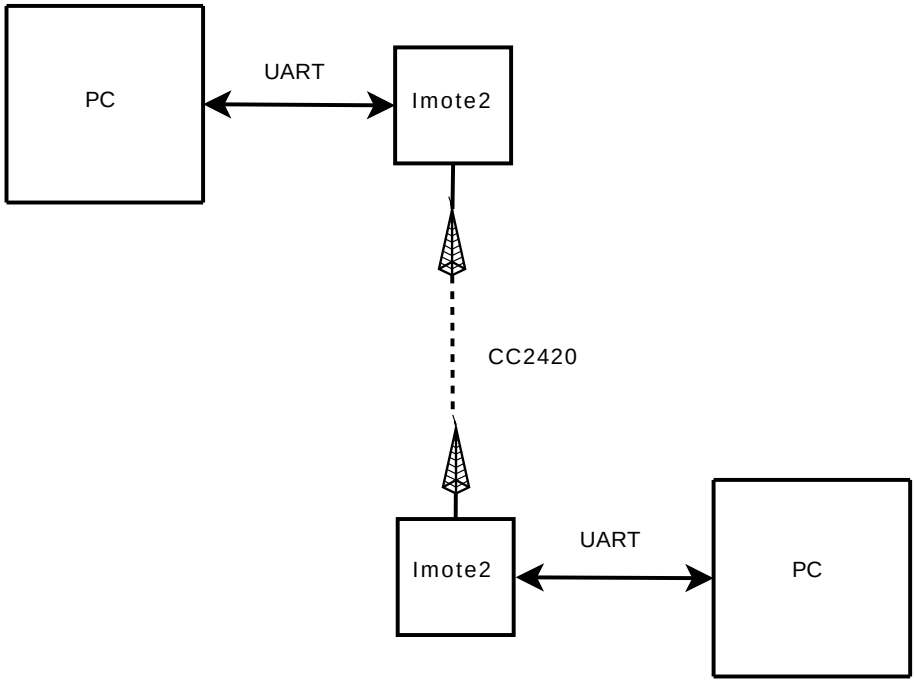

Abbildung 5.1: Anwendungsszenario

Die gesamte Anwendung besteht aus drei Teilen:

- SDL-System auf dem Imote2

  Auf jedem Imote2 läuft ein SDL-System, das die Daten, die es über die UART-Schnittstelle vom PC empfängt, über die CC2420-Schnittstelle versendet und umgekehrt. Daneben sendet das SDL-System in festen Intervallen die Adresse des Imote2 über UART und CC2420 aus. Jeder Imote2 hat eine eindeutige Adresse, die aus acht Hexadezimalziffern besteht. Sie befindet sich im FLASH-Speicher und kann ausgelesen werden. Die Adresse wird in einem Rahmen mit einer speziellen Präambel versendet, um es als Adressrahmen zu kennzeichnen. Ebenfalls eine spezielle Präambel besitzen Rahmen, die blauen LEDs der Imote2 ansteuern. Wenn der

Imote2 einen solchen Rahmen empfängt, vergleicht er die im Rahmen gespeicherte Zieladresse mit seiner eigenen. Stimmen beide Adressen überein, wird die blaue LED umgeschaltet, ansonsten wird der Rahmen an den nächsten Imote2 gesendet.

- SDL-System auf dem PC

  Das SDL-System auf dem PC empfängt und sendet Datenrahmen über die UART-Schnittstelle. Die Kommunikation zwischen dem SDL-System und der C++-Applikation läuft über das in der AG entwickelte VSApplicationInterface. Es definiert eine Reihe von SDL-Signalen und C++-Methoden. Mit diesem Interface ist es einerseits möglich, dass eine Applikation einem SDL-System Signale zusenden kann, andererseits kann das SDL-System Signale versenden, die dann auf Funktionsaufrufe innerhalb der Applikation abgebildet werden. In beiden Fällen werden UDP-Sockets für den Datentransfer genutzt.

- C++-Applikation auf PC

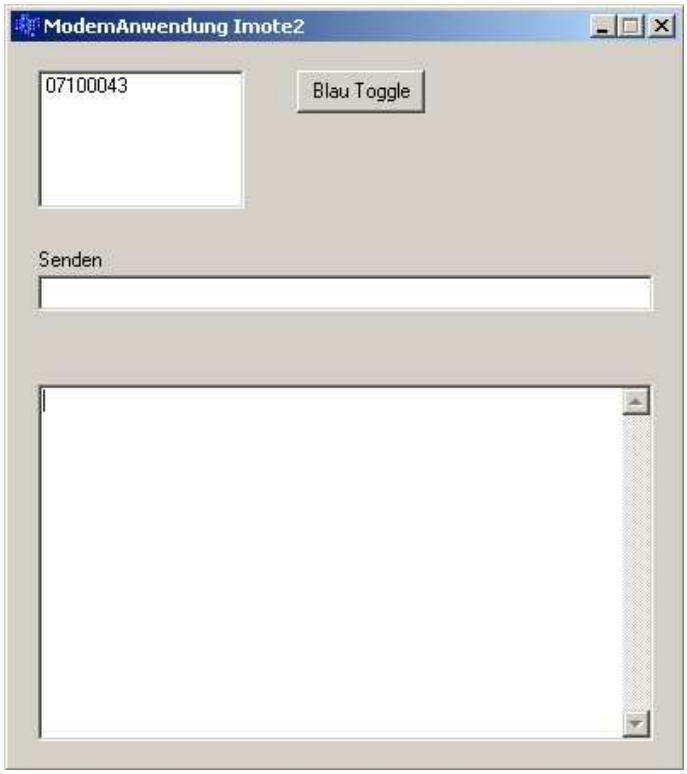

Abbildung 5.2: Die GUI der C++-Applikation

Die C++-Applikation realisiert die Schnittstelle zum Benutzer. Hierfür wurde eine kleine graphische Oberfläche entwickelt (Abbildung 5.2). Das linke obere Textfenster zeigt die Adressen der Imote2 an, die aktuell mit dem Netzwerk verbunden sind (im Moment maximal zwei). Wählt der Benutzer eine der Adressen und klickt auf den BlauToggle-Knopf, so wird die blaue LED des ausgewählten Imote2 umgeschaltet. Das Textfenster in der Mitte dient zur Eingabe der Daten, die an den anderen PC versendet werden sollen und das untere Textfenster zeigt die vom anderen PC empfangenen Daten an. Über das oben erwähnte VSApplicationInterface ist die C++-Applikation mit dem SDL-System des PCs verbunden.

Diese einfache Anwendung zeigt, dass alle neu integrierten SEnF-Module funktionieren und dass der Imote2 in Netzwerken eingesetzt werden kann.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde die SEnF-Erweiterung für den Imote2 vorgestellt. Es wurde die für die Programmierung des Imote2 benötigte Software vorgestellt und ein Minimalbeispiel implementiert, dass die Funktionsfähigkeit des Imote2 demonstrierte. Die dabei aufgetretenen Probleme wurden analysiert und behoben, so dass C++ als Programmiersprache für den Imote2 verwendet werden kann. Um die Kommunikationshardware des Imote2 unter SDL nutzen zu können, wurde die Plattform in das SEnF integriert und drei neue SEnF-Module implementiert, die eine definierte Schnittstelle, die aus SDL-Signalen besteht, besitzen. Dieses ermöglichen es, die UART-Schnittstellen, die CC2420-Schnittstelle und die LEDs zu nutzen. Wie die Nutzung der neu implementierten Module aussehen kann, wurde mit Hilfe einer einfachen Modemanwendung gezeigt.

Nach Abschluss der Arbeit bleiben noch zwei Punkte offen. Erstens ist es im Moment nicht möglich, die 32 MB SDRAM des Imote2 zu nutzen. Die von TinyOS übernommenen Speicherinitialisierungsroutinen aktivieren den SDRAM des Imote2 nicht und im Linkerskript wird der SDRAM nicht als Speicherbereich angelegt. Es werden bis jetzt ausschließlich die 256 kB SRAM genutzt. Gerade im Hinblick auf zukünftige Anwendungen auf dem Imote2, die einen höheren Speicherbedarf besitzen, sollte es in Zukunft möglich sein, den SDRAM zu nutzen. Der zweite Punkt ist die Implementierung des Datentransfers über das SPI-Interface zwischen dem PXA271 und dem CC2420. Aus Zeitgründen wurde die Implementierung des MICAz übernommen, die aktives Warten nutzt. Der Imote2 erlaubt aber wegen seiner Empfangs- und Sendepuffer eine interruptgesteuerte Datenübertragung über das SPI-Interface. So wäre der PXA271 während einer Übertragung nicht mehr blockiert, was wertvolle Ressourcen spart. Der interruptgesteuerte Datentransfer ist aber bis jetzt noch nicht implementiert.

Für weitere Kommunikationsschnittstellen, wie zum Beispiel das I^2C-Interface, sollten ebenfalls SEnF-Module entwickelt werden, damit die Schnittstellen in einem SDL-System nutzbar sind. Über das Sensorboard, das parallel zu dieser Arbeit in der AG entwickelt wird, können in Zukunft weitere Sensoren an den Imote2 angeschlossen werden. Außerdem dient das Sensorboard zum Anschluss einer Energieversorgung, enthält Audio-Hardware und stellt Anschlüsse für UART- und I^2C-Schnittstellen bereit. Die Verarbeitung der Daten der Sensoren muss ebenfalls implementiert werden, auch hierfür müssen SEnF-Module erstellt werden.

Anhang A

Anleitung zum Laden von Images auf den Imote2

Die folgende Anleitung beschreibt, welche Schritte man unternehmen muss, um ein SDL-System mit integriertem SEnF auf den Imote2 zu laden. Alle wichtigen benötigten Dateien befinden sich in einem Ordner auf dem AG-Server.

- Cygwin installieren. In der Arbeit wurde die aktuelle Cygwin Version benutzt (`http://www.cygwin.com/`).
- Wasabi Toolchain für Cygwin installieren, am besten in das Rootverzeichnis von Cygwin. Nun muss der Order *Installationsverzeichnis*/usr/local/bin/ in die Cygwin Umgebungsvariable PATH eingetragen werden. Danach sollte der Compiler in einer Cygwin Shell mit dem Befehl

  ```
  xscale-elf-gcc
  ```

  in jedem Ordner ausführbar sein.
- SDL-System mit Telelogic Tau erstellen. Möchte man bestimmte Kommunikationsschnittstellen benutzen, so muss man die für die Schnittstelle benötigten Signale definieren und benutzen. Danach in Tau vom erstellten SDL-System eine PR-Datei generieren (Menü ConTraST -> Generate PR).
- Nun muss aus der PR-Datei Kode generiert werden. Dazu startet man eine Cygwin-Shell wechselt in den Ordner in dem sich die PR-Datei befindet und gibt folgenden Befehl ein:

  ```
  ConTraSTv2.exe -O "-DXENV" -C "V:\Contrast\config.ini" --linux
  --sdlre /cygdrive/p/SDL/SdlRE/ --senf /cygdrive/p/SDL/SENF/ PRfile
  ```

  Sollten SDLRE und SEnF sich in anderen Ordern befinden, muss der Befehl entsprechend angepasst werden. Laufwerke spricht Cygwin grundsätzlich über /cygdrive/laufwerksname an. In Zukunft ist geplant die Kodeerzeugung über einen Menüpunkt in Tau zu regeln.
- Bei der Kodegenerierung wird automatisch ein Makefile erstellt, das jedoch für den Imote2 angepasst werden muss. Da das Makefile bei jeder neuen Kodegnerierung überschrieben wird, empfiehlt es sich sich eine Datei `Makefile.custom` anzulegen. In dieser Datei kann man Imote2 spezifische Einstellungen speichern. Folgende Einstellungen müssen vorgenommen werden.

  ```
  CPP = xscale-elf-g++
  LINKEROPTIONS = $(LOPTIONS) -nostartfiles -Timote2.ld
  ADDLINK += asms.o crtend.o crtn.o crtbegin.o crti.o
  ```

Die erste Zeile setzt den Compiler für die XScale-Architektur, die zweite setzt das Linkerskript und die dritte übergibt dem Linker die zusätzlich benötigten Objektdateien. Es wird in diesem Beispiel davon ausgegangen, dass sich das Linkerskript und die Objektdateien im selben Verzeichnis wie die Datei `Makefile.custom` befindet. Ist dies nicht der Fall, müssen die Pfade entsprechend angepasst werden.

- Nun kann das System mit dem Aufruf `make` compiliert werden. Sollte dabei eine Fehlermeldung wie

  ```
  cc1plus: /cygdrive/e/wasabi_drops/wasabi031117/install031117/include/c++
  /Wasabi-3.3.1: No medium found
  ```

  auftauchen, so muss man eine CD in das CD-Laufwerk einlegen, danach sollte das System compilieren.

- Der Compiler erstellt die ausführbare Datei `component`. Diese muss vor der Programmierung des Imote2 noch ins Binärformat umgewandelt werden. Dies geschieht mit dem Befehl:

  ```
  xscale-elf-objcopy -O binary component component.bin.out
  ```

  Die Binärdatei, die auf dem Imote2 geladen wird, ist `component.bin.out`.

- Nun wird der Imote2 mit einem USB-Kabel mit dem PC verbunden. Dann wird das Programm USBLoader (befindet sich ebenfalls im Imote2-Ordner) gestartet.

  ```
  USBLoaderHost.exe -p component.bin.out
  ```

  Durch fortlaufende Punkte zeigt das Programm an, dass es auf dem USB-Host nach einem Imote2 sucht. Nun kann der Imote2 gestartet werden. Der USBLoader erkennt den Imote2 (Ausgabe DEVICE DETECTED) und lädt das erstellte Programm auf den Imote2. Der vollständige Vorgang inklusive der Verifikation des geladenen Binärprogramms ist abhängig von der Größe des Programms. Für ein 500 kB großes Programm dauert der Vorgang etwa eine Minute. Es kann passieren, dass zwar die Meldung DEVICE DETECTED erscheint, danach aber nichts passiert. In diesem Fall muss der Imote2 neu gestartet werden. Dies geschieht entweder durch Unterbrechung der Stromversorgung oder durch ca. 5 sekündiges Drücken des Reset-Knopfes. Danach wird der Uploadvorgang gestartet.

- Ist der Ladevorgang abgeschlossen, beendet sich der USBLoader automatisch, der Imote2 startet sich selbständig neu und führt nun das geladene Programm aus.

Literaturverzeichnis

[1] *ARM Ltd, Milestones*. http://www.arm.com/aboutarm/milestones.html.

[2] *ARM Ltd, Processor Families*. http://www.arm.com/products/CPUs/families.html.

[3] *Blink example application for TinyOS*. http://www4.ncsu.edu/~ptkampan/files/BlinkI.zip.

[4] Crossbow: *MICAz - Data Sheet*. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[5] Dialog: *DA9030 Preliminary Product Data Sheet*, 2005. http://enaweb.eng.yale.edu/drupal/system/files/DA9030-DS07-0501.pdf.

[6] Fliege, I., A. Geraldy, S. Jung, T. Kuhn, C. Webel und C. Weber: *Konzept und Struktur des SDL Environment Frameworks (SEnF)*. Techn. Ber. 341/05, TU Kaiserslautern, 2005.

[7] Fliege, I., R. Grammes und C. Weber: *ConTraST - A Configurable SDL Transpiler And Runtime Environment*. In: Gotzhein, R. und R. Reed (Hrsg.): *System Analysis and Modeling: Language Profiles, Lecture Notes in Computer Science 4320*, S. 216–228. Springer, 2006.

[8] *GCC Toolchain for Intel XScale Processor*. http://enaweb.eng.yale.edu/drupal/files/resources.htm.

[9] *Imote2 Yahoo Group*. http://tech.groups.yahoo.com/group/intel-mote2-community/.

[10] Intel: *XScale Microarchitecture - Programmers Reference Manual*, Februar 2001. http://download.intel.com/design/intelxscale/27343601.pdf.

[11] Intel: *PXA27x Processor Family Electrical, Mechanical, and Thermal Specification*, 2005. http://www.xscale-freak.com/XSDoc/PXA27X/28000304.pdf.

[12] Intel: *PXA27x Processor Family - Developer's Manual*, Januar 2006. http://enaweb.eng.yale.edu/drupal/system/files/PXA27x_Developers_Manual.pdf.

[13] *Intel Mote 2 - Engineering Platform Data Sheet Rev 2.1*, 2006. http://tech.groups.yahoo.com/group/intel-mote2-community/files/.

[14] *JFlashmm for Imote2*. http://www.xbow.com/Support/wobjectDetail.aspx?id=50160000000KXVYAA4&type=Solution&page=0.

[15] Lea, D.: *A Memory Allocator*. http://gee.cs.oswego.edu/dl/html/malloc.html.

[16] *SDL Forum*. http://www.sdl-forum.org/SDL/index.htm.

[17] Shahabdeen, J. A.: *Boot Loader Architecture*, 2005. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-1.x/contrib/imote2/tools/src/bootloader.doc?revision=1.1.

[18] *Special sections in the ELF-Format*. http://www.linux-foundation.org/spec/refspecs/LSB_1.3.0/gLSB/gLSB/specialsections.html.

[19] *Telelogic Tau SDL Suite*. http://www.telelogic.com/products/tau/sdl/index.cfm.

[20] Texas Instruments: *CC2420 Datasheet Rev SWRS041b*, März 2007. http://focus.ti.com/lit/ds/symlink/cc2420.pdf.

[21] *TinyOS*. http://www.tinyos.net/.

[22] *Using ld, the Gnu Linker*. http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/gnu-linker/index.html.

[23] *Wasabi Toolchain for Intel XScale Processor*. http://www.intel.com/design/intelxscale/dev_tools/031121/wasabi_031121.htm.

[24] Wasabi: *Wasabi Software Developement Tools User's Guide for Intel XScale Microarchitecture*, März 2004. http://download.intel.com/design/intelxscale/dev_tools/031121/Wasabi_XScale_Users_Guide1.pdf.

[25] Weber, C.: *Entwurf und Implementierung eines konfigurierbaren SDL Transpilers für eine C++ Laufzeitumgebung*. Diplomarbeit, TU Kaiserslautern, Fachbereich Informatik, 2005.