Dissertation

Modellgetriebene Entwicklung von Kommunikationsprotokollen für drahtlos vernetzte Regelungssysteme

Vom Fachbereich Informatik der
Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
„Doktor der Ingenieurwissenschaften“ (Dr.-Ing.)
genehmigte Dissertation von

Dipl. Inf. Marc Krämer

23. April 2013

Wissenschaftlichen Aussprache:	3. Juli 2013
Dekan:	Prof. Dr. Arnd Poetzsch-Heffter
Promotionskommission:	
Vorsitzender:	Prof. Dr. Klaus Schneider
Berichterstatter:	Prof. Dr. Reinhard Gotzhein
	Prof. Dr.-Ing. Jens B. Schmitt

D 386

Zusammenfassung

Funkvernetzte Sensorsysteme sind heutzutage allgegenwärtig. Sie werden sowohl in Rauchmeldern, in Raumtemperaturüberwachungen und Sicherheitssystemen eingesetzt. Das Sensorsystem soll seine Aufgabe zuverlässig und über viele Jahre ohne Batteriewechsel erfüllen. Durch die Vernetzung der Sensorsysteme und ihre immer komplexer werdenden Aufgaben wird die Programmierung in einer maschinennahen Sprache immer aufwändiger. Die modellgetriebene Entwicklung erhöht die Wartbarkeit und reduziert die Entwicklungszeit wodurch im Allgemeinen die Produktqualität steigt. In Folge der höheren Komplexität, der Abstraktion von der konkreten Hardwareplattform und den immer kürzere Produktentwicklungszeiten bleibt oft keine Zeit für Energieoptimierung, wodurch die Batterielaufzeit geringer ausfällt, als dies möglich wäre.

In dieser Arbeit werden verschiedene Ansätze vorgestellt, die es ermöglichen, bereits während der Modellierung den Stromverbrauch zu berücksichtigen und diesen zu optimieren. Am Beispiel des inversen Pendels, einem sehr instabilen Regelungssystem, wird dazu mit Hilfe der modellgetriebenen Entwicklung eine funkvernetzte, verteilte Regelung spezifiziert. Der aus der Spezifikation erzeugte Kode wird direkt auf den Sensorknoten ausgeführt und muß dazu performant und zuverlässig sein, um die Echtzeitanforderungen des Regelungssystems zu erfüllen, aber gleichzeitig so wenig Energie wie möglich zu verbrauchen. m die Zuverlässigkeit der verteilten Regelung zu gewährleisten ist eine deterministische kollisionsfreie Datenübertragung über das drahtlose Kommunikationsmedium erforderlich. Die Synchronisation ist eine weitere Voraussetzung zur Ermittlung eines konsistenten Systemzustands.

Inhaltsverzeichnis

1

Kapitel 1.

Einleitung

Motivation

Mikrocontroller sind aus dem Alltag nicht mehr wegzudenken. Längst haben sie viele Bereiche des alltäglichen Lebens erobert. Sie werden immer kleiner und sparsamer, wodurch sich immer neue Anwendungsgebiete erschließen. Sie kommen in der Hausautomation in Rauchmeldern, elektronischen Heizungsthermostaten, Mobiltelefonen und Autos vor und werden mittlerweile sogar in Kleidung integriert. Ihre Aufgabe besteht darin, Meßwerte zu erfassen, diese zu verarbeiten und an andere Systeme zu melden oder mit den Meßwerten direkt lokal zu regeln. Doch auch im Bereich der Medizingeräte kommen Mikrocontroller zum Einsatz, beispielsweise in Herzschrittmachern, Infusionspumpen und Defibrillatoren. Neben den hohen Anforderungen bei der Entwicklung stehen besonders im Bereich der Medizingeräte Zuverlässigkeit, Echtzeitfähigkeit und eine lange Lebensdauer im Vordergrund. Betrachtet man das Beispiel des Herzschrittmachers, wird deutlich, daß ein Batteriewechsel aufwändig ist und die Batterie nicht sehr groß sein kann. Damit dieses Gerät lange seinen Dienst verrichten kann, muß die Elektronik besonders energiesparend sein. In Bezug auf den Mikrocontroller bedeutet dies, daß die darauf laufende Software so entworfen werden muß, daß sie wenig Ressourcen benötigt. Um Energie einzusparen und die vorhandene Energie möglichst gut ausnutzen zu können, muß zunächst bekannt sein, wie die verwendete Batterie oder der verwendete Akku funktioniert. Um gezielt den Verbrauch optimieren zu können, ist ein Modell des Energieverbrauchs der verwendeten Hardware nötig. Nur in wenigen Ausnahmefällen wie beispielsweise beim Herzschrittmacher wird der Energieverbrauch direkt beim Entwurf berücksichtigt. Bei den meisten Anwendungen erfolgt die Optimierung erst im Anschluß an die Entwicklung und unter großem Aufwand. Durch Analyse von Leerlaufzeiten kann die Energieeinsparung automatisiert während der Laufzeit erfolgen und in einen Energiesparmodus geschaltet werden. Doch nicht für alle Komponenten ist diese Analyse möglich bzw. liefert optimale Ergebnisse, weshalb bereits in der Spezifikation auf den Energieverbrauch geachtet werden muß.

Ist der Funktionsumfang eines eingebetteten Systems klein, stellt die Programmierung des Systems in hardwarenahen Sprachen wie Assembler und C kein Hindernis dar. Steigt jedoch die Komplexität des Systems, werden moderne Entwicklungskonzepte wie die objektorientierte Programmierung und der modellgetriebene Entwicklungsansatz immer wichtiger, um Fehler bei der Umsetzung aus dem Modell in das fertige Produkt zu minimieren und die Wartbarkeit zu verbessern. Bei der Entwicklung von Kommunikationssystemen hat sich die Sprache SDL mit der grafischen Modellierung von Kommunikationsautomaten etabliert. Durch die Werkzeugunterstützung für SDL kann aus dem Modell mittels Kodegeneratoren und einem Compiler direkt ein ausführbares Programm erzeugt werden. Da Regelungen auf eingebetteten Systemen meist reaktive Systeme sind, die sich gut auf Automaten abbilden lassen, liegt es nahe, diese Funktionalität zusammen mit dem Kommunikationssystemen in SDL zu spezifizieren.

Durch den gemeinsamen Entwurf von Regelung und Kommunikationssystem können die Anforderungen der Regelung an das Kommunikationssystem berücksichtigt und Garantien zur Zuverlässigkeit gegeben werden. Zur Untersuchung dieser Frage wurde das inverse Pendel ausgewählt, das in der Regelungstechnik bekannt und als Referenzsystem anerkannt ist. Werden die Sensoren des inversen Pendels über eine drahtlose Kommunikation mit der Regelung verbunden, ergeben sich hohe Anforderungen an die Zuverlässigkeit des Kommunikationssystems, um eine gute Regelungsgüte zu erreichen. Neben der Zuverlässigkeit des Mediums müssen alle Sensorwerte synchron ausgelesen werden, um einen konsistenten Systemzustand zu erhalten. Die Güte der Regelung hängt somit nicht mehr nur allein von dem Regelungsalgorithmus und der Leistung des Sensorknotens ab, sondern zunehmend auch von einer darauf abgestimmten Kommunikationsarchitektur. Am Beispiel des inversen Pendels wird in dieser Arbeit der Entwurf einer solchen Kommunikationsarchitektur gezeigt.

Struktur der Arbeit

Kapitel 2 Als erstes erfolgt eine Einführung in die Thematik, wozu zunächst die *Specification and Description Language* (SDL) und der darauf aufbauende modellgetriebene Entwicklungsansatz vorgestellt wird. In dieser Arbeit wird die Sprache SDL verwendet, um Protokolle und Anwendungen in einer objektorientierten Sprache zu spezifizieren. Aus der Spezifikation lassen sich mittels Kodegeneratoren direkt ausführbare Modelle generieren, die auf einer eingebetteten Hardwareplattform ausführbar sind. Als Beispiele für funkvernetzte Sensorknoten werden in diesem Kapitel die MICAz- und die Imote2-Plattform vorgestellt und miteinander verglichen. Es folgt eine kurze Einführung in die netzwerkbasierte Regelung und die Beschreibung des inversen Pendels.

Kapitel 3 Sensorknoten werden meist mit Batterien betrieben und sind darauf ausgelegt, über lange Zeit Sensorwerte aufzunehmen, bevor die Energie ihrer Batterien aufgebraucht ist. Mit den Energiemodellen zu den beiden Sensorplattformen wird hier ein Überblick darüber gegeben, wie sich der Energiebedarf der einzelnen Komponenten der Sensorknoten verhält. Für die Planung von Energie ist es jedoch erst einmal nötig zu verstehen, woher die Energie zur Versorgung des Sensorknotens kommt. Abhängig davon, ob es sich bei der Energiequelle um eine Batterie oder einen Akku, können die Art der Stromentnahme und die äußeren Bedingungen Auswirkungen auf die Gesamtkapazität haben.

Basierend auf den vorher gewonnenen Erkenntnissen werden Verfahren zur Energiekontrolle einer Sensorplattform innerhalb von SDL vorgestellt. Dabei werden zum einen manuelle und zum anderen automatische Kontrollmechanismen vorgestellt und deren Grenzen besprochen.

Eine weitere Möglichkeit, Energie zu sparen, besteht darin, die Präzision von Timern zu reduzieren und diese so zu parametrieren, daß diese innerhalb eines Jitter-Intervalls auslösen. Der hieraus gewonnene Freiheitsgrad kann dazu verwendet werden, einzelne Tasks direkt hintereinander auszuführen und damit längere Zeit in einem Energiesparmodus zu bleiben. Hierfür ist eine Anpassung des SDL-Schedulers nötig, der ebenfalls in diesem Kapitel beschrieben wird.

Kapitel 4 Neben der funktionalen Spezifikation enthalten SDL-Systeme meist auch Testroutinen und Ausgaben zur Fehlersuche. Der generierte Kode enthält ebenfalls diese Testroutinen, wodurch der Speicherbedarf wächst und die Ausführungsgeschwindigkeit sinkt. Aufgrund fehlender Separierungsmechanismen lassen sich in SDL nur schwer Komponenten bilden, die unabhängig entwickelt und getestet werden können. In diesem Kapitel wird ein Konzept und ein hierfür entwickeltes Werkzeug vorgestellt, das SDL um diese Funktion erweitert.

Durch den Einsatz des Werkzeugs in SDL und damit der Entfernung nicht benötigten Kodes erhöht sich die Ausführungsgeschwindigkeit. Um jedoch Echtzeitsysteme in SDL spezifizieren zu können, fehlen Mechanismen zur Modellierung. Innerhalb der Spezifikation wird vielfach von der Zeit abstrahiert, was sich jedoch bei der Kodegenerierung auswirkt. In der Spezifikation wird die Zeit für die Signalerstellung und -zustellung nicht berücksichtigt, weshalb sich bei der Ausführung Probleme ergeben können, die in der Spezifikation nicht ersichtlich sind. Hierzu wird eine Spracherweiterung vorgestellt, die Signale zu einer vorher festgelegten Zeit zustellen kann, wodurch die Signalweiterleitung aus dem kritischen Pfad entfernt werden kann.

Kapitel 5 Das nicht-lineare Regelungssystem des inversen Pendels stellt in der Regelungstechnik eine Referenzgröße zur Beurteilung der Regelungsgüte und Zuverlässigkeit von Regelungen dar. Erfolgt die Regelung mittels funkvernetzter Sensorknoten, werden zusätzlich hohe Anforderungen an die Kommunikationsarchitektur gestellt. In diesem Kapitel werden eine dienstorientierte Middleware und eine MAC-Schicht auf Basis des modellgetriebenen Entwicklungsansatzes in SDL entworfen, die den zuvor ermittelten Anforderungen des inversen Pendels genügt.

Kapitel 6 Für die modellgetriebene Entwicklung einer Kommunikationsschicht für das inverse Pendel und die Implementierung auf Sensorknoten wurden Werkzeuge und Hardware benötigt, die zunächst entwickelt bzw. weiterentwickelt werden mußten. Diese Kapitel gibt einen Überblick über die Werkzeuge sowie Anpassungen, Erweiterungen und Optimierungen.

Kapitel 7 Das letzte Kapitel faßt diese Arbeit noch einmal kurz zusammen. Der Anhang enthält die Bildschirmausgabe des SDL-Testsystems zum Test des SDL-Transpilers, der zur Übersetzung von SDL-Kode in C++-Quellkode verwendet wurde. Die vollständige Konfiguration der MAC-Schicht für die einzelnen Knoten des inversen Pendels ist ebenfalls im Anhang enthalten.

2

Kapitel 2.

Grundlagen

In diesem Kapitel werden die in dieser Arbeit verwendeten Techniken und Grundlagen eingeführt, die zum Verständnis der folgenden Kapitel benötigt werden. Als erstes wird dazu die Sprache SDL und der *SDL-MDD*-Entwicklungsansatz (Kapitel 2.1) vorgestellt. Dabei werden auch die Werkzeuge *SPaSs*, *ConTraST*, *SEnF* und *SdlRE* kurz vorgestellt. Für die praktische Evaluation von Kommunikationsprotokollen werden Hardwarekomponenten benötigt, die frei programmierbar sind und deterministische (oder zumindest weitgehend reproduzierbare) Ergebnisse liefern.[1] PC-Plattformen bieten als Testplattformen zwar hohe Leistungsfähigkeit und Flexibilität, aufgrund der hohen System- und Peripheriekomplexität ist jedoch ein komplexes Betriebssystem erforderlich, dessen nebenläufig und (aus Sicht der Anwendung) indeterministisch arbeitenden Verwaltungsprozesse erheblichen Einfluß auf die Reproduzierbarkeit von Ergebnissen, insbesondere auch von Kommunikationsfunktionen, haben. Für das gewählte Anwendungsgebiet bietet sich aufgrund der Einbettung und der beschränkten Komplexität von Anwendungen und Kommunikationssystemen der Einsatz von Mikrocontrollern an, die mit sehr einfachen Echtzeit-Betriebsystemen oder gar mit in die Anwendung integrierten Betriebssystemfunktionen auskommen.

Hier haben sich Sensorknoten bewährt, die Mikrocontroller, Sensor- und Aktor-Schnittstellen und Kommunikationshardware integriert haben. Ich werde dazu die zwei von mir verwendeten Knotentypen, einen einfachen 8 Bit Sensorknoten (*MICAz*, (Kapitel 2.2.1), und einen sehr leistungsfähigen 32 Bit Sensorknoten, den *Imote2* (Kapitel 2.2.2) vorstellen.

Als Testanwendung für das Kommunikationssystem wurde ein Regelungssystem für ein „inverses Pendel“ gewählt (Kapitel 2.3). Bei dem inversen Pendel handelt es sich

---

[1] Eine präzise Reproduzierbarkeit ist aufgrund der im realen Test- und Einsatzumfeld unvermeidlichen Wert- und Zeit-Variationen von Sensor-Eingangsdaten und Kommunikationsstrecken im Allgemeinen nicht erreichbar. Werden Algorithmen realisiert, die bezüglich solcher Variationen stabil sind (ihr Verhalten bei kleinen Variationen also im Allgemeinen nicht vollständig ändern), so kann die geforderte Ausführungsplattform einen erheblichen Beitrag zur Reproduzierbarkeit von Testergebnissen leisten.

um einen Referenztest in der Regelungstechnik, der besonders hohe Anforderungen an das Kommunikationssystem stellt, um dieses Pendel stabil zu halten. Nach der Vorstellung des Pendels folgt ein Überblick über verschiedene Reglerarten und die Probleme, die bei der Vernetzung von Reglern auftreten können.

2.1. Modellgetriebene Softwareentwicklung mit SDL und SDL-MDD

Die Entwicklung von Software lässt sich im Wesentlichen in die Teilaufgaben Anforderungsanalyse, Modellentwicklung, Dokumentation, Kodeerzeugung und Testen strukturieren. Um diesen Aufgaben strukturiert ablaufen zu lassen, werden in der Softwareentwicklung verschiedene Prozessmodelle, wie das *Wasserfallmodell*, das *V-Modell (XT)*, aber auch der *modellgetriebene Ansatz* [13, 51, 92], eingesetzt. Alle Prozessmodelle versuchen dabei das Vorgehen bei der Entwicklung, der Änderung von Anforderungen und auch der Fehlerkorrektur möglichst exakt zu beschreiben. Jeder Entwicklungsprozess hat spezifische Schwerpunkte, z.B. die Nachverfolgbarkeit von Änderungen oder die Erstellung von Modellen und Ablaufdiagrammen. Die Auswahl eines Entwicklungsprozesses erfolgt anhand der Projektvorgaben des Kunden, den Besonderheiten des Anwendungsgebiets, den verwendeten Werkzeugen und den Erfahrungen und Vorlieben der Akteure. Für die Sprache SDL eignet sich aufgrund der modellnahen Spezifikationsweise der modellgetriebene Ansatz, der im Folgenden vorgestellt wird, sehr gut.

2.1.1. SDL

Die *Specification and Description Language* (SDL) [55] ist eine seit 1976 von der ITU (International Telecommunications Union) ursprünglich für den Bereich der Telekommunikation entwickelte Sprache zur Spezifikation und Beschreibung von verteilten Systemen. Die Sprache dient der Spezifikation von nebenläufigen, kommunizierenden, erweiterten endlichen Automaten. In SDL lassen sich durch die Erweiterung um das Konzept der Timer auch nicht-reaktive Systeme spezifizieren. In Abbildung 2.1 wird die grafische Repräsentation eines SDL Systems, bestehend aus zwei nebenläufigen Automaten (Prozessen), gezeigt.

Eine SDL-Spezifikation besteht aus einem oder mehreren SDL-Systemen, die über ihre Umgebung (Environment) miteinander verbunden sind. Die Umgebung besitzt

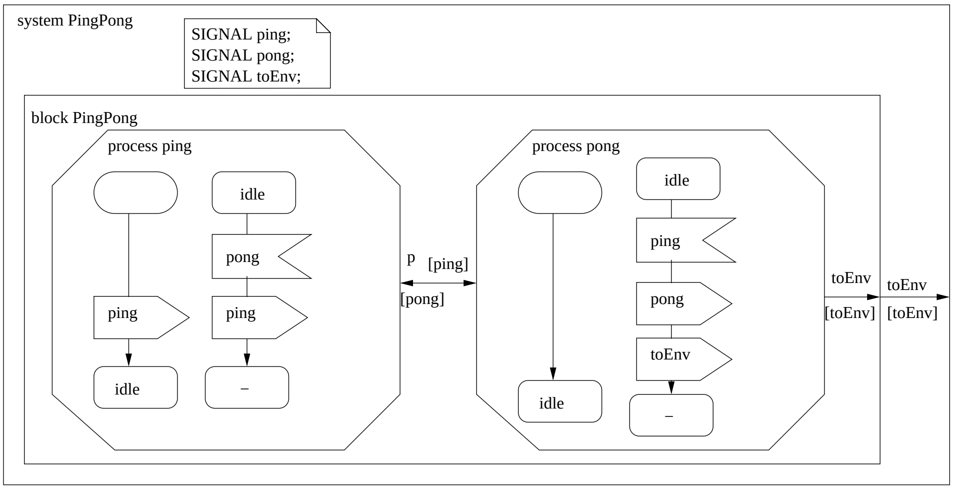


Abbildung 2.1.: Einfaches SDL-System

dabei die *Broadcast-Eigenschaft*, sodaß jedes System in der Umgebung jedes gesendete Signal empfangen kann. Signale, die zwischen den Systemen ausgetauscht werden sollen, werden somit mit Kanälen lediglich an die Systemgrenze geführt, bzw. von der Systemgrenze in das System. An jedem Kanal werden alle Signale, die von ihm transportiert werden können, spezifiziert. Der Kanal erhält dabei die Reihenfolge der empfangenen Signale (*First-In-First-Out*-Eigenschaft – FIFO). Innerhalb eines Systems werden die Signale über Kanäle zwischen *SDL-Blöcken* ausgetauscht. Ein Block stellt in SDL (bis SDL96) lediglich ein syntaktisches Strukturierungsmittel dar, er kapselt Signaldefinitionen und Variablendeklarationen, hat selbst jedoch keine semantische Funktion.[2] Jeder Block kann selbst entweder wieder Blöcke oder *SDL-Prozesse* enthalten. Ein Prozess ist die Beschreibung eines asynchron kommunizierenden erweiterten endlichen Automaten. Der Prozess besteht aus Zuständen und Transitionen die den Wechsel zwischen zwei Zuständen beschreiben. Eine Transition besteht aus einer Bedingung, einer Folge von Anweisungen und einem Folgezustand. Jeder Prozess hat zusätzlich einen speziellen Startzustand, der bei der Erzeugung des Prozesses aktiviert wird. Als Besonderheit besitzt die *Start-Transition* keine Bedingung, wird also sofort nach der Aktivierung des Startzustandes ausgeführt und der Prozess geht in den Folgezustand über. Sobald eine Bedingung einer Transition des aktuellen Zustandes erfüllt ist, kann diese Transition ausgeführt werden. Solange Signale in der Signalwarteschlange enthalten sind, wird die Ausführung von

---

[2] Seit SDL2000 stellen Blöcke selbst aktive Komponenten dar und besitzen einen Agenten der für die Ausführung zuständig ist. Im Unterschied zu Prozessen können innerhalb eines Blocks merhere Transitionen nebenläufig ausgeführt werden.

Transitionen wiederholt. Eine solche Bedingung ist der Empfang eines Signals oder das Ablaufen eines Timers.[3] Enthält die Signalwarteschlange Signale, die im aktuellen Zustand nicht empfangen werden können, werden diese aus der Warteschlange entfernt.[4] Neben der eigentlichen Syntax und den grafischen Symbolen, sind zu der Sprache SDL auch die Semantik und das dazugehörige Ausführungsmodell in der *SDL Virtual Machine* (SVM) beschrieben [56].

Betrachten wir als Beispiel die in Abbildung 2.1 dargestellte grafische Repräsentation eines einfachen SDL-Systems, in dem sich zwei Prozesse wechselseitig Signale zuschicken und ein Signal an die Umgebung gesendet wird. Auf System-Ebene werden drei verschiedene Signaltypen definiert. Das Signal `toEnv` wird von dem Block `PingPong` über den gleichnamigen Kanal an die Systemgrenze geleitet und damit an die Umgebung weitergegeben. Innerhalb des Blocks existieren zwei Prozesse `ping` und `pong`, die über den Signalweg `P` miteinander verbunden sind. Zusätzlich hat der Prozess `pong` noch einen Signalweg `toEnv`, der das gleichnamige Signal an die Blockgrenze weiterleitet. Bei der Initialisierung des Systems werden von beiden Prozessen die Starttransitionen ausgeführt, was bei Prozess `ping` das Aussenden des Signals `ping` an Prozess `pong` zur Folge hat. Danach wechselt der Prozess in den Zustand `idle` und wartet auf Ereignisse. In der Starttransition von Prozess `pong` gibt es nur die Anweisung in den Folgezustand `idle` zu wechseln. Ist das System initialisiert, werden alle Prozesse nebenläufig ausgeführt. Sobald der Prozess `pong` nun ein Signal in der Eingangswarteschlange enthält, das ebenfalls im aktuellen Prozesszustand konsumiert werden kann, wird die Transition zur Ausführung gebracht und die Signale `pong` und `toEnv` erzeugt. Das Signal `toEnv` wird zunächst zur Blockgrenze, dann zur Systemgrenze und schließlich in die Umgebung weitergleitet. Das Signal `pong` wird über den Kanal `P` an den Prozess `ping` geleitet, der das Signal konsumiert und erneut ein Signal `ping` versendet – und die Abfolge beginnt von vorne.

Die erste stabile Fassung von SDL wurde mit SDL'88 erreicht – es folgten Detailverbesserungen und Fehlerkorrekturen bis zur Version SDL'96. Obwohl die Sprache SDL auch danach noch kontinuierlich weiterentwickelt wurde, ist SDL'96 nach wie vor die am meisten genutzte Fassung. Sie wird, bis auf kleine Einschränkungen, von allen gängigen SDL-Werkzeugen unterstützt und auch Lehrbücher wie beispielsweise von Ellsberger et al. [31] beschränken sich meist auf diesen Standard. Die Weiterentwicklungen der Sprache zu SDL'2000 und SDL'2010 werden von den meisten Werkzeugen nur eingeschränkt unterstützt (beispielsweise RTDS von Pragmadev [89]). Grund hierfür ist, daß viele Werkzeughersteller mit dem Aufkommen und der

---

[3] Bei Ablauf eines Timers wird ein Signal mit dem Namen des Timers erzeugt und an die Eingangswarteschlange angehängt. Neben Signalempfang können auch *Enabling-Conditions* verwendet werden.

[4] Ein Save-Symbol kann Signale, die nicht verworfen werden sollen, in der Warteschlange an der Position erhalten.

steigenden Popularität von *UML* bzw. *UML 2* die Werkzeugentwicklung für SDL eingeschränkt haben.

Eines der wichtigsten Merkmale der Sprache SDL ist, daß neben der vollständigen syntaktischen Definition ein semantisch vollständiges Ausführungsmodell existiert, in dem auch Datentypen der Sprache beschrieben sind. Seit SDL2000 liegt das Ausführungsmodell von SDL als *Abstrakter Zustands-Automat* (ASM, *Abstract State Machine*) [15, 42] formal vor. Aufgrund der formalen Definition auf Basis der ASMs stehen SDL-Werkzeugen wie *Tau Suite* [52] oder RTDS [89] umfangreiche Analyse-, Simulations- und Kodegenerierungswerkzeuge zur Verfügung.

Da aktuell alle vorhandenen Werkzeuge als kommerzielle Closed-Source-Projekte entwickelt werden, ist eine Erweiterung und Weiterentwicklung der Sprache über eine direkte Erweiterung dieser Werkzeuge nur eingeschränkt und in direkter Kooperation mit den jeweiligen Herstellern möglich. Aufgrund dieser Beschränkungen wurden in der AG Vernetzte Systeme ein eigener Transpiler *ConTraST* [36] entwickelt, der aus einer SDL-Spezifikation C++-Kode generiert und dadurch schließlich ein eigenständig lauffähiges Programm erzeugen kann. Hiermit ist es möglich, Erweiterungen der Sprache durchzuführen, wodurch sich beispielsweise einige Prozesse besser im Modell abbilden lassen bzw. besserer Kode entsteht.

2.1.2. SDL-MDD

Die Möglichkeit direkt aus dem Modell ausführbaren Kode zu generieren rückt das Modell in den Mittelpunkt der Entwicklung und stellt damit den Kernpunkt der *modellgetriebene Softwareentwicklung* (Model driven development – MDD) [13] dar. Die modellgetriebene Softwareentwicklung in SDL erfolgt dabei dem in Abbildung 2.2 skizzierten Prozess [45, 71]. Als *Message Sequence Charts* (MSC) werden dazu zunächst die Anforderungen an den Kommunikationsautomaten durch die Beschreibung von Anwendungsfällen (*Use-Cases*) beschrieben. Es handelt sich hierbei um ein von der Implementierung unabhängiges Anforderungsmodell (*CIM*, *Computation-Independent Model*).

Auf Basis dieser Anforderungsspezifikation erfolgt die Modellierung in SDL, welche weiterhin unabhängig von der verwendeten Plattform bleibt, also keine Details der verwendeten Technologie mit einbezieht und somit ein *Platform-Independent Model* (PIM) darstellt. Ein PIM läßt sich durch Analyse und Ausführung in Simulatoren bereits auf Fehler im Modell überprüfen und gegen die Testfälle aus dem CIM validieren.

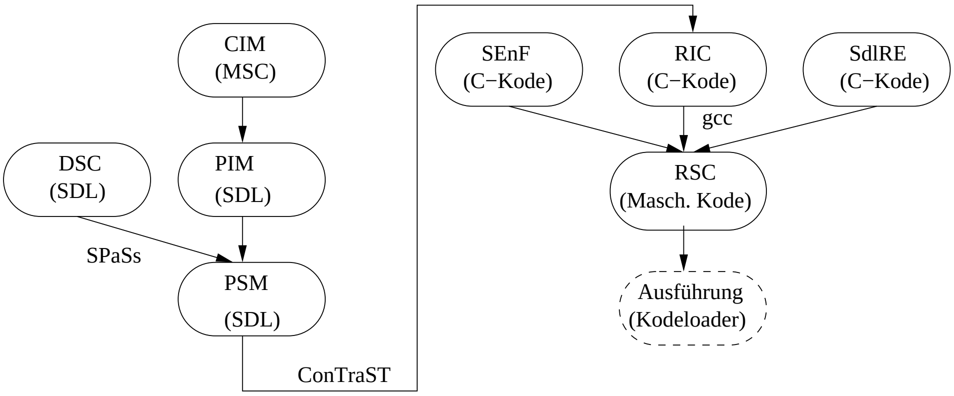


Abbildung 2.2.: SDL-MDD Entwicklung

Mit Hilfe des *Domain Specific Code* (DSC) wird das Modell an die spezifischen Eigenschaften der Domäne und der Plattform gebunden. Dieser Kode ist weiterhin in SDL spezifiziert, enthält aber bereits zeit- und hardwarespezifische Konstanten sowie abstrakte Systemschnittstellen der realen Umgebung. Durch Hinzufügen und den Austausch von Komponenten des PIM durch Komponenten des DSC entsteht das *Platform Specific Model* (PSM), das ebenfalls komplett aus SDL-Kode besteht.

Da die manuelle Transformation vom PIM zum PSM sehr aufwendig ist, ist dieser Schritt einerseits relativ fehleranfällig, andererseits kommt es in der Praxis meist dazu, dass spätere Entwicklungsiterationen nicht mehr auf PIM-Ebene, sondern direkt auf PSM-Ebene erfolgen, wodurch die Konsistenz der abgeleiteten Implementierungen mit PIM und CIM nicht mehr gewährleistet ist. Dies lässt sich durch eine Automatisierung der Transformation effektiv vermeiden. Dazu wurde das Werkzeug *SPaSs* [85] entwickelt, das eine vollständige Automatisierung der Transformation ermöglicht und somit eine effiziente Entwicklungsiteration über die CIM-Ebene unterstützt (siehe Kapitel 4.1).

Für das Ausführen des SDL-Modells auf einer realen Hardware-Plattform werden weitere Komponenten benötigt: Zunächst wird das SDL-Modell mit Hilfe eines in der AG Vernetzte Systeme entwickelten Transpilers *ConTraST*[36] (Kapitel 6.1) in laufzeitunabhängigen C++-Kode (*Runtime-Independent Code* (RIC)) umgewandelt. Die Implementierung *SdlRE* (Kapitel 6.1) enthält sowohl Datentypen und deren Operatoren, als auch Signalwarteschlangen und Komponenten zur Signalweiterleitung. Einen Scheduler / Dispatcher, der den nächsten SDL-Agenten zur Ausführung auswählt und startet, sowie die Implementierung der SDL-Agenten, die alle aktiven SDL-Komponenten steuern. Anpassungen an die spezifische Hardware, wie die Initialisierung oder die Kommunikation mit Betriebssystemtreibern, ist im *SDL Environment Framework* (SEnF) (Details siehe Kapitel 6.1.3) realisiert. Alle drei Komponenten zusammen (SEnF, SdlRE und das RIC) werden mit Hilfe eines plattformspezifischen Compilers zu einem auf dem Zielsystem lauffähigen Kode (*Runtime-Specific Code* (RSC)) übersetzt.

Durch die Werkzeugunterstützung ist dieser Ablauf weitgehend automatisiert, sodaß aus dem SDL-Modell (PIM) auf „Knopfdruck“ ein ausführbares Programm erzeugt wird. Der Entwicklung von Software auf Modellebene (SDL) und der damit verbundenen zusätzlichen Abstraktion innerhalb des Ausführungsmodells (SVM) wird oft unterstellt, zur Laufzeit einen zu großen Overhead zu haben, sodaß sich Systeme mit Echtzeitanforderungen nicht effizient damit realisieren lassen. In dieser Arbeit wird gezeigt, daß sich echtzeitfähige Systeme für Eingebette Systeme in SDL entwerfen lassen und diese sogar unter Verwendung vorhandener Informationen der SVM, Energie einsparen können.

2.2. Sensorplattformen

Für die Arbeiten im Bereich der Eingebetteten Systeme wurden zwei kommerzielle, auf dem Markt frei verfügbare Sensorplattformen der Firma *Memsic*[5] verwendet: Im Bereich der preiswerten und sehr sparsamen Sensorplattformen wurde der *MICAz*-Sensorknoten, basierend auf einem Mikrocontroller der Firma Atmel inklusive ZigBee-kompatiblem Transceiver[6], ausgewählt.

Aufgrund der beschränkten Ressourcen dieser Plattform und steigender Anforderungen durch neue Anwendungen und Kommunikationsprotokolle[7] wurde zusätzlich eine leistungsstärkere Plattform gewählt, die ähnliche Kommunikationseigenschaften wie die MICAz-Plattform besitzt. Die Wahl fiel auf die *Imote2*-Plattform, ebenfalls von der Firma Memsic, die einen schnelleren Mikrocontroller mit einer ARM-Architektur besitzt, jedoch den gleichen Transceiver-Chip wie die MICAz Plattform verwendet.

Beide Sensor-Plattformen können aufgrund der beschränkten herstellerseitigen Werkzeugunterstützung nur unter Windows [83] in einer *cygwin*-Umgebung [90] programmiert werden. Eigene Programme werden auf Basis des Sensorknoten-Betriebssystems *TinyOS* [107] sowie der darin enthaltenen *Tool-Chain* erstellt. Ob-

---

[5] ehemals CrossBow

[6] ZigBee ist ein drahtloses Kommunikationsprotokoll das für kurze Entfernungen entworfen wurde und auf dem IEEE 802.15.4 Standard aufbaut.

[7] wie beispielsweise MacZ [69]

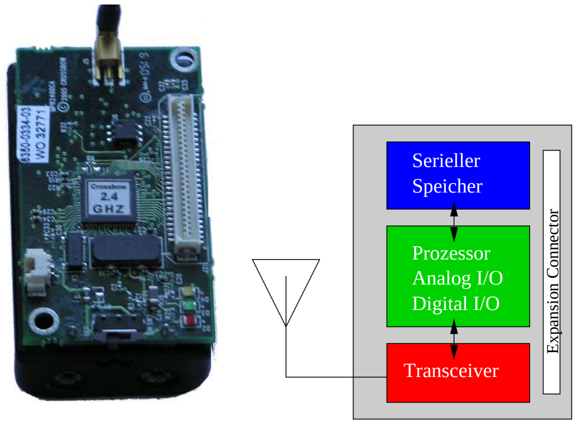
wohl es sich bei cygwin um eine Unix-artige Umgebung unter Windows handelte, existierte unter Linux für den Imote2 lange keine Möglichkeit, Kode auf das System zu übertragen. Erst die Entwicklung eines eigenen Boot- und Kodeladers ermöglichte auch hier eine plattformunabhängige Entwicklung, sowohl unter Windows, Linux, aber auch unter MAC-OS. Im folgenden werden die technischen Details beider Plattformen gezeigt, wobei die Aspekte des Energieverbrauchs detailliert in Kapitel 3.1 diskutiert wird.

2.2.1. MICAz

Die Sensorplattform MICAz [81] ist eine 8 Bit Sensorplattform, die mit 7,3728 MHz Taktfrequenz betrieben wird. Der darauf befindliche Atmel ATMega 128 [8] Mikrocontroller kann viele Befehle in nur einem Takt ausführen und kommt damit annähernd auf 7,3 MIPS (Million Instructions Per Second). Neben dem Mikrocontroller ist noch ein Transceiver, ein serieller Datenspeicher und ein Chip, der eine eindeutige ID liefert, auf dem Sensorknoten verbaut. Der Knoten und ein Blockschaltbild sind in Abbildung 2.3 dargestellt. Durch den auf dem Knoten verbauten Transceiver kann der Knoten zur Sensorwerterfassung und Weiterleitung an eine Datensenke verwendet werden. Steht keine Datensenke zur Verfügung, kann der ebenfalls verbaute serielle Datenspeicher verwendet werden, um Sensorwerte dauerhaft zu speichern. Diese Werte bleiben bis zum Löschen des Speichers, auch ohne Batterie, erhalten. Der Hersteller gibt für den Sensorknoten mit zwei Alkalinen Standard AA-Zellen unter Verwendung der Energiesparmodi des Knotens, eine maximale Laufzeit von einem Jahr an. Ich werde nun auf die einzelnen Komponenten des MICAz näher eingehen.

2.2.1.1. Mikrocontroller ATMega 128L

Der Mikrocontroller des MICAz ist mit 48 Eingabe-/Ausgabe-Pins (*GPIO*) ausgestattet, von denen 8 über einen Multiplexer als Analog-Eingänge an einen 10-Bit-Analog-Digital-Wandler geschaltet werden können. An einen der analogen Eingänge kann mittels Spannungsteiler direkt ein analoger Sensor, wie beispielsweise ein Lichtsensor oder ein Temperatur-Sensor, angeschlossen werden. Jeder GPIO-Pin kann, sofern er als Ausgang konfiguriert ist, kleinere Lasten wie eine *Leuchtdiode* (LED) direkt ansteuern. Auf der MICAz-Plattform sind drei dieser GPIO-Pins bereits mit verschiedenfarbigen LEDs beschaltet, die z.B. als Statusanzeige genutzt werden können. An vielen der GPIO-Pins stehen weitere spezielle Funktionenen, wie zwei *Universal Synchronous/Asynchronous Receiver Transmitter* (USART)-Controller, ein



(a) Bild eines MICAz (58 mm x 32 mm) (b) Blockschaltbild eines MICAz

Abbildung 2.3.: MICAz-Sensorknoten

Serial Peripheral Interface (SPI) und weitere zur Verfügung. Über diese Schnittstellen erfolgt die Kommunikation zwischen PC und Mikrocontroller, aber auch die Kommunikation mit weiteren Mikrocontrollern oder Peripheriegeräten. Der Flash-EEPROM (*Electrically Erasable Programmable Read-Only Memory*), in dem das Programm gespeichert wird, ist mit 128 Kibibyte[8] sehr großzügig bemessen und für typische Anwendungen eines 8 Bit-Mikrocontrollers ausreichend. Zusätzlich gibt es noch einen vom Programm unabhängigen 4 KiB großen EEPROM-Speicher, der beim Beschreiben des Programmspeichers nicht gelöscht wird, so dass hier beispielsweise langlebige Konfigurationsdaten abgelegt werden können. Der Hauptspeicher von 4 KiB ist zwar für einen 8 Bit-Mikrocontroller üppig bemessen, erweist sich bei Entwicklung komplexerer Kommunikations- und Verarbeitungsaufgaben aber schnell als limitierender Faktor.

2.2.1.2. Serieller Datenspeicher Atmel AT45DB041B

Der MICAz besitzt einen 528 KiB großen seriellen Datenspeicher von Atmel (AT45DB041B) [6]. Der Speicher ist in 2048 Seiten zu je 264 Byte organisiert. Schreibzugriffe erfolgen byteweise und werden zunächst in einem von zwei Puffern zwischengespeichert, bevor sie in den Datenspeicher übertragen werden. Das Löschen und Auslesen der Daten aus dem Speicher erfolgt jedoch immer seitenweise.

---

[8] 1 Kibibyte (KiB) = 1024 Bytes

2.2.1.3. Transceiver Texas Instruments CC2420

Für den Datenaustausch über Funk besitzt der MICAz einen CC2420 Transceiver [104] von Texas Instruments (ehemals Chipcon). Es handelt sich hierbei um einen Transceiver für das 2,4 GHz Band, der für die Kommunikation über den ZigBee-Standard [58] vorgesehen ist.

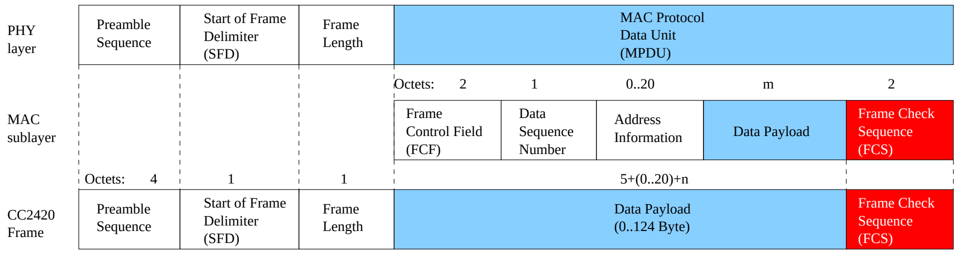


Abbildung 2.4.: IEEE 802.15.4 (Data) Frame Format

Der Chip eignet sich außerdem sehr gut für die Entwicklung eigener angepaßter Protokolle, da der Transceiver lediglich die physikalische Schicht zur Verfügung stellt, aber kein Protokoll implementiert. Dem ZigBee-Standard folgend werden die Symbole vom Transceiver selbst kodiert und dekodiert, zeitliche Abläufe und die maximale Länge des Datenrahmens folgen ebenfalls dem Standard. Der Aufbau des CC2420-Datenrahmens entspricht weitestgehend dem PHY Layer des ZigBee-Standards (siehe Abbildung 2.4). Der CC2420-Datenrahmen wurde lediglich um eine Prüfsumme des MAC-Sublayers ergänzt, die komplett in Hardware realisiert ist. Ohne weitere Anpassungen stehen somit 124 Byte Nutzdaten zur Verfügung. Durch weitere Konfigurationen kann die Präambel auf 2 Byte Länge verkleinert werden, wodurch der Rahmen zum einen nicht mehr standardkonform ist und durch die kürzere Präambel auch unzuverlässiger erkannt wird. Andere in der Umgebung verwendete Transceiver erkennen die Übertragung entweder gar nicht oder als Störung auf dem Medium.

Für die Entwicklung der *Black Bursts* zur Ticksynchronisation [72] ist ein dedizierter *Clear-Channel-Assessment* (CCA) Ausgang von entscheidender Bedeutung. Der Transceiver summiert hierbei kontinuierlich die Energie auf dem Trägerkanal über 8 Symbol-Perioden auf. Übersteigt die Summe eine festgelegte Grenze, wird der Kanal als belegt erkannt und dies über einen Pin signalisiert. Sobald die Summe wieder unter diese Grenze fällt, gilt der Kanal als frei, was ebenfalls über diesen Pin signalisiert wird. Sowohl der Empfang von Black Bursts, aber auch normale Datenrahmen, können so von dem Mikrocontroller auf Empfängerseite zeitlich sehr genau eingeordnet werden, womit eine hohe Synchronisationsgenauigkeit der beteiligten Stationen erreicht wird. Diese hohe Genauigkeit ist für MAC-Protokolle, die das *Time Divisi-*

on Multiple Access (TDMA) Verfahren verwenden, entscheidend, wie beispielsweise MacZ [69], um die nutzbare Kapazität des Mediums besser auszunutzen bzw. Kollisionen von Rahmen zu vermeiden.

2.2.2. Imote2

Der Imote2 Sensorknoten[80] ist ebenfalls von der Firma Memsic, ist aber deutlich leistungsfähiger als der MICAz. Seine Komponenten umfassen eine CPU, einen Transceiver und drei bzw. fünf LEDs.[9] Ein serieller Datenspeicher und analoge Eingänge fehlen bei diesem Sensorknoten. Die 32 Bit-CPU hat eine maximale Taktfrequenz von 416 MHz, einen 32 MB großen Hauptspeicher, sowie diverse ausschließlich digitale GPIOs. Trotz der höheren Leistung ist der Sensorknoten mit Maßen von 36x48 mm (siehe Abbildung 2.5) sogar kleiner als der MICAz. Der Transceiver des Imote2 ist, genau wie beim MICAz, ein TI CC2420 [104], wodurch beide Sensorknoten direkt per Funk miteinander kommunizieren können.

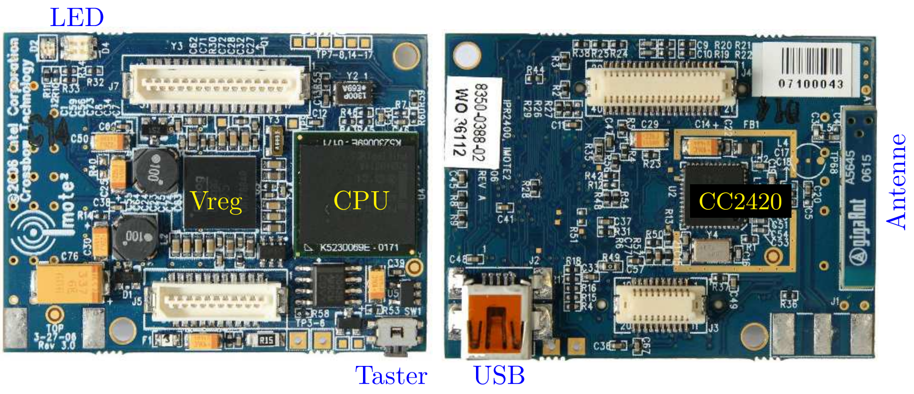


Abbildung 2.5.: Imote2 (48 mm x 36 mm)

CPU Marvell PXA271 Der Kern des Imote2 ist eine 32 Bit Marvell (ehemals Intel) XScale (PXA271) CPU [78]. Die XScale Architektur wurde von Intel basierend auf einem ARM-Kern der fünften Generation mit einem ARM-v5TE Befehlssatz [5] entworfen, wobei die Gleitkomma-Befehle entfernt wurden. ARM-Kerne sind *RISC*-Architekturen (*Reduced Instruction Set Computer*), die sich durch einen geringen

---

[9] 3 Leuchtdioden können direkt über die CPU gesteuert werden, 2 weitere Leuchtdioden sind nur über den Spannungscontroller steuerbar.

Energieverbrauch bei gleichzeitig hoher Leistung auszeichnen. Im Vergleich zu konventionellen 8-Bit Mikrocontrollern weist der XScale PXA271 viele Elemente auf, die in modernen Prozessorarchitekturen verwendet werden, verzichtet jedoch auf analoge Eingänge und integriert stattdessen viele digitale Busse. Folgende Merkmale sind für eine PC normal, unterscheiden jedoch einen gewöhnlichen 8-Bit-Mikrocontroller wie den ATMega 128, von einer ARM-Architektur:

- getrennter Daten- und Befehlscache (jeweils 32 kiB)
- Sprungvorhersage-Einheit (*Branch prediction*)
- MMU (*Memory Management Unit*)
- TLB (*Translation Lookaside Buffer*)
- DMA-Einheit (*Direct Memory Access*) mit 32 Plätzen für parallelen Transfer
- zwei Interruptlevel (normal/fast)

Die integrierten peripheren Anschlüsse, wie USB Host und Slave, I^2C (*Inter-Integrated Circuit*), I^2S (*Inter-IC Sound Interface*), SPI (*Serial Peripheral Interface*), Displaycontroller für mittelgroße Displays, sowie ausschließlich digitale Ein- und Ausgänge erklären sich auch aus dem vorgesehenen Einsatzgebiet in Handys und DVD-Playern. Die Ansteuerung der Peripherie erfolgt entweder direkt über den integrierten 13 MHz-Bus (direkter Zugriff auf die GPIOs bzw. die Register) oder erfolgt über einen DMA-Transfer, der auch komplexe Bedingungen und Schleifen enthalten kann. Das Schreiben des Programmspeichers erfolgt über die integrierte, aber langsame JTAG-Schnittstelle (*Joint Test Action Group*), die auch zu Debugging-Zwecken verwendet werden kann. Zum Zugriff von einem PC auf die JTAG-Schnittstelle wird ein spezieller Programmieradapter benötigt. Der Prozessor erlaubt zusätzlich den schreibenden Zugriff vom laufenden Programm auf den eigenen Programmspeicher, wodurch auch ein installierter Bootloader, wie er in Kapitel 6.4 beschrieben wird, diese Aufgabe übernehmen kann (Selbstprogrammierfähigkeit). Der Bootloader ermöglicht es dann, den Imote2 direkt über die USB-Schnittstelle zu beschreiben.

Bereits aus der Aufzählung der Komponenten wird deutlich, daß es sich bei der Imote2-Plattform, gegenüber dem MICAz, um eine deutlich leistungsfähigere Plattform handelt, deren Energieverbrauch auch über dem der MICAz-Plattform liegt. Demgegenüber erhält man im laufenden Betrieb eine deutlich höhere Rechenleistung bezogen auf den Energieverbrauch. Genaueres hierzu folgt in Kapitel 3.1.

Transceiver Der Imote2 besitzt den selben Transceiver wie der MICAz (siehe Kapitel 2.2.1.3). Im Unterschied zum MICAz kommt beim Imote2 eine auf dem Board direkt angebrachte Antenne zum Einsatz, Anschlüsse für eine externe Antenne sind nur durch Lötpunkte auf dem Board vorhanden.

2.2.3. Vergleich beider Plattformen

Ein kurzer tabellarischer Überblick über die wichtigsten Komponenten beider Plattformen:

	MICAz	Imote2
CPU	ATMega 128L	Memsic XScale PXA271
Wortbreite	8 Bit	32 Bit
SRAM	–	256 kiB
RAM	4 kiB	32 MiB
ROM	128 kiB	32 MiB
Cache	–	√ je 32 Zeilen für Daten/Befehle
DMA-Transfer	–	√ 32 Kanäle
TLB	–	√
EEPROM	4 kiB	–
GPIO	48	128
Analog-Input	8	–
Analog-Auflösung	10 Bit	–
Digitale IOs	UART, 2·SPI	2·UART, 3·SPI, I^2C, I^2S, USB
Interrupt-Level	1	2
Transceiver	802.15.4 (TI CC2420)	802.15.4 (TI CC2420)
Datenrate	250 kb/s	250 kb/s
Antenne	extern (SMA)	OnBoard (GigaAnt)
Maße	58 x 32 mm	48 x 36 mm
Gewicht	12 g	11 g
Gewicht Batterien	2·AA=55 g	3·AAA=49 g
Kapazität LiPo	1500 mAh	1200 mAh
Gewicht LiPo	35 g	33 g

2.3. Inverses Pendel

Für die Evaluation des modellgetriebenen Entwicklungsprozesses für drahtlose Regelungssysteme wurde der offene Regelkreis des stark instabilen inversen Pendels[10] gewählt [20, 21, 22].

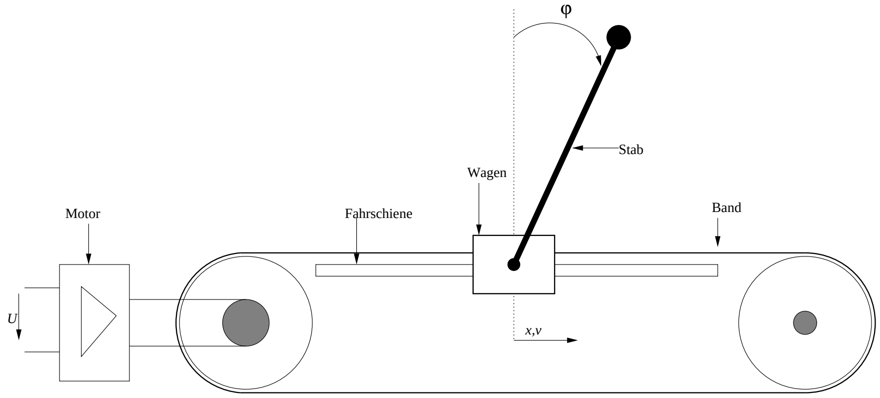


Abbildung 2.6.: Inverses Pendel

Wie in Abbildung 2.6 zu sehen ist, besteht das Pendel aus einer Schiene, auf der sich ein Wagen auf der x-Achse in beiden Richtungen bewegen kann. An dem Wagen selbst ist das Pendel, bestehend aus einem Stab mit einem Gewicht, in einem besonders reibungsarmen Lager befestigt. In der Ruheposition hängt das Pendel mit dem Gewicht nach unten ($\varphi = 180\,^\circ$). Durch das Anlegen einer positiven Spannung (U) wird ein Motor angetrieben, der Wagen über ein daran befestigtes Band nach rechts bewegt, das Anlegen einer negativen Spannung entsprechend nach links. Über die Höhe der Spannung läßt sich die Geschwindigkeit (v) und damit auch die Beschleunigung (a) steuern. Für die Regelung des Systems stehen die Sensoren Position (x), Winkel (φ) sowie die Geschwindigkeit (v) über einen Tachometer zur Verfügung. Mit der angelegten Spannung U, wird der Motor angesteuert, der dann den Wagen bewegt.

Für die Regelungstechnik ergeben sich an dem inversen Pendel zwei wesentliche Aufgaben: das Aufschwingen, um das Pendel aus der Ruhelage ($\varphi = 180\,^\circ$) in die invertierte Position ($\varphi = 0\,^\circ$) zu bringen und das Halten des Pendels in dieser instabilen Position. Bei dem Pendel handelt es sich um ein offenes, nicht minimalphasiges

---

[10] Im Rahmen eines DFG-Schwerpunktprogramms 1305 „Regelungstheorie digital vernetzter dynamischer Systeme“, Förderungsnummer LI 724/15-1 und GO 503-8-1.

Regelungssystem [76], dessen mathematisches Modell nicht linear ist. Störungen wirken hier von außen auf den Stab und die Masse und bringen es aus der quasi stabilen Lage in eine Fallbewegung ($\varphi \neq 0\,°$). Diese Fallbewegung muß durch eine Folge von Bewegungen des Wagens durch den Regler ausgeglichen werden, um das Fallen zu verhindern. Die Stabilisierung des Systems kann dabei mit verschiedenen Reglerarten erfolgen, wie PID-, Kaskaden-, Zustands- oder IMC-Regler. Sowohl die Wahl des Reglers als auch die Abtastung haben dabei entscheidenden Einfluß auf die Regelgüte und die Stabilität des Pendels. Ein mathematisches Modell des inversen Pendels ist in [20] gegeben.

2.4. Regelung

Eine Regelung besteht aus einer zu regelnden Strecke, Sensoren und Aktuatoren, sowie dem Regler selbst. Die Verschaltung der Komponenten erfolgt wie im Blockschaltbild in Abbildung 2.7 gezeigt. Ziel der Regelung ist es, einen vorher festgelegten Zielwert zu erreichen und diesen gegenüber Störeinflüssen konstant zu halten. Die Sensoren liefern dazu den aktuellen Ist-Wert der zu regelnden Größe, aufgrund dessen der Regler Stellwerte für die Aktuatoren berechnet und an diese weitergibt. Im Gegensatz zu einer reinen Steuerung können bei der Regelung die geregelten Größen direkt gemessen und direkt bei der Berechnung berücksichtigt werden.

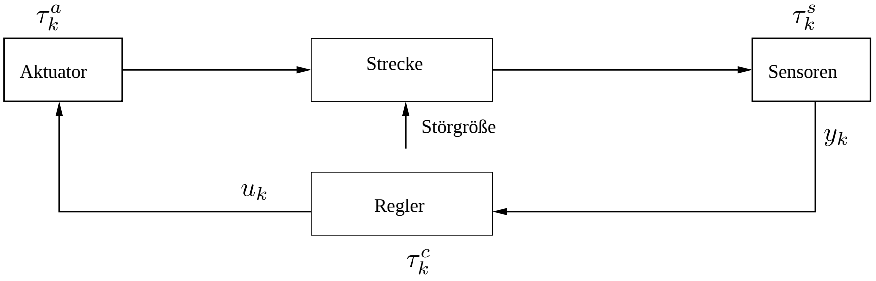


Abbildung 2.7.: Regelungssystem

Alle an einer Regelung beteiligten Komponenten Sensoren, Regler und auch Aktuatoren benötigen Zeit, entweder um die Werte zu ermitteln, um sie zu verarbeiten, oder auch bis der Stellwert an der Strecke anliegt. Diese Zeit wird als τ für jede der Komponenten angegeben und bezieht sich immer auf einen konkreten Zyklus k. Eine weitere Verzögerung bei der Regelung ist die Übertragungsverzögerung, die durch Kabel, Funkstrecken oder Vermittlungsstationen zwischen den Komponenten (y_k,u_k) entsteht. Bei digitalen Regelungssystemen müssen die Sensorwerte digitalisiert werden, wodurch eine diskrete Abtastung entsteht, die man als Abtastfrequenz bezeichnet.

2.4.1. Reglerarten

Um den Anforderungen der jeweiligen Regelungsstrecke gerecht zu werden, stehen in der Regelungstechnik verschiedene Regelerarten zur Verfügung. Es wird unterschieden zwischen Reglern, deren Parameter auf eine Strecke und einen Zielwert adaptiert werden und Reglern, die explizites Wissen über die zu regelnde Strecke benötigen. Zur ersten Gruppe gehören die Regler PI, PD und PID, wohin gegen zustands- und modellbasierte Regler Wissen über die Regelungsstrecke benutzen.

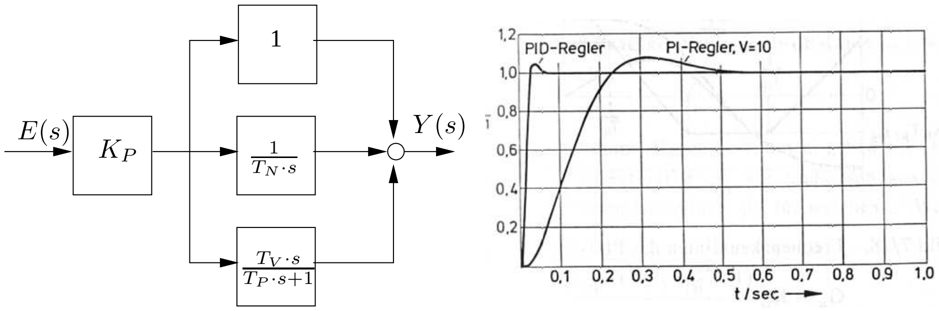
2.4.1.1. PID Regler

Der Proportional Integral Differential Regler stellt in der Regelungstechnik einen einfachen Standardregler dar, der durch folgende Übertragungsfunktion (aus [77]) beschrieben wird:

$$K(s) = k_P + \frac{k_I}{s} + k_D s \equiv k_P \left(1 + \frac{1}{T_I s} + T_D s\right)$$

Wie im Blockschaltbild des PID Reglers zu sehen ist (siehe Abbildung 2.8(a)), findet keine Rückführung der Stellgröße direkt an den Eingang statt. Die Stellgröße beeinflußt den PID-Regler nur über die an der Regelstrecke gemessenen Werte. Die Übertragungsfunktion setzt sich hierbei aus einem Verstärkungsfaktor (P) der gemessenen Eingangsgröße, einem integrierten (I) und einem differenzierten (D) Anteil zusammen. Daraus folgt, daß dieser Regler einfach zu realisieren ist, da für den Entwurf nur diese Parameter bestimmt werden müssen. Eine exakte Anpassung an die zu regelnde Größe (Regelungsmodell) ist nicht vorgesehen. Vereinfachungen des PID-Reglers sind PI- und PD-Regler, bei denen entweder der Differential- oder der Integral-Anteil entfallen ist. Im direkten Vergleich zwischen einem PID- und einem PI-Regler zeigt sich, daß der PID-Regler eine höhere Performanz erzielt, wie in Abbildung 2.8(b) zu sehen ist.

Auch wenn die Regelungsperformance von PI- und PD-Reglern unter der des PID-Reglers liegt, so ist ihr Vorteil eine geringere Rechenlast.



(a) Blockschema (b) Sprungantwort PI vs. PID-Regler (aus [40])

Abbildung 2.8.: PID Regler

2.4.1.2. Zustandsregler

Ein Zustandsregler regelt die Ausgangsgröße, anders als der PID Regler, aufgrund des aktuellen Systemzustands. Der Systemzustand wird zum einen über messbare Größen, aber auch über geschätzte Größen ermittelt. Die Systemordnung, die physikalisch vorgegeben ist, bestimmt die Anzahl der zu ermittelnden bzw. zu schätzenden Größen. In Abbildung 2.9 ist das Blockdiagramm eines Zustandsreglers gezeigt.

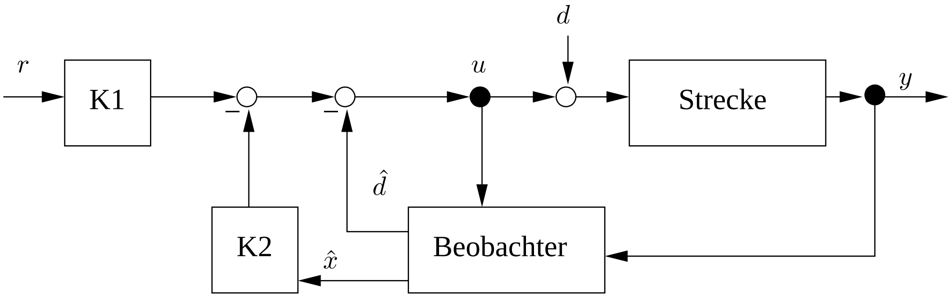


Abbildung 2.9.: Zustandsregler

Großer Vorteil des Zustandsreglers ist die bessere Regelungsperformance, die zu Lasten der Rechenlast geht, da für jeden Systemzustand ein Optimierungsproblem mittels LMI (linear matrix inequality) zu lösen ist. Die Struktur der linearen Matrix-Ungleichung ist hierbei über die folgende Formel gegeben:

$$\mathrm{LMI}(y) := A_0 + y_1 A_1 + y_2 A_2 + \ldots + y_n A_n \geq 0$$

Die Systemlast läßt sich reduzieren, indem diskrete Systemzustände bereits vorberechnet werden und in einer Tabelle auf dem Regler gespeichert sind. Problematisch ist dabei die Größe der Wertetabelle, die alle möglichen Systemzustände wiedergeben muß.

2.4.1.3. Modelbasierter Regler

Der IM-Regler (Internal Model Control) setzt ein genaues Modell des zu regelnden Systems voraus. Das Blockdiagramm in Abbildung 2.10 zeigt, daß die Stellwerte sowohl in das Modell als auch auf die Strecke geschaltet werden.

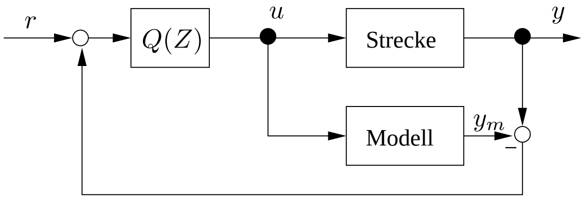


Abbildung 2.10.: IM-Regler (IMC)

Die Berechnung des Stellwertes wird hier nur über einen Filter eingestellt, der den Unterschied zwischen Strecke und Modell angleicht. Die Regelgüte und die Performanz des Systems ist hier stark von der Genauigkeit des Modells abhängig.

2.4.1.4. Kaskadenregler

Der Kaskadenregler (siehe Abbildung 2.11) ist kein eigenständiger Regler. Er beschreibt vielmehr die Kombination von mehreren Reglern, die eine Strecke gemeinsam regeln. Vorteil gegenüber einem einzigen PID Regler besteht in einer einfacheren Einstellung der Regelparameter für die Eingangsgröße. Abbildung 2.11 zeigt 3 kombinierte Einzelregler, die jeweils eine Größe regeln. Beim Kaskadenregler müssen die Typen der Einzelregler nicht gleich sein, so kann beispielsweise G_a ein P-Regler, G_b ein Filter und G_c ein PID Regler sein.

Kaskadenregler werden häufig auch dafür eingesetzt, um Performanz-Probleme von Einzelreglern in Grenzbereichen auszugleichen.

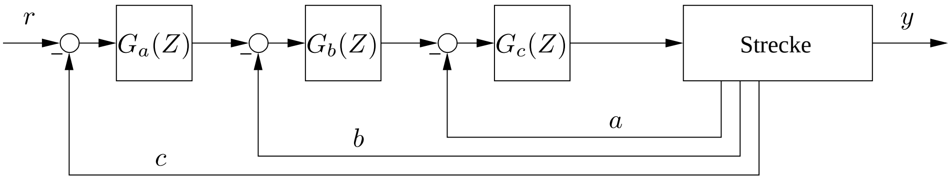


Abbildung 2.11.: Kaskadenregler

2.4.2. Netzwerk-Regelung

Unter einer Netzwerk-Regelung versteht man einen Regelkreis, bei dem der Regler nur durch ein Netzwerk mit den Sensoren und den Aktuatoren verbunden ist [27, 50]. In der Regelungstechnik ist es dabei verbreitet, das Netzwerk als Kasten zu modellieren, der die Eigenschaft hat, Rahmen zu verzögern und mit einer Wahrscheinlichkeit Rahmen zu verwerfen. Das Blockschaltbild, wie es in der Regelungstechnik verwendet wird, sieht dann aus wie in Abbildung 2.12. Die Verzögerungen der Sensoren, des Reglers und der Aktuatoren bleibt, wie bei einer herkömmlichen Regelung, gleich $(\tau_k^s, \tau_k^c, \tau_k^a)$. Die Verzögerung des Funknetzwerks wird dabei einmal mit τ_k^{sc} und τ_k^{ca} angegeben, wohingegen die Rahmenverluste als Schalter bei einer Gleichverteilung der Wahrscheinlichkeit von α_k bzw. β_k modelliert werden [75].

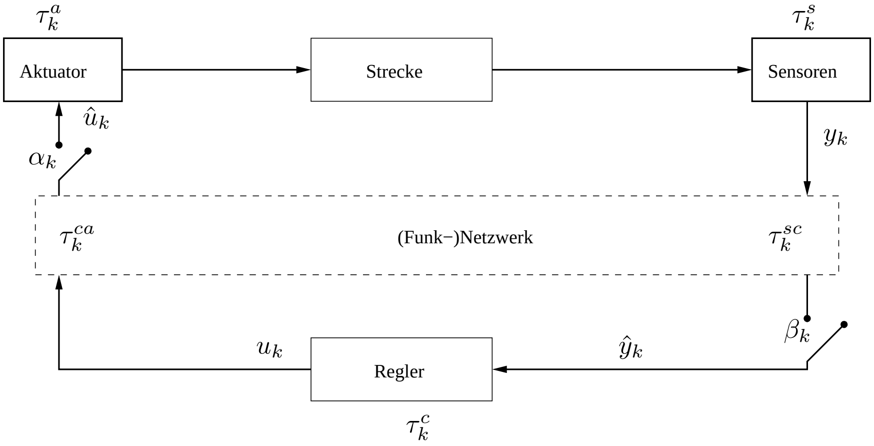


Abbildung 2.12.: klassische Ansicht einer Funknetzwerk-Regelung

Diese stark vereinfachte Sichtweise auf ein Netzwerk geht in keiner Weise auf Besonderheiten von Protokollen oder physikalische Eigenschaften ein. Bereits auf der physikalischen Ebene lassen sich verschiedene Fehlertypen unterscheiden, die sich in obiger Anschauung nur in der Ausfallwahrscheinlichkeit wiederfinden:

Verfälschung Durch fehlerhafte oder verrauschte Kommunikationsstrecke verursachte Bit-Fehler innerhalb eines Rahmens.

Verlust Entweder ist ein Bit-Fehler in der Präambel des Rahmens aufgetreten, sodaß der Empfänger die Übertragung nicht erkannt hat, oder der Amplitudenunterschied zwischen Signal und Rauschen (Rauschabstand) auf dem Kanal ist zu klein – meist durch Überschreitung der Sendereichweite oder aber auf dem Weg des Rahmens ist eine Weiterleitungsstation ausgefallen

Duplikate Verursacht durch Hardwarefehler einer Zwischenstation oder im Falle eines Funknetzwerkes durch Mehrwegeausbreitung

Reihenfolgenänderung Verursacht durch Hardwarefehler einer Zwischenstation, oder im Falle eines Funknetzwerkes durch Mehrwegeausbreitung

Alle genannten Fehler können sowohl in kabelgebundenen, aber auch in funkbasierten Netzwerken auftreten – die Häufung der jeweiligen Typen unterscheidet sich jedoch stark. In kabelgebundenen Netzwerken sind viele der Fehler auf fehlerhafte Hardware oder Implementierungsfehler zurückzuführen. Störeinflüsse auf verwendete Kabel sind bei geschirmtem und korrekt terminiertem Twisted Pair[11] sehr selten und können in der Regel vernachlässigt werden.

CSMA Verlust und Verfälschung treten in kabelgebundenen Netzwerken auch deshalb seltener auf, da über CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*)[100] die Leitung während einer Übertragung gleichzeitig mitgehört werden kann und überprüft wird das die Daten unverfälscht auf dem Medium liegen. Senden mehrere Stationen überlagern sich die Daten auf dem Medium und die Daten werden verfälscht. Mittels CSMA/CD erkennen die Sender die Kollision, beenden ihre Übertragung und versuchen die Übertragung zu einem späteren Zeitpunkt erneut.[12] Da die Kanalbelegung vor dem Senden geprüft wird, treten Kollsionen nur noch während der Bewerbung um das Medium auf. Um die Verfälschung von Nutzdaten zu vermeiden, sendet jede Station vor der Übertragung eine Präambel die so

---

[11] Hier werden 2 zusammengehörige Adernpaare miteinander verdreht verlegt, sodaß Störeinflüsse auf beide Leitungen gleichmäßig einwirken und somit aus dem Nutzsignal wieder herausgerechnet werden können. Die Terminierung bei TP erfolgt normalerweise direkt auf der Netzwerkkarte.

[12] Um die Wahrscheinlichkeit einer erneuten Kollision zu vermeiden warten alle Sender die gleichzeitig den Kanal belegen wollten eine zufällige Zeit, und belegen diesen danach, sofern er frei ist

groß gewählt ist, daß alle an einem Netzwerk beteiligten Stationen den Sendeversuch einer anderen Station erkennen, bevor die Nutzdaten übertragen werden.

In Funknetzwerken besteht jedoch keine Möglichkeit, den Funkkanal während einer laufenden Übertragung abzuhören. Zum einen besitzen Funktransceiver nur eine gemeinsame Sende-/Empfangseinheit, zum anderen ist die Signalstärke auf dem Medium abhängig von der Entfernung des Senders. Die Überlagerung gleichzeitig sendender Stationen geht in unmittelbarer Nähe zum Sender „im Rauschen unter“.[13]

Ohne eine Koordinierung des Mediums können Kollisionen von der sendenden Station nicht erkannt, sondern nur vermieden werden. Mit CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance)[100] wird das Medium vor einer Übertragung überprüft und sofern dieses als frei erkannt wird, die Daten versandt. Wird der Kanal als belegt erkannt, wird mindestens solange gewartet, bis der Kanal wieder frei ist. Damit nicht alle wartenden Sender gleichzeitig eine erneute Übertragung beginnen, wird der Sendewunsch um eine zufällig gewählte Zeit verschoben – dies senkt somit die Wahrscheinlichkeit für eine Kollision, schließt diese aber nicht aus. Ein weiteres Problem, wodurch es bei Funkempfängern leichter zu Kollisionen kommt, ist, daß die Umschaltung vom Empfangsmodus in den Sendemodus Zeit benötigt in der keine Überwachung des Mediums möglich ist. Wird von einer anderen Station in dieser Zeit eine Übertragung begonnen, kommt es zu einer Kollision, da der umschaltende Sender die Übertragung nicht mehr erkennt.

Hidden-Station-Problem Anders als in Kabelnetzwerken, können sich in Funknetzwerken nicht immer alle beteiligten Stationen gegenseitig hören. Es kann dabei das sogenannten *Hidden-Station-Problem* auftreten, bei dem ein Datenübertragung zwischen zwei Knoten von einem dritten Knoten nicht bemerkt wird, dieser den Kanal als frei erkennt und selbst eine Übertragung startet wodurch auch die erste Übertragung gestört wird. Damit es auch in Funknetznwerken nicht ständig zu Kollisionen kommt, kann CSMA/CA um das *RTS/CTS-Protokoll*[100] erweitert werden, bei dem ein Sender seinen Sendewunsch über einen RTS (*Request To Send*) ankündigt. Der Empfänger bestätigt bei positivem Empfang des RTS diesem mit einem CTS (*Clear To Send*). Durch den vorherigen Austausch dieser Daten sind alle Knoten in Empfangsreichweite der kommunizierenden Knoten über die Belegung des Mediums informiert und damit das *Hidden-Station-Problem* vermieden.

---

[13] Die Amplitude des Trägersignals des Senders ist so hoch, das die deutlich kleinere Amplitude anderer Sender keinen großen Einfluß auf das Signal in unmittelbarer Nähe hat.

3

Kapitel 3.

Energie

Die zunehmende Technisierung der letzten Jahre sorgt für einen stetig wachsenden Bedarf an Energie, die jederzeit und überall zur Verfügung stehen soll. Neuere Geräte versuchen stetig den Bedarf an Energie bei größerer Leistungsfähigkeit zu verkleinern. Im Bereich der Computer-Industrie passiert dies beispielsweise durch neue Herstellungsverfahren, die im Betrieb weniger Energie benötigen. Überprüft man die Aussagen der Hersteller, stellt sich oft heraus, daß der geringere Energie-Verbrauch sich oft nur auf die Instruktionen pro Leistung bezieht. Angaben zum Verbrauch im Ruhebetrieb (Standby) sind nur äußerst selten zu bekommen und müssen oft einzeln nachgeprüft werden. Energieeinsparung wurde lange Zeit nur als reines Hardwareproblem verstanden, erst durch das Aufkommen und die Verbreitung leistungsfähiger mobiler Geräte, wie PDAs, Handys, Smartphones, Laptops, die möglichst lange mit einer Batterieladung auskommen sollten, wurde auch in den verwendeten Betriebssystemen Unterstützung zum Energiesparen geschaffen.

Die Möglichkeiten zum systematischen Energiesparen sollen an dem in dieser Arbeit betrachteten Sensorknoten Imote2 (siehe Kapitel 2.2.2) gezeigt werden. Dazu werden zunächst Energie-Modelle von dem Sensorknoten benötigt, um die Möglichkeiten des Knotens bewerten zu können. Da es sich bei der Plattform um einen mobilen Sensorknoten handelt, der mit Batterien oder Akkus betrieben wird, stellt sich auch die Frage, wie sich seine Rest- bzw. Gesamtkapazität bestimmen läßt und ob die Spartechniken Einfluß auf den Akku haben. Außerdem werden in diesem Kapitel einige Techniken vorgestellt, mit deren Hilfe sich bereits während des Softwaredesigns Energiesparaspekte modellieren lassen.

Mit Hilfe des SDL-MDD-Ansatzes [45, 71] können sogar große Systeme entwickelt werden, die anschließend auf der Zielplattform ausgeführt werden. Es ist daher unerläßlich, bereits in SDL Möglichkeiten zu schaffen, den Energieverbrauch der Zielplattform zu kontrollieren. Um dies zu erreichen, werden zunächst in Kapitel 3.1 Modelle für den Energieverbrauch der beiden Sensorknoten MICAz und Imote2 gegeben. Messungen an modernen Lithium-Polymer-Akkus in Kapitel 3.2 zeigen, welche

Kapazität eines Akkus sich wirklich nutzen läßt. Die Ergebnisse dazu wurden auf einem Fachgespräch vorgestellt und als Technischer Bericht [65] publiziert. Aufbauend auf diesen Modellen zeigt Kapitel 3.3, wie sowohl eine explizite und eine implizite Steuerung des Energieverbrauchs erfolgen kann. Die Ergebnisse daraus wurden auf der internationalen SAM-Konferenz vorgestellt und veröffentlicht [47]. Abschließend diskutiert Kapitel 3.4 die Möglichkeit, die störende Eigenschaft *Jitter* als Energiesparaspekt nutzbar zu machen.

3.1. Energiemodelle

Für das systematische Einsparen von Energie ist es nötig, zunächst Energie-Modelle von der Hardware zu erstellen. Alle auf der Hardwarekomponente befindlichen Teilkomponenten müssen hierbei untersucht und ihre Energiezustände in ein Zustandsdiagramm übertragen werden. Bei den Zustandsübergängen von S_{vorher} nach S_{nachher} sind dabei die Umschaltzeiten (t_{Wechsel}) zu notieren. Den meisten Datenblättern ist nicht zu entnehmen, wie der Energiebedarf während dieser Zeit ist; sofern hier keine Angaben zu entnehmen sind, wird er mit $P = \max(P(S_{\text{vorher}}), P(S_{\text{nachher}})) \cdot t_{\text{Wechsel}}$ nach oben abgeschätzt.

Für das Erstellen eines Energiemodells werden zunächst immer die Datenblätter herangezogen, die einen Überblick über alle möglichen Betriebszustände und ggf. deren Energieverbrauch geben. Neben den Betriebszuständen werden dort Umschaltzeiten und Bedingungen für den Zustandswechsel angegeben. Komplexe Sensorknoten sollten jedoch per se nicht einfach nur als die Summe der Einzelkomponenten angenommen werden, da weitere elektronische Komponenten (Widerstände, Kondensatoren, Spulen, Spannungswandler, ...) benötigt werden; eine untere Abschätzung des Verbrauchs läßt sich hiermit jedoch angeben. Eine deutliche Verfeinerung der aus dem Datenblatt entnommen Werte kann durch eine Meßreihe erreicht werden, die einzelne Komponenten in die jeweiligen Betriebsmodi versetzen und den Stromverbrauch mißt.

Exemplarisch wird das Energiemodell des MICAz, eines leistungsschwächeren Sensorknotens (siehe Abbildung 3.1) gezeigt, danach wird das Modell des moderneren und leistungsfähigeren Imote2-Sensorknotens behandelt.

3.1.1. MICAz

Der MICAz (siehe Kapitel 2.2.1) ist ein kleiner Sensorknoten, der als CPU einen ATMega128L, einen CC2420 Transceiver und einen seriellen Datenspeicher AT45DB041B (serieller Flash) besitzt. Alle Komponenten sind für geringen Energieverbrauch ausgelegt. Das Energiemodell des MICAz-Knotens auf Grundlage der Datenblätter ist in Abbildung 3.1 gezeigt. Ein in dieser Arbeit verfeinertes Modell, basierend auf Messungen, wurde auf dem Fachgespräch „Drahtlose Sensornetze“ [65] veröffentlicht.

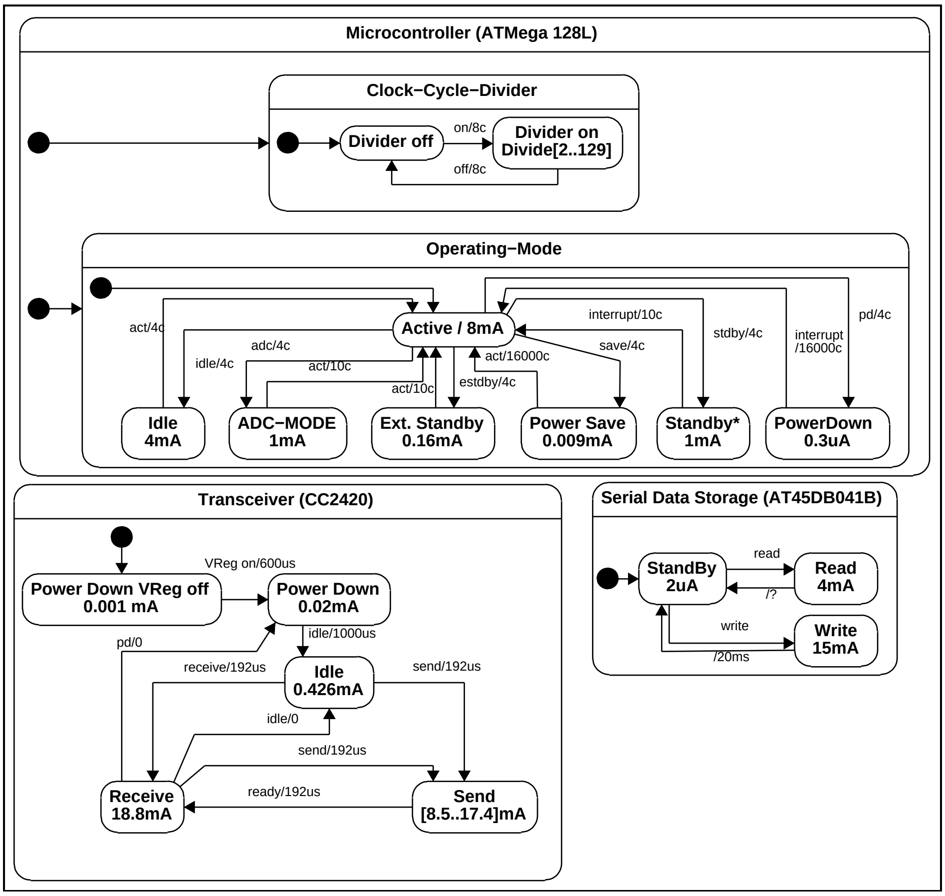


Abbildung 3.1.: Energiemodell des MICAz

Das Modell spiegelt die drei Komponenten Mikrocontroller, Transceiver und Serial Data Storage wieder. Der Gesamtenergiebedarf des Knotens wird aus der Summe aller drei Komponenten ermittelt. Jede Komponente befindet sich in einem Betriebsmodus, kann diesen aber wechseln, wodurch sich der Gesamtenergiebedarf ändert.

Mikrocontroller Der Mikrocontroller besitzt laut Datenblatt [8] für das Energiemodell zwei relevante Komponenten, den *Clock-Cycle-Divider* und den *Operating Mode.* Der Clock-Cycle-Divider benötigt selbst keine Energie, er steuert jedoch die Taktrate, mit der der Knoten betrieben wird und damit die Umschaltzeiten. Der Operating-Mode gibt den Zustand des Mikrocontrollers an: *Active, Idle, ADC-Mode, Extended Standby, PowerSave, Standby* und *PowerDown.* Nur im Active-Mode ist der Mikrocontroller voll betriebsbereit und arbeitet. Alle anderen Modi sind Schlafzustände, in denen jeweils einzelne Subkomponenten deaktiviert sind, um Energie einzusparen. Zustandswechsel des Mikrocontrollers können nur im aktiven Zustand durchgeführt werden, deshalb führen alle Zustandsübergänge wieder zum Active-Mode. Bei den tiefen Schlafmodi, wie PowerSave und PowerDown, stellt man fest, daß sie nicht nur viel Energie sparen, sondern auch viel Zeit benötigen, um aus diesem Modus wieder wach zu werden (16000 Zyklen). Bei einer Taktfrequenz von ∼8 MHz entspricht das immerhin 2 ms, bei geringerer Taktfrequenz entsprechend länger. Des weiteren zeichnen sich beide Modi auch dadurch aus, daß der Mikrocontroller nicht mehr von selbst aufwachen kann, sondern durch einen externen Interrupt (z. B. ein angeschlossener Taster) aufgeweckt werden muß. Zusätzlich werden alle internen Uhren gestoppt, was die mögliche Verwendung weiter einschränkt, da keine anderen kontinuierlichen Zeitgeber (beispielsweise eine Echtzeituhr) vorhanden sind.

Transceiver Der Transceiver CC2420 [104] ist sowohl auf dem MICAz als auch auf dem Imote2 verbaut. Es handelt sich hierbei um einen Transceiver für den IEEE 802.15.4 Standard [54], der die Bitübertragungsschicht (physical layer) implementiert, und den darauf aufbauenden ZigBee-Standard unterstützt, aber nur rudimentär implementiert. Der Transceiver selbst kann sich in einem der Modi *Power Down, Power Down VReg off, Idle, Send* und *Receive* befinden. Die Modi Power Down mit oder ohne Spannungsregulierung (VReg) finden nur bei sehr langen Abschaltphasen Verwendung, da nicht nur die lange Wiederanlaufzeit über die Zustandswechsel, sondern auch die völlige Neukonfiguration des Chips erfolgen muß. Somit bleibt in der Praxis meist nur der Idle Modus zum Stromsparen übrig, der den Stromverbrauch aber bereits drastisch reduziert. Zu beachten ist, daß der Transceiver ausschließlich im Receive Modus empfangsbereit ist, und auch nur dort Daten auf dem Medium erkennt und empfängt! Werden auf dem Sensorknoten keine angepaßten Protokolle verwendet, die dieser Tatsache Rechnung tragen und dauernd den Kanal abhören, benötigt der Transceiver mehr als doppelt so viel Energie wie der aktive Mikrocontroller! Auffällig ist auch, daß der Transceiver im Empfangsmodus deutlich mehr Energie benötigt als im Sendemodus. Dieses Phänomen tritt bei vielen komplexen Empfängern auf, deren Sendeleistung im Bereich einiger Milliampere liegt (wie beispielsweise [7, 103]). Grund hierfür ist, daß der Transceiver im Empfangsmodus das Signal verstärken, den Kanal abhören muß (mittels Filter und Demodulator), das Signal in einen Bitstrom verwandeln und mittels Komparatoren in dem Bitstrom nach der Präambel suchen muß – all das kostet Energie.

Serieller Datenspeicher Der serielle Datenspeicher [6] besitzt drei Betriebsmodi: *StandBy, Read* und *Write*. In unseren Arbeiten wurde der serielle Speicher wegen seines hohen Stromverbrauchs beim Schreiben nie verwendet und wird hier nur der Vollständigkeit aufgeführt. Für unsere Betrachtungen spielt nur der StandBy-Modus eine Rolle, und dieser hat, durch den geringen Verbrauch von $2\,\mu$A, in der Gesamtrechnung kaum Einfluß.

Gesamtenergiebedarf Ein MICAz-Sensorknoten unter Verwendung von TinyOS [107], einem populären Sensorknotenbetriebssystem, besitzt standardmäßig keine Energiekontrolle. Der Energiebedarf, bei aktiviertem Transceiver errechnet sich damit als $P = 8\,\text{mA} + 18.8\,\text{mA} + 0.002\,\text{mA} =\sim 26.8\,\text{mA}$. Wird der MICAz nun von 2 handelsüblichen Akkus mit je 2000 mAh versorgt, sind diese bereits nach $\frac{2\cdot 2000\,\text{mAh}}{26.8\,\text{mA}} = 149.3\,\text{h}$ – also nach 6,2 Tagen vollständig leer. In der Praxis wird dies noch früher passieren, da nicht die ganze Kapazität eines Akkus nutzbar ist und der Verbrauch des MICAz etwas höher liegen wird als die Daten aus dem Datenblatt, was in [65] gezeigt wurde.

3.1.2. Imote2

Der Imote2 ist, wie in Kapitel 2.2.2 bereits angesprochen, aus weniger Einzelkomponenten aufgebaut als der MICAz. Im Gegensatz zum MICAz handelt es sich hier aber um eine komplexere, stärker integrierte Plattform. In Abbildung 3.2 ist das aus den Datenblättern [26, 78, 104] ermittelte Energiemodell des Imote2 gezeigt. Als Einzelkomponenten sind hier die CPU inklusive Spannungscontroller und der Transceiver CC2420, der bereits beim MICAz eingesetzt wurde, vorhanden. Da Spannungscontroller und CPU direkt zusammenarbeiten und diese immer auf einander abgestimmt sind, werden beide zusammen als eine Komponente betrachtet.

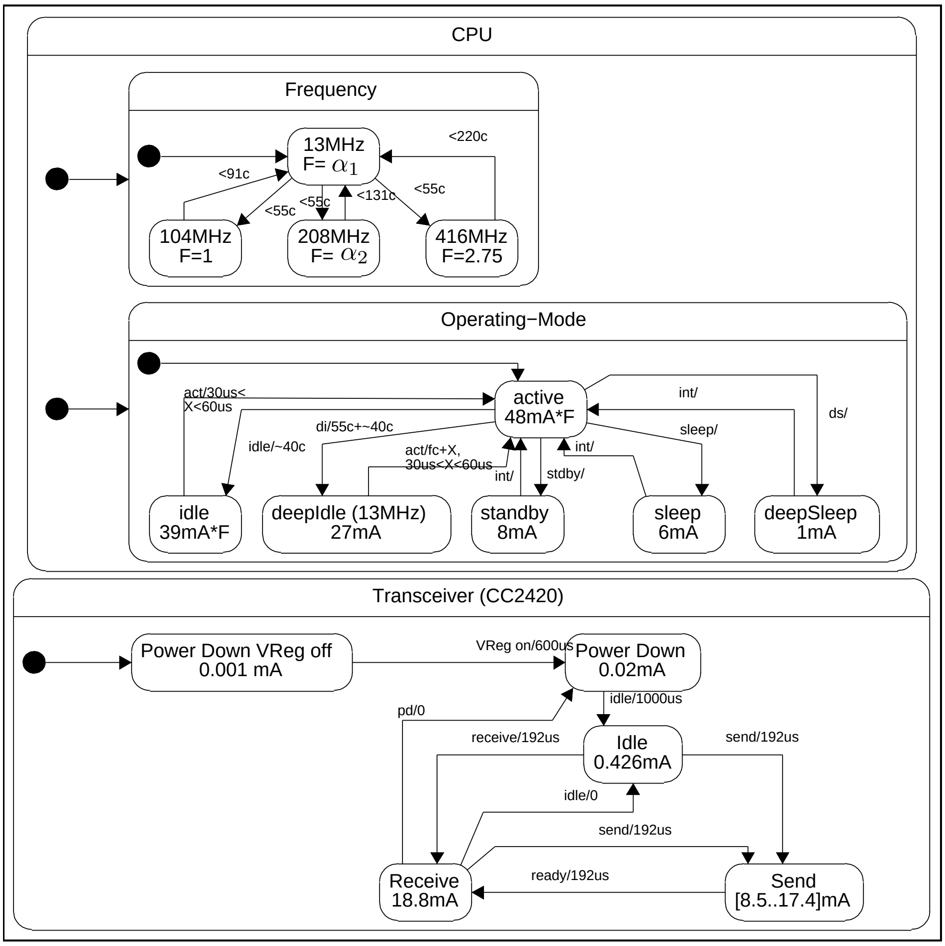


Abbildung 3.2.: Energiemodell des Imote2

CPU (PXA271) Wie beim ATMega128L besitzt auch die CPU des Imote2 verschiedene Betriebsmodi und Frequenzen. Die CPU kann hierbei in 4 Stufen zwischen 13 MHz, 104 MHz, 208 MHz und 416 MHz umgeschaltet werden, was auch den Energiebedarf des Prozessors beeinflußt. Der Operating-Mode kann in 6 relevante Modi geschaltet werden: *active, idle, deepIdle, standby, sleep* und *deepSleep*. Da der Taktgeber nur in den Modi active und idle aktiv ist, wird auch nur hier die Frequenz bei der Berechnung des Energiebedarfs berücksichtigt. Der deepIdle-Modus entspricht dem idle Modus mit einem festen Prozessor-Takt von 13 MHz. Neben den hier angesprochenen Modi besitzt der PXA271 weitere Schlafmodi (standby, sleep und deepSleep), die beim Aufwachen einem kompletten System-Neustart entsprechen, da bis auf 2 Register alle Einstellungen verloren gehen. Aufgrund dieser Einschränkungen wurden diese Modi nicht weiter berücksichtigt. Bei genauer Betrachtung von Abbildung 3.2 fällt auf, daß Zeitangaben für den Wechsel aus oder in bestimmte Modi nicht bzw. nur als Zeitspanne angegeben sind. Dies liegt zu großen Teilen daran, daß diese Angaben nicht dem Datenblatt [80, 78] oder anderen Quellen wie [112] zu entnehmen sind. Einige Zeiten werden in den Datenblättern auch nur als Ereignisse definiert, wie beispielsweise „nachdem der Cache geleert wurde“. Da diese Werte schwer zu ermitteln, bzw. abhängig von den verwendeten prozessorinternen Komponenten sind, sollten sie für das spezielle Anwendungsszenario exakt ermittelt werden.

Gesamtenergiebedarf Da es sich bei dem Imote2 selbst um eine deutlich leistungsfähigere Plattform als dem MICAz handelt, ist es nicht verwunderlich, daß der Gesamtenergiebedarf hier deutlich höher liegt. Trägt man der höheren Leistungsfähigkeit Rechnung und betreibt die CPU beispielsweise mit 104 MHz und legt sonst die gleichen Annahmen wie beim MICAz zugrunde, kann man den Energiebedarf zu $P = 48\,\text{mA} + 18,8\,\text{mA} = 66,8\,\text{mA}$ ermitteln. Würde der Knoten auch mit 2 Batterien zu je 2000 mAh versorgt, wären diese bereits nach $\frac{2 \cdot 2000\,\text{mAh}}{66.8\,\text{mA}} = 59.9\,\text{h}$ – also ca. 2,5 Tagen leer.

Sowohl beim MICAz, also auch beim Imote2, ist die Laufzeit von 2,5 bis 6,2 Tagen sehr gering, sodaß ohne explizite Energieeinsparung durch Software bereits nach kurzer Zeit die Batterien gewechselt werden müssen. Absolut gesehen hält die Batterie des MICAz 2,5 mal so lange wie die des MICAz, der Imote2 wird in diesem Vergleich mit 104 MHz und 32 Bit Registerbreite zu 8 MHz und 8 Bit Registerbreite des MICAz betrieben – also $\geq$ 13fachen Leistung.[14] Da beide Knoten dazu gedacht sind, lange Zeit Sensordaten zu sammeln und diese über Funk zu versenden, wird klar, daß dies nur unter der Verwendung der Energiesparmodi und spezialisierter Kommunikationsprotokolle funktionieren kann, um neben dem Prozessor auch den Transceiver möglichst oft in einen Energiesparzustand versetzen zu können.

---

[14] Hierbei handelt es sich um einen Vergleich der bei Hardwareherstellern gerne durchgeführt wird. Im Sensorknotenumfeld kommt es häufiger auf den absoluten Energiebedarf an, bei dem die Rechenleistung keine große Rolle spielt.

3.2. Messungen an Lithium Polymer-Akkus

Sowohl der MICAz als auch der Imote2 werden mit einem normalen Batterie-Adapter ausgeliefert, in den zwei AA bzw. drei AAA-Zellen einzusetzen sind. Zwar sind diese Zellen einfach und kostengünstig zu beschaffen, passen jedoch nicht in das Gesamtbild eines kleinen und leichten Sensorknotens. Aus diesem Grund wurden sowohl der MICAz als auch der Imote2 mit passenden Lithium Polymer-Akkus (LiPo) ausgestattet, die im Vergleich zu handelsüblichen Nickel-Metallhydrid-Akkus (NiMH) eine höhere Energiedichte besitzen und damit pro Leistung leichter sind. Gerade auf dem Imote2 bietet sich dies an, da sowohl die Versorgungsspannung des Knotens sowie der integrierte Batteriecontroller für Lithium-Ionen- und Lithium-Polymer-Akkus ausgelegt sind.

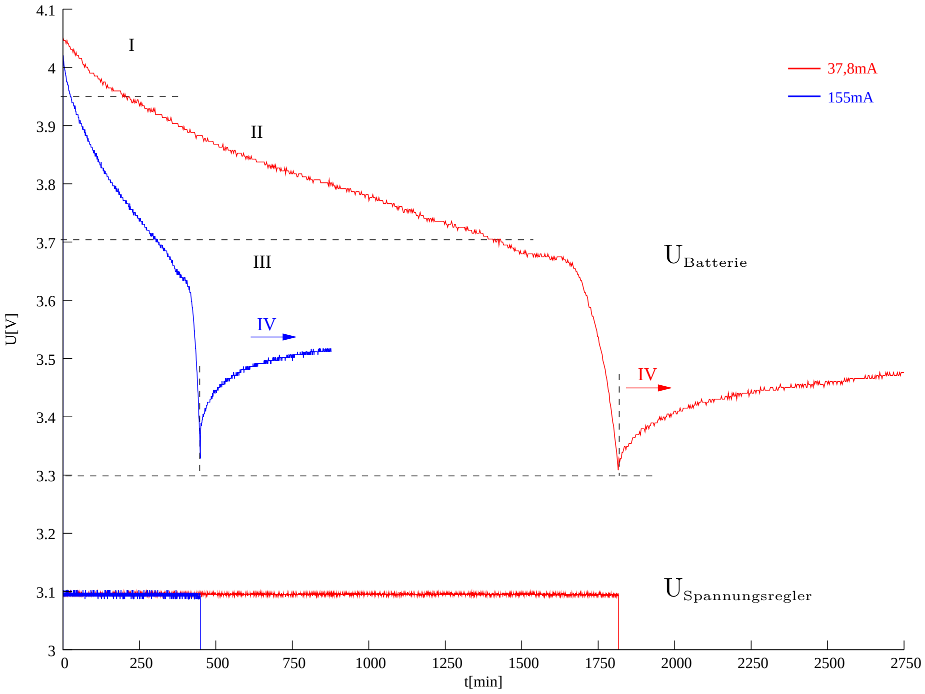


Abbildung 3.3.: Akkuentladung $Q_{\text{Nenn}} = 1500\,\text{mAh}$ bei $T = 20\,°\text{C}$ mit $I_{\text{dis}} = 37{,}8\,\text{mA}$ und $I_{\text{dis}} = 155\,\text{mA}$

Um diese Akku-Technik besser verstehen zu können, wurden einige Meßreihen [65] durchgeführt. Hierbei wurde jeweils ein vollgeladener LiPo-Akku mit einer Nennkapazität von $Q_{\text{Nenn}} = 1500\,\text{mAh}$ an die Batteriemanagement-Platine des MICAz (siehe Kapitel 2.2.1) angeschlossen. Die Platine enthält einen Spannungswandler, der eine Ausgangsspannung von $U_{\text{out}} = 3{,}1\,\text{V}$ erzeugt. Zusätzlich schützt ein Tiefentladeschutz den LiPo-Akku vor Zerstörung, indem der Verbraucher bei einer Spannung von $U_{\text{cancel}} = 3{,}3\,\text{V}$ abgeschaltet wird.[15] Ein an den Spannungsregler angeschlossener Lastwiderstand simulierte den konstanten Stromverbrauch eines Sensorknotens. In Abbildung 3.3 dargestellt sind die Messungen einer Konstantstromentladung mit $I_{\text{dis}} = 37{,}8\,\text{mA}$ ($R = 82\,\Omega$) und $I_{\text{dis}} = 155\,\text{mA}$ ($R = 20\,\Omega$) bei Raumtemperatur ($T = 20\,^{\circ}\text{C}$).[16] Beide Messungen dienten dazu, einen kleinen und größeren Verbraucher zu simulieren und das Verhalten des Akkus dabei zu untersuchen.[17] Dabei wurde sowohl die Spannung des Akkus (obere Kurve), als auch die Spannung am Spannungswandler (untere Kurve) aufgetragen.

Die Messung ergab, daß bei einem Entladestrom von $I_{\text{dis}} = 37{,}8\,\text{mA}$ der Akku nach 1815 Minuten soweit entladen war, daß die Schutzschaltung abgeschaltet hat. Dies entspricht nun rechnerisch einer nutzbaren Kapazität von

$$Q_{\text{Nutz}} = \frac{1815\,\text{min}}{60} \cdot 37{,}8\,\text{mA} = 1143\,\text{mAh}.$$

Das gleiche Experiment wurde auch bei einer höheren Belastung des Akkus mit $I_{\text{dis}} = 155\,\text{mA}$ Stromverbrauch durchgeführt – hier löste die Schutzschaltung bereits nach $t = 449{,}5$ Minuten aus, was einer nutzbaren Kapazität von

$$Q_{\text{Nutz}} = \frac{449{,}5\,\text{min}}{60} \cdot 155\,\text{mA} = 1161\,\text{mAh}$$

entspricht. Die Messung zeigt somit eine Diskrepanz zwischen der vom Hersteller angebenen Nenn-Kapazität von $Q_{\text{Nenn}} = 1500\,\text{mAh}$ und der ermittelten Kapazität in Höhe von ca. $Q_{\Delta} = 300\,\text{mAh}$.

In der Arbeit von Szente-Varga [109] wurden einige dieser Messungen an NiMH-Akkus automatisiert durchgeführt. Sowohl bei NiMH- als auch bei Lithium Polymer-Akkus läßt sich der Spannungsverlauf während der Entladung in 4 Bereiche unterteilen. Am Anfang der Entladung sinkt die Batteriespannung schnell bis auf 3,95 V (I), danach folgt ein linearer Abschnitt bis 3,7 V (II), danach sinkt die Spannung exponentiell bis zur Abschaltung bei 3,3 V (III) ab. Anschließend setzt nach Abschaltung der Entladung (IV) eine Erholung des Akkus (*self-recharge*-Effekt [29, 109]) ein und die Spannung steigt wieder. Bei bekannter Nennkapazität und Stromaufnahme des

---

[15] Die Spannung einer LiPo-Zelle sollte nie unter die minimale Zellspannung von 3,2 V fallen.

[16] Alle Werte sind durch Messungen mit dem gleichen Meßgerät bestimmt; Abweichungen vom Nennwert ergeben sich aus Fertigungstoleranzen.

[17] 37,8 mA $\cong$ Imote2 (104 MHz, idle-Modus), 155 mA $\cong$ Imote2 (416 MHz, active-Modus)

Verbrauchers kann somit durch eine Spannungsmessung die Restkapazität des Akkus ermittelt werden.[18]

3.3. Energie-Planung mit SDL

Aus den vorherigen Abschnitten ist bereits deutlich geworden, daß Energiesparen nötig und wichtig für eine lange Laufzeit von Geräten ist. Häufig wird bei der Entwicklung keine Zeit für diesen Aspekt eingeplant, sondern das Produkt erstellt und erst in späten Entwicklungsphasen die Laufzeit wieder verlängert. Die Energiekontrolle kann dabei nicht ausschließlich nur auf Ebene des Betriebssystems erfolgen, sondern muß sowohl in der Applikation einer Middleware und ggf. auch einer MAC-Schicht berücksichtigt werden. Viele Informationen, die zur Energiekontrolle benötigt werden, werden durch die Abstraktion nicht an das Betriebssystem weitergeleitet. Deshalb ist es für eine optimale Laufzeit nötig, die Kontrolle bereits schon in der Spezifikationssprache SDL (siehe Kapitel 2.1) zu ermöglichen. Im Folgenden wird gezeigt, wie dies bereits in der Spezifikation berücksichtigt werden kann, aber auch wie dieser Aspekt im laufenden Betrieb angewendet werden kann. Alle Ergebnisse wurden auf der internationalen Konferenz „SDL Forum 2009“ vorgestellt, die Ausarbeitung dazu ist als Publikation in den LNCS-Proceedings [47] enthalten.

Zur Kontrolle von Energie lassen sich zwei komplementäre, aber ergänzende Ansätze finden: explizite und implizite Energiekontrolle.

Explizite Energiekontrolle erfolgt als *Energiemodussignalisierung* direkt im SDL-Modell. Hierbei nutzt der SDL-Entwickler bei der Spezifikation spezielle Signale, um den Energiezustand zu wechseln. Die Übergänge innerhalb des Energiemodells aus Kapitel 3.1 werden hierbei explizit durchgeführt. In größeren Systemen sollte die Energiemodussignalisierung pro Hardwarekomponente zentral in einer eigenen Energiekomponente gekapselt werden, damit es nicht zu gegenläufigen Entscheidungen innerhalb eines Systems kommt.

Implizite Energiekontrolle wird direkt innerhalb der *SDL Virtual Machine (SVM)* [43] gesteuert und als *Energieplanung* (*Energy Scheduling*) bezeichnet. Hierbei werden Eigenschaften von SDL, wie Signalwarteschlangen und Timer benutzt. Aufbauend darauf und mit Hilfe der Hardwaremodelle kann der Energieverbrauch gesteuert

---

[18] Bei Akkus spielt auch das Alter eine Rolle, diese Größe sollte bei einer Berechnung mitberücksichtigt werden.

werden. Bei dieser Steuerung ist es nicht nötig, Veränderungen an der Spezifikation vorzunehmen, deshalb bezeichnen wir dies als implizite Energiekontrolle.

Mittels Energiemodussignalisierung kann die CPU zwar in einen Schlafzustand versetzt werden, jedoch hat der steuernde Prozess immer nur eine eingeschränkte Sicht und kann keine globalen Systemzustände, wie Anzahl der feuerbereiten Transitionen, aktive Signale bzw. Timer im System prüfen. Genau diese Informationen kann die SVM intern aber prüfen und aufgrund dessen einen geeigneten Schlafzustand auswählen, in dem sie die Zeit bis zum nächsten Auslösen eines Timers ermittelt. Auf der anderen Seite kann ein Transceiver nur über explizite Energiekontrolle sinnvoll gesteuert werden, da der aktuelle Systemzustand der SVM keine Rückschlüsse auf den Benutzungszyklus (duty cycling) [23] zuläßt. Eine Kontrolle solcher Hardwarekomponenten muß immer explizit aufgrund von dynamischen oder statischen Plänen erfolgen. Es zeigt sich, daß sowohl implizite als auch die explizite Energiekontrolle Einschränkungen haben, weshalb nicht nur eine, sondern immer beide Techniken zur optimalen Kontrolle eingesetzt werden sollten.

3.3.1. Energiekontrolle der CPU

Am Beispiel des Imote2 wird nun gezeigt, wie Energie für die CPU sowohl implizit als auch explizit kontrolliert werden kann. Aus den Energiemodellen des Imote2 (vergleiche Kapitel 3.1.2) geht hervor, daß sowohl die CPU-Frequenz als auch der verwendete Zustand der CPU Einfluß auf den aktuellen Verbrauch haben. Die CPU-Frequenz steuert hierbei die Verarbeitungsgeschwindigkeit, schränkt aber die Funktionsweise der CPU nicht ein. Befindet sich die CPU hingegen im *idle* oder *sleep* Modus, ist die Abarbeitung von Befehlen abgeschaltet.

CPU-Frequenzsignalisierung In speziellen Situationen kann es sinnvoll sein, die Leistungsfähigkeit eines Systems, also die CPU-Frequenz, zu beeinflussen. Zur Steuerung der Frequenz aus SDL heraus wurde ein spezielles Signal `CPU_FREQ_MODE` mit der gewünschten Frequenz als Parameter (siehe Listing 3.1, Zeile 5) eingeführt. Wird das Signal gesendet (Zeile 7), wird es von der Umgebung (SEnF – siehe Kapitel 2.1 und Kapitel 6.1.3) verarbeitet und auf die Hardware oder das darunterliegende Betriebssystem abgebildet. Auf der Imote2-Plattform kann man zwischen den Frequenzen 13 MHz, 104 MHz, 208 MHz und 416 MHz wählen.

CPU-Zustandssignalisierung Die Kontrolle des CPU-Zustandes läßt sich über das spezielle Signal `CPU_OP_MODE` (Listing 3.2, Zeile 5) erreichen. Als Parameter wird

```
syntype CPU_Frequency = Integer
  constants 13, 104, 208, 416
endsyntype

signal CPU_FREQ_MODE (CPU_Frequency);

output CPU_FREQ_MODE (104);
```

Listing 3.1: Auschnitt: CPU-Frequenzsignalisierung in SDL

zum einen der gewünschte CPU-Modus als auch die Zeit, zu der die CPU wieder in den ursprünglichen Modus zurückkehren soll, angegeben. Befindet sich die CPU in einem Schlafzustand, wird die Ausführung komplett angehalten; damit ist eine Kontrolle der CPU aus SDL heraus nicht mehr möglich. Der zusätzliche Parameter gibt den Zeitpunkt an, zu dem das System wieder im vorherigen Betriebsmodus sein soll. Die Anwendung des Signals ist in Zeile 7 gezeigt, welches wie auch die Frequenzsignalisierung von SEnF, verarbeitet wird.

```
1 newtype CPU_Operation
2   literals active, idle, deepIdle, standby, sleep, deepSleep;
3 endnewtype
4
5 signal CPU_OP_MODE (CPU_Operation, Time);
6
7 output CPU_OP_MODE (idle, now+0.01);
```

Listing 3.2: Definition und Anwendung des `CPU_OP_MODE` Signals

Energieplanung CPU-Mode Die Steuerung der CPU-Modi mittels SDL-Signalen ist in kleinen Systemen einfach zu realisieren. Mit wachsender Größe der Systeme ist eine feingranulare Steuerung so kaum mehr möglich. Eine Signalisierung kann jedoch dazu eingesetzt werden, ein System gezielt für eine bestimmte Zeit abzuschalten. Im Normalfall ist es nicht möglich oder einfach zu aufwändig, ein System soweit zu analysieren, daß ein Ablaufplan erstellt werden kann, der bei jeder Änderung angepaßt werden müßte. Andererseits sind viele Informationen, die zur Energieplanung benötigt werden, bereits in der SDL Virtual Machine (SVM) vorhanden. Unter der Annahme, daß die SVM Zugriff auf den Ablaufplan und die aktiven Signale und Timer innerhalb der Prozesse hat, kann die globale Entscheidung über den CPU-Modus zentral innerhalb der SVM gesteuert werden.

Listing 3.3 zeigt die Energieplanung der CPU unter Verwendung der SDL-Transitionsauswahl (`selectTransition`) und der SDL-Timerauswahl (`selectNextExpi_` `ringTimer`) in Pseudo-Kode. Wird keine feuerbereite Transition gefunden (Zeile 3), prüft der Scheduler, ob sich im System aktive Timer befinden. Wurde ein Timer gefunden, erfolgt die Auswahl des CPU-Modus innerhalb des SEnF (siehe auch Kapitel 3.3.3) aufgrund der aktivierten Hardwarekomponenten und der Zeit, zu dem der Timer auslösen soll. Das System wird dann für diese Zeitspanne in diesen Modus versetzt (Zeile 8), und kann entweder durch das Ablaufen der Zeit oder durch ein externes Ereignis (Interrupt) geweckt werden. Ist kein Timer im System aktiv, wird das System vollständig reaktiv (Zeile 10), d.h. das System kann nur noch durch einen externen Interrupt geweckt werden. In diesem Fall wird der sparsamste CPU-Modus gewählt, der einen Betrieb der verwendeten Hardwarekomponenten in ihrem aktuellen Betriebsmodus noch zuläßt.

```
 1 while (running) do
 2   selectTransition
 3   if (transitionFound) then
 4     fireTransition
 5   else
 6     selectNextExpiringTimer in timerQueue
 7     if (timerFound) then
 8       selectSleepMode until (timer expires or interrupt occurs)
 9     else
10       selectSleepMode until (interrupt occurs)
11     fi
12   fi
13 od
```

Listing 3.3: CPU Energie-Scheduler

Damit der Scheduler korrekt arbeitet, müssen bei der Spezifikation zwei Dinge beachtet werden:

1. In *Continuous*-Signalen dürfen keine Abhängigkeiten zur aktuellen Systemzeit bestehen. Dies stellt keine nennenswerte Einschränkung dar, da dies auch leicht über Timer abgebildet werden kann.

2. Indeterminismus, wie es mit Hilfe von *Spontaneous Transitions* ausgedrückt wird, sollte nicht verwendet werden, da dies bei der Wahl des Schlafzustands nicht ohne weiteres berücksichtigt werden kann.

Energieplanung CPU-Frequenz Auch die richtige Wahl der CPU-Frequenz ist ohne exaktes Wissen über alle beteiligten Komponenten schwierig. Wird die Frequenz zu gering gewählt, entsteht im System eine Überlast, die das Verhalten negativ beeinflußt. Die Latenzzeiten steigen an, Timerereignisse lösen zu spät aus, wodurch der Jitter der Prozesse steigt (siehe auch Kapitel 3.4). Ist die Ressourcenanforderung über längere Zeit zu groß, kann das gesamte System zum Erliegen kommen (meist durch höhere Speicheranforderungen als dem Gesamtsystem zur Verfügung stehen).

Wird die Frequenz zu hoch gewählt, hat dies keine negativen Auswirkungen auf die Ausführung, jedoch ist der Energieverbrauch höher als nötig.

Auch hier kann die SVM bei der Wahl der richtigen Frequenz helfen, indem sie die durchschnittliche Wartezeit innerhalb der Signalqueues mißt und auswertet. Auch die Anzahl der im System aktiven Prozesse kann mitberücksichtigt werden. Analysen wie von D. Schmitt [96] zeigen, daß es energetisch sinnvoller ist, mit maximaler Geschwindigkeit die Berechnung auszuführen und danach eine längere Schlafphase einzunehmen. Die ermittelten Daten decken sich auch mit dem Energiemodell, daß die CPU bei 416 MHz um den Faktor 2,75-fach mehr Energie bei 4-facher Rechenleistung gegenüber 104 MHz benötigt!

3.3.2. Energiekontrolle des Transceivers

Die Verwaltung der Energiemodi des Transceivers ist im Gegensatz zur CPU recht einfach. Zum einen wird der Transceiver nicht gleichermaßen von jedem Befehl benötigt, zum anderen ist die Anzahl an möglichen Zuständen vergleichsweise klein. Aus dem Energiemodell des Transceivers (siehe Kapitel 3.1.1) geht hervor, daß es neben den zwei aktiven Modi *send* und *receive* nur noch drei Schlafzustände gibt. Sofern nicht längere Ruhezeiten geplant sind, schließen sich die *Power Down* Modi wegen der langen Wiederanlaufzeiten (Einstellungen des Transceivers müssen wiederhergestellt werden) aus und es bleibt nur der *idle* Modus übrig. Aus der SVM selbst läßt sich die Verwendung des Transceivers nicht bestimmen, weshalb hier nur eine explizite Kontrolle des Zustands verwendet werden kann. Mittels Strategien zur Optimierung des Benutzungszyklus kann der Transceiver über ein spezielles Signal `CC2420_OP_MODE` (siehe Listing 3.4) explizit gesteuert werden. Da der Medienzugriff (MAC) meist in einer gekapselten Komponente realisiert wird, kann die Steuerung des Energiezustandes dort zentral erfolgen, sofern ein Duty-Cycling-Plan vorhanden ist.

In Zeile 2 sind lediglich die Modi `powerDown`, `idle` und `receive` genannt. Der Betriebsmodus *send* wird bereits explizit über ein SDL-Signal, das die zu versendenden Daten enthält, realisiert, sodaß hier keine explizite Betriebsmodus-Umschaltung nötig ist.

```
newtype CC2420_Operation
  literals powerDown, idle, receive;
endnewtype

signal CC2420_OP_MODE (CC2420_Operation);

output CC2420_OP_MODE (powerDown);
```

Listing 3.4: Definition und Anwendung des `CC2420_OP_MODE` Signals

3.3.3. Implementierung

Die Implementierung der verschiedenen Energiemodi ist von der verwendeten Hardware abhängig. Alle hardwarespezifischen Implementierungen wurden deshalb in das von der AG Vernetzte Systeme entwickelte SDL Environment Framework (SEnF) integriert.

Energiemodussignalisierung Bei der Signalisierung eines neuen Energiemodus werden Signale in SDL erzeugt und diese an das SEnF weitergeleitet. Die Umsetzung der Signale wird dann im dafür vorgesehenen Hardwaretreiber durchgeführt. Bevor jedoch eine Umschaltung des jeweiligen Modus durchgeführt werden kann, müssen Vorbedingungen erfüllt sein: Beispielsweise sollte der Transceiver nicht in einen Energiesparmodus versetzt werden, wenn noch ein Sendevorgang läuft. Für den Wechsel des CPU-Zustandes müssen ggf. zunächst Puffer geleert werden. Danach kann erst der Timer gesetzt werden, damit die CPU aus dem ausgewählten Modus wieder aufwacht.

Sind alle Vorbedingungen erfüllt, kann der Treiber die Umschaltung durchführen. Handelt es sich um einen Energiesparmodus der CPU, kann das Aufwachen zum einen aus einem abgelaufenem Timer, aber auch aus einem externen Ereignis (Interrupt) hervorgerufen worden sein. War der Grund des Aufwachens ein Interrupt, muß dieser zunächst verarbeitet werden. Da durch das Aufwachen aus dem Schlafzustand bereits mehr Zeit bis zur Verarbeitung des Interrupts vergangen ist, sollte diese Abarbeitung zeitnah erfolgen! SDL-Signale, die aufgrund des aufgetretenen Interrupts generiert wurden, werden in das SDL-System eingefügt und das SDL-System erneut ausgeführt. Wurden keine Signale generiert oder die generierten Signale vom SDL-System verarbeitet, wird von der Laufzeitumgebung erneut das SEnF ausgeführt und dort ggf. erneut in den Schlafzustand gewechselt.

Kapitel 4.2 und Kapitel 5 zeigen eine praktische Anwendung der Steuerung, die durch *Echtzeitsignalisierung* sehr präzise arbeitet. Die dort vorgestellten Ergänzungen stellen sicher, daß ein Schlafzustand weder zu früh, noch zu spät betreten oder verlassen wird.

Energieplanung Wie bereits erwähnt, erfolgt die Energieplanung in Zusammenarbeit mit der SVM. Innerhalb der SVM sind alle Agenten gleichberechtigt, werden also nach dem FIFO-Prinzip aktiviert. Die Umgebung (Environment), die durch das SEnF repräsentiert ist, wird ebenso wie jeder andere Agent in diese Warteschlange aufgenommen. Wird der Agent für das SEnF aktiviert, um Signale an die Hardware zu senden oder bereits empfangene Signale in das SDL-System zu versenden, wird von der SVM zusätzlich noch der Zeitpunkt der nächsten Systemaktivität mitgeteilt:

```
void xInEnv (SDL_Time next_timer_expiration);
```

Listing 3.5: Signatur für das Polling des SEnF

Listing 3.5 zeigt die Signatur für das Polling der Eingangswarteschlange des Environment. Der Parameter `next_timer_expiration` gibt hierbei die Zeit an, bis der nächste SDL Timer auslöst. Sind noch feuerbereite Transitionen im System vorhanden, wird dieser Parameter mit dem Zeitpunkt „now“ übergeben. Aus

$$d := (\texttt{next\_timer\_expiration} - \texttt{now})$$

kann die maximale Schlafdauer bis zur nächstens Systemaktivität berechnet werden. Ist $d > 0$, besteht die Möglichkeit zum Schlaf; um negative Einflüsse durch das Schlafen zu minimieren, muß jedoch die Zeit für das Wiederaufwachen einkalkuliert werden. Das Umschalten in den *idle* Modus erfolgt deshalb erst, wenn $d > 60\,\mu\mathrm{s}$ ist. Falls $d > 2\,\mathrm{s}$, lohnt sich auch die Umschaltung in den energieärmeren *deep idle* Modus. Der Hardwaretimer für das Wiederaufwachen wird dabei auf

$$t := \texttt{now} + (d - \texttt{estimated\_wakeup\_time})$$

eingestellt.

3.3.4. Evaluation

Eine allgemeine Evaluation der Energieplanung ist nicht möglich und sehr stark vom Anwendungsgebiet abhängig. Ich möchte jedoch kurz am Beispiel einer Middleware für das inverse Pendel, wie sie in Kapitel 5.2.2 vorgestellt wird, zeigen daß mittels Energieplanung bereits Einsparungen möglich sind. Bei dem gewählten Szenario senden drei Imote2-Sensorknoten über den Transceiver mit einer Periode von 60 ms Daten an einen Regler-Knoten.

Ohne Energiekontrolle verbraucht die CPU des Imote2 innerhalb von 5 Minuten im Active-Modus somit

$$Q_{\text{activeSz}} = 48\,\text{mA} \cdot 300\,\text{s} = 14400\,\text{mAs} = 4\,\text{mAh}.$$

Nach 5 Minuten bei einem CPU-Takt von 104 MHz ergeben sich mit aktivierter impliziter Energiekontrolle in der NCSCom die Werte, wie sie in Tabelle 3.1 gezeigt sind.

	Kernel	Agenten	Environment	Interrupt	Sleep
Sensor	4,8 %	14,3 %	2,5 %	1,6 %	76,8 %
Regler	6,7 %	27,0 %	2,5 %	1,8 %	62,2 %
Aktuator	4,8 %	10,9 %	2,8 %	1,4 %	80,0 %

Tabelle 3.1.: CPU-Anteil nach 5 Minuten Ausführung der NCSCom

Am Beispiel des Regler-Knotens zeigt sich eine Auslastung von ∼37,8%. Es ergibt sich so ein Energieverbrauch von

$$Q_{\text{save}} = 37{,}8\,\% \cdot 48\,\text{mA} \cdot 300\,\text{s} + 62{,}2 \cdot 39\,\text{mA} \cdot 300\,\text{s} = 12720{,}6\,\text{mAs} \approx 3{,}53\,\text{mAh}.$$

Das Aktivieren der impliziten Energiekontrolle für die CPU kann somit beim Pendel-Szenario auf dem Imote2 $Q_{\text{save}}/Q_{\text{activeSz}} \approx -12\,\%$ Energie einsparen.

3.4. Jitter

Der Begriff *Jitter* wird meist im Zusammenhang mit analogen oder digitalen Signalen, aber auch in der Netzwerktechnik bei Netzwerkframes verwendet. Er bezeichnet dabei sowohl die Varianz der Zeit von periodischen Abläufen, aber auch die Abweichung des erwarteten Ausführungszeitpunktes zum tatsächlichen Auslösezeitpunkt. Im Unterschied zur normalen Verzögerung ist beim Jitter auch eine verfrühte Ausführung möglich. Bei *Echtzeitsystemen* wird dieser Begriff verwendet um die mögliche Abweichung zum definierten Zeitpunkt anzugeben. Dabei versteht man unter einem Echtzeitsystem ein System, welches das Ergebnis einer Berechnung bis zu einem vorher definierten Zeitpunkt garantieren muß [101], wobei man zwischen harten

(Verfehlung des Zeitpunktes ist kritisch) und weichen Echtzeitsystemen (Verfehlung unerwünscht, aber nicht kritisch) unterscheidet. Dabei spielt der Begriff nicht nur in der Datenübertragung, sondern auch bei der Ausführung von Prozessen, Threads und Tasks, eine wichtige Rolle.

3.4.1. Jitter bei Prozessen

Häufiges Aufwachen ist eine der Ursachen für hohen Energiebedarf; dies soll vermieden werden, indem Ereignisse zusammengelegt und somit die Schlafzeit seltener unterbrochen wird. Die serielle Ausführung (auf einem Kern) dieser gleichzeitigen Ereignisse führt in der Folge dazu, daß jeder Prozess außer dem ersten verzögert wird. Die Reihenfolge der Ereignisse und Prozesse wird hierbei vom Betriebssystem geplant und kann nicht weiter vom Prozess beeinflußt werden.

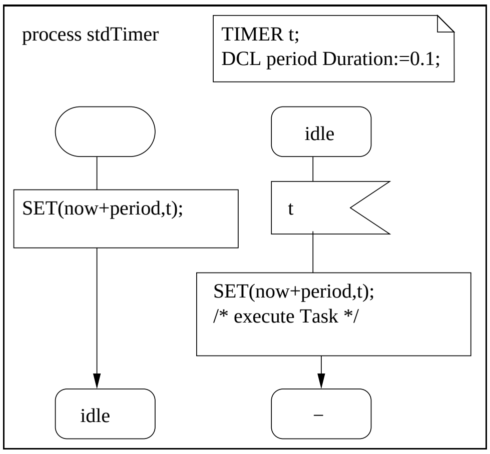


Abbildung 3.4.: Setzen eines Timers in SDL

SDL stellt mit seiner Virtuellen Maschine, der Laufzeitumgebung und den Treibern selbst ein Betriebssystem dar. Dieses Betriebssystem ist im Falle von Sensorknoten, direkt auf der Hardware lauffähig oder setzt bei normalen PCs auf einem bestehenden Betriebssystem wie Linux und Windows [83] auf. Auch in SDL existieren Prozesse, die unabhängig voneinander agieren, und mit Timern besteht die Möglichkeit, zeitgesteuert lokale Ereignisse auszulösen. Da alle Prozesse in SDL ebenfalls um die Rechen-Ressource konkurrieren, kann es hier zu Verzögerungen bei der Ausführung der Prozesse kommen.

Abbildung 3.4 zeigt zunächst exemplarisch, wie Timer in SDL verwendet werden. Beim Setzen des Timers wird die aktuelle Systemzeit mit dem Schlüsselwort `now` als Referenz verwendet. Jeder Prozess hat eine Eingangswarteschlange, in der sowohl Signale, als auch abgelaufene Timer einsortiert werden. Befinden sich in der Queue weitere Signale, die vor dem abgelaufenem Timer abgearbeitet werden müssen oder belegt ein andere Prozess noch die Rechenressource, verzögert sich dadurch die Ausführung der Transition, die durch den Timer ausgelöst werden soll. Der Zeitpunkt `now` entspricht somit nicht mehr der erwarteten Zeit für die Timerauslösung und alle weiteren Timer verschieben sich. In der Praxis sinkt somit die Häufigkeit der Timerauslösungen pro Zeitintervall gegenüber dem aus der Spezifikation erwarteten Wert.

3.4.2. Jitter im Scheduler

Bevor nun die Frage nach der exakten Timerauslösung geklärt wird (siehe Kapitel 3.4.3), soll zunächst gezeigt werden, wie sich Jitter auch positiv nutzen läßt. In Abbildung 3.5 sind dazu drei Prozesse mit periodischer Ausführung gezeigt, die zum Zeitpunkt 0 starten. Jeder der Prozesse benötigt 20 Einheiten Rechenzeit, wobei die Periode von Prozess `P1` 180 Einheiten, von Prozess `P2` 200 Einheiten und von Prozess `P3` 220 Einheiten beträgt. Ein resultierender Ausführungsplan ohne Beachtung von Jitter ist in der Zeile „normal“ dargestellt. Beim Start des Systems konkurrieren alle drei Prozesse um die Ressource und werden in eine Reihenfolge gebracht, sodaß die erste Ausführung von `P1` direkt zum Startzeitpunkt (0), `P2` zum Zeitpunkt 20 und von `P3` zum Zeitpunkt 40 stattfinden kann. Wie zu erwarten, verteilen sich danach die Ausführungszeiten und es kommt innerhalb des dargestellten Zeitraums zu keiner weiteren Konkurrenz. Die weißen Bereiche zwischen der Ausführung markieren dabei die Zeitspannen, in denen der Prozessor nicht benötigt wird und Energie sparen kann. Wie leicht zu erkennen ist, muß der Prozessor häufig wieder aufwachen (dargestellt sind 17 Aufwachzeitpunkte), wodurch die kontinuierliche Schlafzeit klein ausfällt.

Wird bereits zum Zeitpunkt der Spezifikation explizit ein zulässiger Jitter angegeben, erlaubt der Entwickler Abweichungen vom spezifizierten Beginn der Ausführung und gibt dem Scheduler dadurch die Möglichkeit, einen besseren und energieeffizienteren Ausführungsplan zu erstellen. Für die Prozesse `P1`, `P2` und `P3` wurde ein Jitter von jeweils 40 Einheiten um den Startzeitpunkt angegeben; dies ist in Abbildung 3.5 als farbiger Rahmen zu erkennen. Für Prozess `P1` ergeben sich somit die Ausführungsintervalle [0,40], [140,220], [320,400] usw. Für die Prozesse `P2` und `P3` ist dies ebenfalls, wie in der Abbildung gezeigt, definiert. Da kein Prozess vor dem Startzeitpunkt „0“

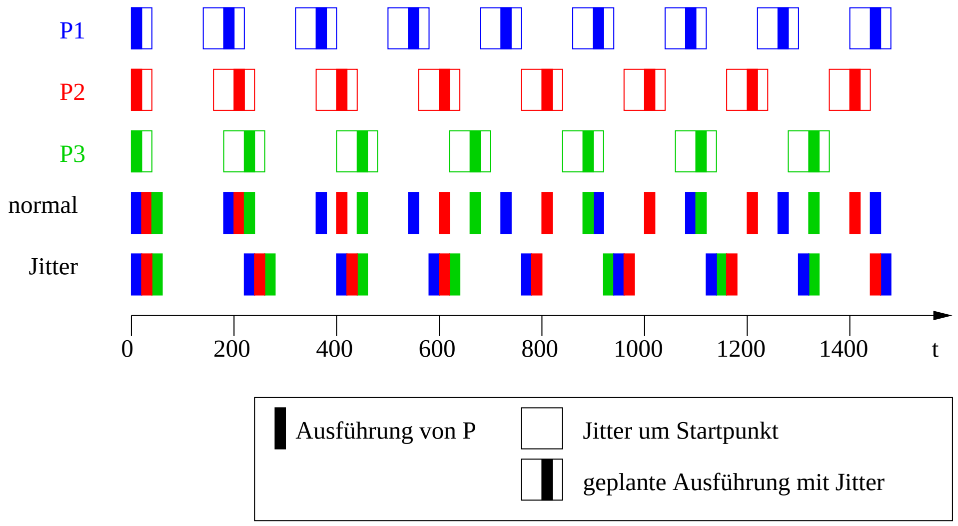


Abbildung 3.5.: Ausführung von Prozessen mit/ohne Jitter mit exakten Timern

gestartet werden kann, unterscheidet sich die Darstellung dieses Zeitpunktes in der Abbildung.

Unter Berücksichtigung des spezifizierten Jitters ergibt sich mit einem angepaßten Scheduling-Algorithmus der Verlauf wie in Abbildung 3.5 in der Zeile „Jitter“ dargestellt. Das Verfahren versucht, die Anzahl der Aufwachvorgänge klein zu halten und möglichst viele Tasks hintereinander abzuarbeiten, sofern der Jitter dies erlaubt. Im Vergleich zu dem Verfahren ohne Jitter zeigen sich weniger Aufwachvorgänge (nur 8 statt 17) sowie eine bessere Gruppierung der Tasks, wodurch die zusammenhängenden Schlafzeiten deutlich länger ausfallen.

Viele kurze Schlafvorgänge sind im Vergleich zu einer durchgehend langen Schlafphase als schlechter zu bewerten, da zum einen der Wechsel zum Schlafzustand selbst Zeit und Energie benötigt. Zum anderen entsteht immer ein Verschnitt, da der Prozessor rechtzeitig aufwachen muß, um den Prozess möglichst zum geplanten Zeitpunkt auszuführen. Lange Phasen der Inaktivität erlauben die Nutzung eines Schlafzustandes, der längere Aufwachzeiten benötigt, aber einen geringeren Energieverbrauch besitzt.

In SDL wird die Angabe des Jitters über qualifizierte Kommentare realisiert. Listing 3.6 zeigt die Definition zweier Timer `t1` und `t2`, die jeweils nach 200 ms auslösen und über einen Jitter von ±40 ms variieren, also bereits in 160 ms, späte-

```
TIMER t1;
SET(now + 0.2, t1);/*#Jitter:40ms*/

TIMER t2:=0.2;/*#Jitter:40ms*/
SET(t2);
```

Listing 3.6: Timer in SDL mit Jitter

stens aber nach 240 ms ausgelöst haben sollen. Timer `t2` ist äquivalent zu `t1`, es handelt sich lediglich um eine Kurzschreibweise. Der in der AG Vernetzte Systeme entwickelte Transpiler ConTraST (siehe Kapitel 6.1) setzt die SDL-Spezifikation in C++-Kode um, interpretiert diese Kommentare und ordnet sie dem jeweiligen Timer (`si.jitter`) zu. Die Ausführung der SDL-Spezifikation erfolgt dabei unter Zuhilfenahme des SDL-Laufzeitmodells (SdlRE), das anhand der formalen Semantik [57] entwickelt wurde. SdlRE stellt dabei alle Strukturen, Warteschlangen und einen Scheduler zur Transitionsauswahl bereit. Zusätzlich zu der normalen Eingangswarteschlange, die jeder Prozess besitzt, verwaltet SdlRE eine weitere Warteschlange, in der aktive Timer gelistet sind (`timerQueue`). Listing 3.7 zeigt die Anpassungen in Pseudo-Kode die am SDL-Scheduler in SdlRE umgesetzt wurden um damit das Scheduling wie in Abbildung 3.5 zu erreichen.

```
1  while (running) do
2    selectTransition
3    if (transitionFound) then
4      fireTransition
5    else
6      selectNextExpiringTimer in timerQueue with (si.arrival − si. jitter )<=now
7      if (timerFound) then
8        fire transition
9      else
10       selectNextExpiringTimer in timerQueue with min(si.arrival + si.jitter)
11       if (timerFound) then
12         sleep until (si . arrival + si. jitter or interrupt occurs)
13       else
14         sleep until (interrupt occurs)
15       fi
16     fi
17   fi
18 od
```

Listing 3.7: Scheduling-Algorithmus für Timer in Pseudo-Kode in Anlehnung an die SDL-Semantik

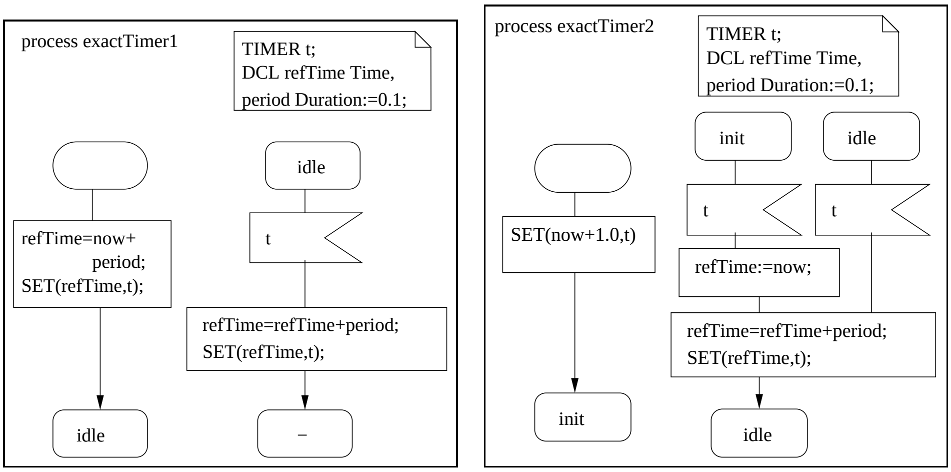
Der Scheduler führt zunächst, genau wie der Original-Scheduler, alle Transitionen aus, die bereits feuerbereit sind (siehe Zeile 2 – Zeile 4). Nachdem diese Liste leer ist, wird im nächsten Schritt die Liste der Prozesse durchsucht, die einen Timer gesetzt haben. Hier werden alle Tasks gesucht, die Timer-Ereignisse in der Zukunft haben, die aber nach Abzug des eingestellten Jitters feuerbereit wären (Zeile 6). Alle so gefundenen Transitionen werden behandelt, als wären sie bereits feuerbereit, wobei die Auswahl in der Reihenfolge des Ablauf-Zeitpunktes erfolgt. Ist sowohl diese Liste leer als auch die Liste der feuerbereiten Transitionen, beginnt die Suche nach der Transition, der nach Ausnutzung des maximalen Jitters am frühesten auslöst (Zeile 10). Ist ein solcher Timer vorhanden, erfolgt die Auswahl des zugehörigen Schlafmodus für die ermittelte Zeit abzüglich der Aufwachzeit in einer plattformspezifischen Schlafroutine. Sind keine Timer mehr im System aktiv, wird das System reaktiv und wartet auf einen auftretenden externen Interrupt.

Der hier vorgestellte Algorithmus versucht die Dauer der durchgehenden Schlafphasen zu maximieren. Hierbei bleibt unberücksichtigt, daß bei vielen parallel ausführbaren Prozessen deren maximaler Jitter überschritten werden kann. In Zeile 10 müßte also nicht nur der späteste Zeitpunkt ermittelt, sondern alle in diesem Zeitfenster auslösenden Tasks gefunden werden und diese unter Verwendung eines angepaßten *Earliest Deadline First*-Algorithmus [101] in eine Reihenfolge gebracht werden – bis zu dem so ermittelten Anfangszeitpunkt kann dann geschlafen werden.

Die Höhe der Einsparung hängt im Detail von dem spezifizierten System und der verwendeten Hardware ab. Generell eigenen sich längere Schlafphasen besonders für sehr sparsame Energiemodi, da hier die Aufwachzeiten länger sind und damit mehr Energie eingespart werden kann. Häufige Wechsel der Energiezustände (Einschlafen bzw. Aufwachen) kostet zusätzlich Energie, da in der Zeit, die für den Wechsel benötigt wird, weder gerechnet noch Energie eingespart wird.

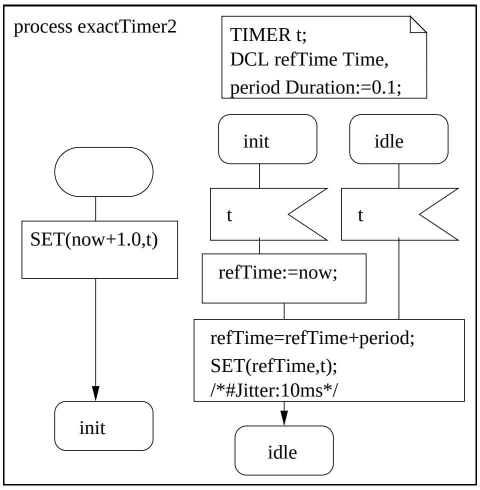
3.4.3. Jitter und exakte SDL-Timer

In SDL werden Timer üblicherweise, wie in Abbildung 3.4 gezeigt, gesetzt [31]. Dabei wird für das Setzen des nächsten Auslösezeitpunktes die aktuelle Systemzeit (`now`) benutzt und die Dauer bis zur nächsten Ausführung addiert. Der SDL-Standard [57] gibt für Timer nur an, daß diese nicht vor ihrem gesetzten Zeitpunkt auslösen sollen, aber nicht bis zu welchem Zeitpunkt dies geschehen sein muß. Aus der Spezifikation gemäß Abbildung 3.4 würde man eine Periodizität von 100 ms erwarten, jedoch besagt die Spezifikation nur, daß die Periode nicht kleiner als 100 ms ist. In der Implementierung der SDL Virtual Machine (beispielsweise SdlRE [36] oder CAdvanced [52]) erfolgt die Abarbeitung der Timer zeitnah, aber nie exakt, d.h. die aktuelle Systemzeit liegt nach der spezifizierten Auslösezeit des Timers, wodurch sich beim erneuten Setzen des Timers der nächste Auslösezeitpunkt verschiebt.



(a) Realisierung im idealisierten Fall

(b) praktische Anwendung **ohne** Jitter



(c) praktische Anwendung **mit** Jitter

Abbildung 3.6.: exakte Timer

Eine Lösung für die beschriebene Problematik ist in Abbildung 3.6(a) dargestellt. Da es sich hierbei um eine Lösung in SDL handelt, erfolgt auch hier die Ausführung des Timers nur zeitnah und nicht „exakt“. Der Unterschied besteht jedoch darin, daß zur Berechnung des nächsten Auslösezeitpunktes nicht die aktuelle Systemzeit (`now`), sondern der berechnete Zeitpunkt, zu dem der Timer hätte auslösen sollen, verwendet wird. Um dieses Verhalten zu erreichen, wird bei der Initialisierung des Prozesses (initiale Transition) der aktuelle Zeitpunkt in einer Variablen `refTime` gespeichert. Löst nun der Timer aus und wird die entsprechende Transition zur Ausführung gebracht, erfolgt das Neusetzen des Timers als Vielfaches der Periode auf diesem Referenzzeitpunkt. Durch dieses Verfahren hat auch eine Transition, die verspätet auslöst, keine kumulative Wirkung auf nachfolgende Timer.

In der praktischen Anwendung reicht die gezeigte Lösung nicht aus, sie muß, wie in Abbildung 3.6(b) gezeigt, abgeändert werden. Werden hochfrequente Timer, wie hier, alle 100 ms gesetzt, sollte die erste Ausführung, wie gezeigt, verzögert stattfinden. Beim Start des SDL Systems wird das System initialisiert, dabei sowohl Blöcke, Prozesse, Gates, Queues usw. angelegt, aber auch die Starttransition der Prozesse ausgeführt, die direkt Signale im SDL-System versenden. Beim Start kommt es dabei zu einer kurzzeitigen maximalen Auslastung der realen Hardware, wodurch sich auch die Ausführung von Transitionen, die von Timern ausgelöst wurden, verzögert. Eine Verwendung von exakten Timern ohne ein verspätetes erstes Ausführen kann damit dazu führen, daß von dem Referenzzeitpunkt bereits mehrere Perioden abgelaufen sind, bevor der erste Timer verarbeitet wurde. In der Folge werden die nächsten Auslösezeitpunkte ebenfalls in der Vergangenheit gesetzt, was die Systemlast weiter erhöht. Abhängig von der Spezifikation und der verwendeten Hardware kann es einige Zeit dauern bis das System eingeschwungen ist. Stehen auf der Hardware sehr wenige Ressourcen zur Verfügung, können die hierdurch vergrößerten Warteschlangen sogar zum Systemabsturz führen.

Eine Überlast des Systems tritt zunächst beim Start auf, kann aber auch im laufenden Betrieb auf Grund von externen Ereignissen auftreten. Entweder handelt es sich hierbei um selten auftretende Ereignisse, die beim Design absichtlich so modelliert wurden oder unvermeidlich sind. Eine kurzfristige Überlast kann auch durch Service-Routinen auftreten, die beispielsweise den Speicher defragmentieren. Sind in dem System noch genügend freie Ressourcen vorhanden, fallen diese Situationen nur durch geringere Abweichungen (Jitter) um den eigentlichen Auslösezeitpunkt auf. Bindet jedoch das Ereignis zu viele Ressourcen, kann die Überlast persistent werden. Dieses Verhalten kann durch die angegebene Implementierung der exakten Timer noch verstärkt werden. Eine zusätzliche Überprüfung vor dem Setzen des exakten Timers, ob dieser in der Zukunft liegt (`>now`), kann eine Überlastsituation begrenzen. Hierbei sollten dann, sofern möglich, einzelne Timerereignisse ausgelassen oder kurzzeitig die Periode erhöht werden, was auf die jeweilige Anwendung anzupassen ist.

Die beschriebenen exakten Timer stellen eine Voraussetzung für den geeigneten Einsatz von explizitem Jitter dar, wie er in Kapitel 3.4.2 beschrieben wurde. Unter der Verwendung der exakten Timer wird der nächste Auslösezeitpunkt exakt angegeben und der Jitter als Schwankungsbreite um diesen Zeitpunkt angegeben. In Abbildung 3.6(c) ist dies für einen Timer mit einen Jitter von 10 ms noch einmal verdeutlicht.

Das hier diskutierte Problem, daß viele Aufwachvorgänge sich negativ auf den Stromverbrauch auswirken, wurde auch in modernen Betriebssystemen wie Linux [105] erkannt. Das ursprünglich periodische Ausführen einer Service-Routine wurde durch einen *On-Demand* Mechanismus ausgetauscht, wodurch ein *tickless-Kernel* entstand. Zusätzlich stellt der Kernel Funktionen bereit, die Ereignisse zusammenfassen und zu einem gemeinsamen Zeitpunkt ausführen, was die Anzahl der Aufwachvorgänge und dadurch den Stromverbrauch reduziert.

4

Kapitel 4.

Modellierung und Echtzeitfähigkeit von SDL-Systemen

Bei der Modellierung von Software wird das Gesamtsystem in einzelne kleinere Teilbereiche zerteilt, um somit die Komplexität zu verringern. Realisiert ein solcher Teilbereich eine bestimmte Funktion, spricht man von einem Modul oder einer Komponente. Können während der Aufteilung mehrere gleiche oder ähnliche Komponenten identifiziert werden, können diese bei der Entwicklung zu einer parametrierten Komponente zusammengefaßt und im Modell als Referenz auf die gemeinsame Komponente eingefügt werden. Entstehende Komponenten, die wiederkehrende Funktionen realisieren und in anderen Projekten gebraucht werden, werden in eine Bibliothek eingefügt. Nachfolgende Projekte können direkt die bereits vorhandenen Komponenten verwenden und sparen damit deren Entwicklungszeit. Jede Komponente wird spearat getestet und es existieren idealerweise auch Testfälle die eine korrekte Funktion sicherstellen. Sollte dennoch ein Fehler in einer Komponente gefunden werden, kann die fehlerbereinigte Version in allen Systemen eingesetzt werden. Für das Design von Protokollen in SDL hat Fliege et. al. in [34] einen Mikroprotokollansatz gezeigt, bei dem ein Protokoll in kleinere Unterprotokolle aufgeteilt wird. Das System kann dann die Mikroprotokolle verwenden und daraus ein komplettes Protokoll erstellen. In SDL lassen sich somit auch komponentenbasierte Designs erstellen; leider fehlen den gängigen Werkzeugen Funktionen, um diese bei der Entwicklung effizient einzusetzen. In Kapitel 4.1 wird eine neuer Ansatz zur komponentenbasierten Entwicklung von SDL-Systemen [12] vorgestellt.

In Kapitel 4.2 wird ein Ansatz zur effizienteren und exakteren Signalzustellung in SDL vorgestellt. Die Sprache SDL wird dazu formal um einen neuen Signaltyp, die *Echtzeitsignale*, erweitert. Im Unterschied zum priorisierten Empfang sind diese Signale direkt beim Versenden als solche in der SDL-Spezifikation zu erkennen. Die Laufzeitumgebung kann mit Hilfe dieser zusätzlichen Information eine deutlich höhere Genauigkeit bei der Zustellung der Signale erreichen. Die SDL-Echtzeitsignale wurden auf der internationalen SDL-Konferenz „SDL Forum 2011“ in Toulouse vorgestellt und im Konferenzband [62] veröffentlicht.

4.1. SDL-Module

In der Softwareentwicklung ist es Stand der Technik, ein komponentenbasiertes Design zu erstellen. Jede Komponente kann durch eine Komponente mit gleicher Schnittstellensignatur ausgetauscht werden.[19] Eine so spezifizierte Komponente kann ebenfalls in anderen Projekten wiederverwendet werden. In SDL wird dieses Design über SDL-Typen (`block type`, `process type`, etc.) realisiert, die dann in SDL-Paketen (`Packages`) organisiert sind. Ein SDL-Paket ist als Sammlung von Typen und Definitionen anzusehen, die bei Bedarf in einem SDL-System instantiiert werden. Vor der Instantiierung eines Typs aus einem Paket muß die Verwendung des Pakets zunächst dem System mittels `USE`-Anweisung bekannt gemacht werden. Viele Werkzeuge, wie auch das in der AG Vernetzte Systeme verwendete Werkzeug SDL-Tau Suite [52] bzw. das Real Time Developer Studio [89] fordern zusätzlich eine Einbindung jedes Pakets in den aktuellen Arbeitsbereich, was über die explizite Angabe des Speicherortes geschieht. Keines der Werkzeuge bietet die Möglichkeit, ein Paket anhand des Namens aus der `USE`-Anweisung und einem zentralen Speicherort selbst zu suchen. Pakete können in SDL jedoch nicht nur in Systemen referenziert werden, sondern auch in Paketen selbst. Es lassen sich somit komplexe Funktionen in wiederverwendbare Subfunktionen zerlegen und diese in verschiedenen Paketen zusammenzufassen. Bei der Einbindung eines so zerlegten Pakets in ein neues System müssen alle Referenzen geändert werden. Auf System- und Paketebene müssen die `USE` Anweisungen auf den neuen Paketnamen angepaßt und ggf. um neue Paketdefinitionen ergänzt werden. Innerhalb des SDL-Entwicklungswerkzeugs müssen noch die neuen Pakete dem Arbeitsbereich hinzugefügt und die alten entfernt werden. Unter dieser Beschränkung leidet die Austauschbarkeit von Paketen gleicher Funktionalität, da die Abhängigkeiten zunächst von Hand aufgelöst werden müssen. Das gleiche Problem besteht ebenfalls, wenn eine Partitionierung einer Funktionalität zum Zweck der unabhängigen Weiterentwicklung erfolgt, wie es beispielsweise bei der Entwicklung von Protokollen auf Mikroprotokollbasis [34] geschieht, denn auch hier müssen bei Änderungen eines Mikroprotokolls aufwändig alle neuen Referenzen eingefügt werden.

---

[19] Die Komponente mit gleicher Schnittstellensignatur muß jedoch nicht das gleiche Verhalten wie die alte besitzen.

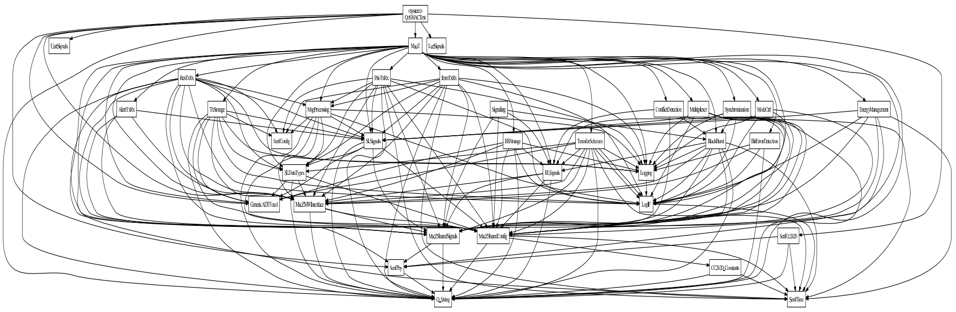


Abbildung 4.1.: Package-Abhängigkeitsgraph zu MacZ

Am Beispiel des realen in der AG Vernetzte Systeme entwickelten MAC-Protokolls MacZ [11, 69] erkennt man, wieviele Abhängigkeiten bereits ein Protokoll mittlerer Größe besitzt. In Abbildung 4.1 ist der Abhängigkeitsgraph zwischen den in diesem Protokoll verwendeten Paketen dargestellt. Jedes der dort gezeigten 35 Pakete hat wiederum Abhängigkeiten zu anderen Paketen – insgesamt zeigen sich mehr als 190 Verbindungen zwischen den Paketen. Um aus dem Medienzugriffsprotokoll ein komplettes Kommunikationsprotokoll und darauf aufbauend eine Anwendung aufzusetzen, wird MacZ ebenfalls in ein SDL-Package gepackt. Es steht damit als universelle MAC-Schicht zur Verfügung, bei der Instantiierung müssen jedoch alle Abhängigkeiten erfüllt sein. Ein weiteres Problem entsteht beim Hinzufügen der Abhängigkeiten: Da SDL die Eindeutigkeit eines Paketnamens fordert, kann es hier ebenfalls zu unerwünschten Nebenwirkungen kommen. Ein Vorteil der Kapselung in einzelne Komponenten stellt ja gerade die unabhängige Weiterentwicklung dar, doch genau hier entstehen bei der Aktualisierung vorhandener Komponenten in aufbauenden Systemen das Problem, daß alle Verknüpfungen aktualisiert werden müssen. Im folgenden wird nun vorgestellt, wie sich SDL-Packages als *SDL-Module* organisieren lassen, die über eine definierte Schnittstelle verfügen und sich automatisiert mit dem hierfür entwickelten ***S**DL **Pa**ckage **S**ubstitution Tool* (SPaSs) [85] in SDL-Systeme mit gleicher Schnittstelle einsetzen lassen.

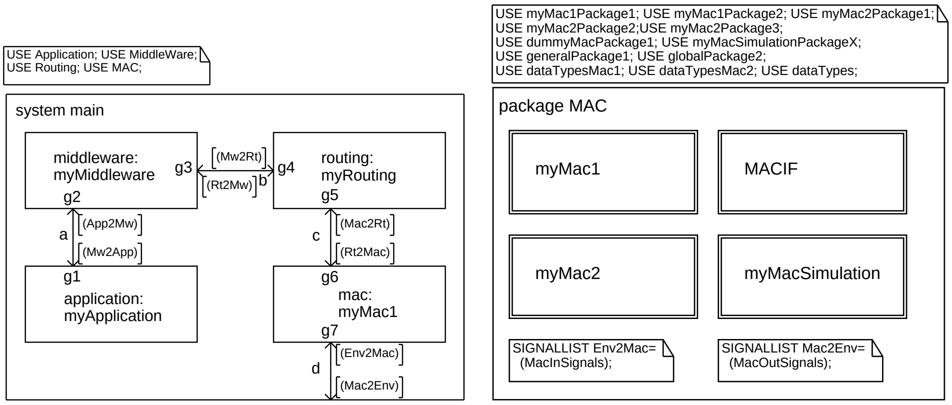
Eine MAC-Schicht in SDL als generische Kommunikationsschicht anzulegen erfordert ein aufwändiges Design: Auf der einen Seite soll die SDL Spezifikation unabhängig von der verwendeten Hardware sein, auf der anderen Seite sollen keine Ressourcen ungenutzt bleiben. Teilweise kann die Konfiguration dabei über die SDL-Konfigurationsschnittstelle (Kapitel 6.2) erfolgen. Steht die Hardware aber gar nicht zur Verfügung und das Protokoll soll unabhängig von einer speziellen Hardware in einem Simulator getestet werden, muß dies ebenfalls in der Spezifikation vorgesehen werden. In der Folge wird die Spezifikation größer und enthält auch im realen System Teile, die eigentlich nur für die Simulation nötig sind. Damit können nicht nur Fehler im Protokoll einfacher gefunden werden, es ist so auch möglich, das Protokoll mit verschiedenen MAC-Schichten zu verwenden, die sowohl drahtlos als auch drahtgebunden sein können. Es entstehen mehrere Konfigurationen, die in jeweils eigenen SDL-Systemen von Hand zusammengebaut werden müssen – obwohl die jeweiligen Komponenten bereits vorhanden sind. Sind nur wenige Konfigurationen möglich oder sinnvoll, ist dies eine praktikable Möglichkeit. Doch die Erfahrung zeigt, daß selbst kleine Systeme mit der Zeit wachsen und damit auch die Anzahl der möglichen Konfigurationen. Eine Alternative besteht darin, nur ein System zu spezifizieren und je nach gewünschter Konfiguration die Typen der jeweiligen Instanzen auszutauschen. Auch hier manifestieren sich die Nachteile erst bei großen Systemen – der Austausch muß händisch vorgenommen werden, er ist fehleranfällig und es kann nötig sein, in weiten Teilen des Systems Änderungen vorzunehmen.

Das neue Konzept der *SDL-Module* versucht diese Probleme zu eliminieren, wobei weder neue Sprachkonstrukte, noch Änderungen an bestehenden Werkzeugen nötig sind. Grundsätzlich soll es möglich werden, automatisiert Teile einer Spezifikation mit Teilen einer anderen Spezifikation auszutauschen und am Ende ein vollständiges und funktionierendes SDL-System zu erhalten. Da es nötig sein kann, diesen Vorgang zu wiederholen, soll es möglich sein, mehrere Konfigurationen abzulegen und diese bei der Ausführung auszuwählen, wie es bei dem Compiler-Werkzeug GNU make [39] üblich ist.

Am Beispiel einer Kommunikationsarchitektur wird nun zunächst gezeigt, welche Nachteile die Lösungen innerhalb von SDL haben. Danach wird als Lösung das Konzept der *SDL-Module* vorgestellt und wie diese mit dem dafür entwickelten Hilfswerkzeug SPaSs anwenden lassen. Abschließend wird der Algorithmus an einem praktischen Beispiel evaluiert.

4.1.1. Module innerhalb einer Kommunikationsarchitektur

Eine typische Kommunikationsarchitektur besteht aus den vier Hauptkomponenten: Anwendung, Zwischenschicht (*Middleware*), Routing und einer MAC-Schicht. In Abbildung 4.2(a) ist die Spezifikation dieser Hauptkomponenten als SDL-Blockinstanzen gezeigt. Jeder Block repräsentiert eine Kommunikationsschicht. Der Signalaustausch erfolgt ausschließlich zwischen direkt benachbarten Blöcken. Signale können somit nicht direkt aus der MAC-Schicht an die Applikationsschicht gesendet werden, sondern müssen dem Schichtaufbau folgend zunächst durch die Routing- (`myRouting`) und dann durch die Middleware-Schicht (`myMiddleware`) geleitet werden. In jeder Schicht wird das Signal empfangen, ggf. verarbeitet und ein neues Signal an die darüber liegende Schicht gesendet.



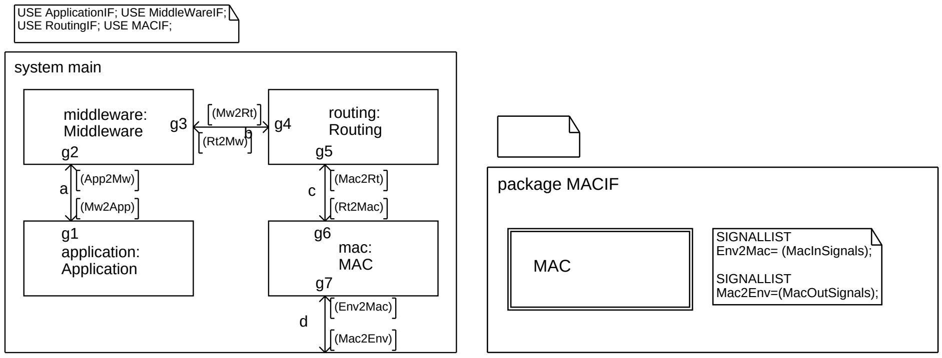
(a) Typische Kommunikationsarchitektur

(b) Verschiedene MAC-Protokollspezifikationen in einem Package

Abbildung 4.2.: Mehrere SDL Spezifikationen in einem Package

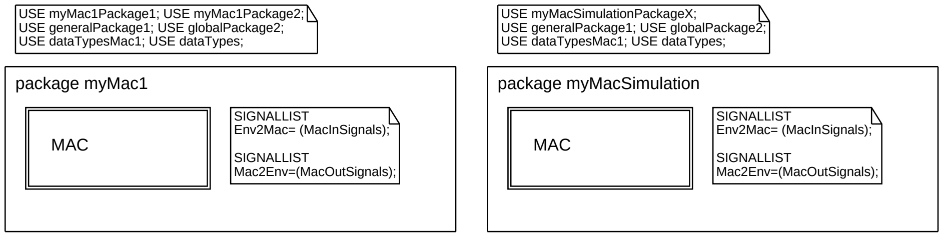
Die Möglichkeiten der Austauschbarkeit innerhalb von SDL sollen nun am Beispiel der MAC-Schicht erläutert werden: Innerhalb des SDL-Packages `MAC` stehen verschiedene Instanzen von MAC-Schichten mit der gleichen Schnittstelle zur Verfügung (siehe Abbildung 4.2(b)). Durch die Instantiierung eines der Blocktypen, wie es in Abbildung 4.2(a) mit `myMac1` geschehen ist, läßt sich eine der MAC-Schichten aus dem Paket auswählen. Innerhalb des `MAC`-Pakets müssen alle Abhängigkeiten aller Blocktypen erfüllt sein, weshalb in diesem Paket **12** `USE`-Anweisungen eingefügt sind. Auch wenn alle MAC-Schichten die gleiche Schnittstelle implementieren, ist es möglich, daß eine MAC-Schicht weitere Signale verarbeitet. Die Signallisten der MAC-Schichten müssen dementsprechend alle Signale enthalten, auch wenn eine spezifische MAC-Schicht das Signal gar nicht verarbeitet.

Sollen viele größere MAC-Schichten in einem Paket verwaltet werden, kann dies leicht unübersichtlich werden. Es besteht auch im Gegensatz dazu die Möglichkeit, die Spezifikation jeweils in einem eigenen Paket zu organisieren, wie dies in Abbildung 4.3 gezeigt ist. In den Abbildungen 4.3(b), 4.3(c) und 4.3(d) sind drei der MAC-Protokolle aus Abbildung 4.2(b) gezeigt, wobei hier der Paketname das jeweilige MAC-Protokoll repräsentiert. Bei dieser Lösung werden die spezifischen Signallisten in jedem Package angegeben. Jede Signalliste enthält dabei die schnittstellenspezifischen und die für diese eine MAC-Schicht spezifischen Signale, aber keine Signale, die in anderen MAC-Schichten verwendet werden. Ein Austausch des MAC-Protokolls innerhalb des Systems (Abbildung 4.2(a)) wird hierbei nur durch einen Austausch



(a) Kommunikationsarchitektur, angepaßt auf nur eine Spezifikation pro SDL-Package

(b) Schnittstelle für alle MAC-Schichten



(c) Spezifikation von `myMac1`

(d) Spezifikation von `myMacSimulation`

Abbildung 4.3.: Basis für das SDL-Modul-Konzept, nur eine Spezifikation pro SDL-Package

der `USE`-Anweisung erledigt.[20] Besitzen alle Blocktypen wie in dem Beispiel gezeigt den gleichen Namen `MAC`, ist es nicht nötig, den Namen des Blocktyps innerhalb des Systems zu ändern. Werden in einem System Signale verwendet, die nicht in der Schnittstelle definiert sind, ist ein Austausch des Blocks nicht mehr ohne weiteres möglich.

SDL bietet mit den vorhandenen Sprachmitteln keine Möglichkeit, eine Schnittstelle zu definieren und dafür eine Spezifikation zu entwerfen. Die einzige Möglichkeit, die sich bietet, eine Schnittstelle anzugeben, ist die Definition von Signalen, Signallisten und Signalwegen, sodaß Blöcke und Prozesse diese verarbeiten müssen. Eine andere Möglichkeit, eine Schnittstelle zu definieren ist es, eine funktional leere, syntaktisch aber vollständige Spezifikation eines MAC-Layers anzugeben, wie es in

---

[20] In den Entwicklungsumgebungen ist es zusätzlich noch nötig, die Datei, in der das Paket spezifiziert ist, zum Arbeitsbereich hinzuzufügen.

Abbildung 4.3(b) gezeigt ist. Während der Entwicklung muß sich die MAC-Schicht immer durch die MAC-Schnittstelle austauschen lassen.[21]

Beide Ansätze der Organisation mehrerer verschiedener MAC-Schichten haben sowohl Vor- als auch Nachteile. Im ersten Ansatz, alle Spezifikationen in ein Paket zu packen, hat der Entwickler direkt einen Überblick über alle vorhandenen und verfügbaren MAC-Protokolle. Sowohl die Struktur als auch die Signallisten müssen nur einmal angelegt werden und sind im kompletten Paket gültig. Der zweite Ansatz hat den Vorteil, daß nur die Teile, die für das jeweilige MAC-Protokoll benötigt werden, in dem Paket vorhanden sind. Er ist dadurch kleiner, besser strukturiert und bindet nur die direkt abhängigen Pakete ein, wodurch auch die spätere textuelle Umsetzung (in SDL-PR[22]) kompakter ist und die SDL-Werkzeuge schneller arbeiten. Ist es erforderlich die Schnittstelle der MAC-Schicht um ein weiteres Signal zu erweitern, auch wenn dies nur zur Fehlersuche geschieht, zeigt sich ein immanenter Nachteil des ersten Ansatzes: Auch wenn dieses Signal nur in einer Blockinstanz benötigt wird, müssen auch alle anderen in dem Paket definierten Blocktypen an die geänderte Signalliste angepaßt werden. Dieser Nachteil entfällt beim zweiten Ansatz, jedoch ist hier die Konsistenz anderer Pakete zur geänderten Schnittstelle nicht mehr sichergestellt. Der Austausch von Modulen in SDL über das hier vorgestellte Konzept wird über den zweiten Ansatz erfolgen.

4.1.2. Austauschbare Module in SDL

Das Austauschen von Typen in SDL-Spezifikationen läßt sich bereits durch recht einfache Operationen erledigen. Wurde beispielsweise ein MAC-Protokoll klassisch in SDL entwickelt, enthält die Spezifikation des Protokolls zum einen Adaptionen an spezielle Hardware, Berechnungen, die zur Fehlersuche vorhanden sind, aber auch Kode der ausschließlich in Simulationsumgebungen benötigt wird. Während der Übersetzung dieses Pakets ist es nicht möglich, alle überflüssigen Teile zu erkennen und diese zu entfernen, was sich gerade in Ressourcenkritischen Bereichen negativ auswirkt. Bei der Entwicklung einer MAC-Schicht und Middleware für die Imote2-Plattform, wie es in Kapitel 5.2.2 beschrieben ist, entstand durch überflüssige Pakete Überlast im System. Einige der Pakete waren für die Unterstützung einer PC-Simulation nötig, andere wurden speziell für die Fehlersuche eingebunden, um eine Fehlerausgabe zu ermöglichen. Auf der Zielplattform, dem Imote2, sollten diese Teile nicht im System verfügbar sein. Die Spezifikation zu kopieren, um diese

---

[21] Mit Hilfe des SPaSs-Werkzeugs geschieht dies automatisch. In der Spezifikation im SDL-System bleibt immer nur die Schnittstelle stehen.

[22] SDL-PR ist die textuelle Repräsentation von der graphischen Darstellung. Sofern der Export mittels *CIF* exportiert wird, enthält die PR Repräsentation auch grafische Informationen.

Teile aus der Spezifikation zu entfernen, würde die weitere Entwicklung erschweren, da dann beide Spezifikationen synchron gehalten werden müßten. Angelehnt an die Werkzeuge *Maven* [2] und *make* [39] sollte mit Hilfe des Versionskontrollsystems *Subversion* [3] ein Werkzeug zum automatischen Austausch von Modulen und der Auflösung von Konflikten für SDL entstehen. Jedes SDL System sollte sich, je nach Anforderung, *einfach* zwischen verschiedenen Zielen, wie beispielsweise *Simulations*-, *Debug*- oder *Release*-Ziel, umschalten lassen. Um dieses Ziel der Austauschbarkeit als *SDL-Module* zu erreichen, mußten erst einige Grundlagen geschaffen werden, die im folgenden erläutert werden.

4.1.2.1. Konzepte zu SDL-Modulen

Anders wie die meisten Programmiersprachen unterstützt SDL keine speziell definierten Schnittstellen (*Interfaces*), denen eine Spezifikation genügen muß. In SDL ergibt sich die Schnittstelle aus der System-Struktur, den Signalen und deren Parametern, die zwischen den Einheiten (Blöcke, Prozesse etc.) ausgetauscht werden.

```
PACKAGE MACIF;
  /* Interface for simple MAC Layers, defines 2 signals for communication:
   * inSignal and outSignal
   * used by: SimpleMac, SimpleMac2
  */
  SIGNAL inSignal;
  SIGNAL outSignal;
  BLOCK TYPE MAC;
    GATE inGate IN WITH inSignal;
    GATE outGate OUT WITH outSignal;
    SIGNALROUTE rt FROM env TO emptyProc WITH inSignal;
    SIGNALROUTE rt FROM emptyProc TO env WITH outSignal;

    PROCESS emptyProc;
      start;
      NEXTSTATE idle;
      STATE idle;
    ENDPROCESS emptyProc;

  ENDBLOCK TYPE MAC;
ENDPACKAGE MACIF;
```

Listing 4.1: Beispiel einer SDL-Module-Schnittstelle in SDL-PR

Listing 4.1 zeigt ein Paket mit einem Blocktyp `MAC`, der als funktionsloser Platzhalter in einer Spezifikation und somit als quasi Schnittstelle verwendet werden kann. In dieser Schnittstelle, sind lediglich die beiden Signale `inSignal` und `out_signal` zum Senden und Empfangen von Daten über eine MAC-Schicht enthalten. Der innerhalb des Blocktyps enthaltene Prozess ist minimal und enthält lediglich den zwingend erforderlichen Startzustand und einen Folgezustand `idle`. Der Prozess enthält keine weitere Funktionalität, ist jedoch syntaktisch korrekt und läßt sich damit in jede SDL-Spezifikation integrieren. Weiterhin sollte das SDL-Package einen Kommentar enthalten, der die Schnittstelle kurz beschreibt. Es erweist sich als sehr hilfreich, wenn zusätzlich zu der Beschreibung auch angegeben wird, welche Pakete die Schnittstelle verwenden, damit diese bei Änderungen an der Schnittstelle ebenfalls angepaßt werden können.

Jedes SDL-Package, das die gleichen Signale bzw. Signallisten und Blocktypen wie die Schnittstelle definiert und die in der Schnittstelle beschriebene Funktionalität spezifiziert, bezeichnen wir als *SDL-Modul*. Im Unterschied zu einer Komponente in der Softwareentwicklung sind bei einem SDL-Modul auch alle benötigten abhängigen Pakete enthalten, es ist damit in sich abgeschlossen. Aus den grafischen SDL-Werkzeugen lassen sich SDL-Module als textueller SDL-PR-Kode meist direkt erzeugen, indem die Kodegenerierung für ein SDL-Package ausgewählt wird.

4.1.2.2. Anwendung der SDL-Module

Ein SDL-Modul bezeichnet also eine abgeschlossene Funktionalität, die einer Schnittstellendefinition genügt. Da eine SDL-Schnittstelle selbst bereits syntaktisch komplett ist, implementiert sie die Schnittstellendefinition (die sie vorgibt). Fügt man nun die minimale Funktion hinzu, die informell im Modul beschrieben ist, erfüllt die Schnittstelle selbst alle Anforderungen an ein SDL-Modul und kann während der Entwicklung bereits verwendet werden. Abbildung 4.4 zeigt die Verwendung solcher Schnittstellen-Module bei der Erstellung eines neuen Routing-Protokolls.

Das dargestellte System `RoutingDevelopment` besteht sowohl aus einer MAC-Schicht, einer Middleware und dem zu entwickelnden Routing selbst. Die beiden Blöcke `myMiddleware` und `myMAC` sind angelehnt an die minimale Schnittstelle, aus Listing 4.1 realisiert und enthalten ebenfalls keine Funktionalität. Der Entwickler hat damit ein minimales System, das es ihm ermöglicht sein Modul zu entwickeln, SDL-Analyse-Werkzeuge[23] zu verwenden, um Syntaxfehler oder falsche und fehlende Signale zu finden. Außerdem stellt die Verwendung der jeweiligen Schnittstellenmodule bereits sicher, daß das Routing-Modul mit anderen Modulen, die diese Schnittstellen implementieren, zusammenarbeitet.

Durch die Verwendung der Schnittstellen-Module während der Entwicklung beschränkt sich Analyse, Test und Fehlersuche ausschließlich auf das zu entwickelnde

---

[23] beispielsweise der Tau-*Analyzer*

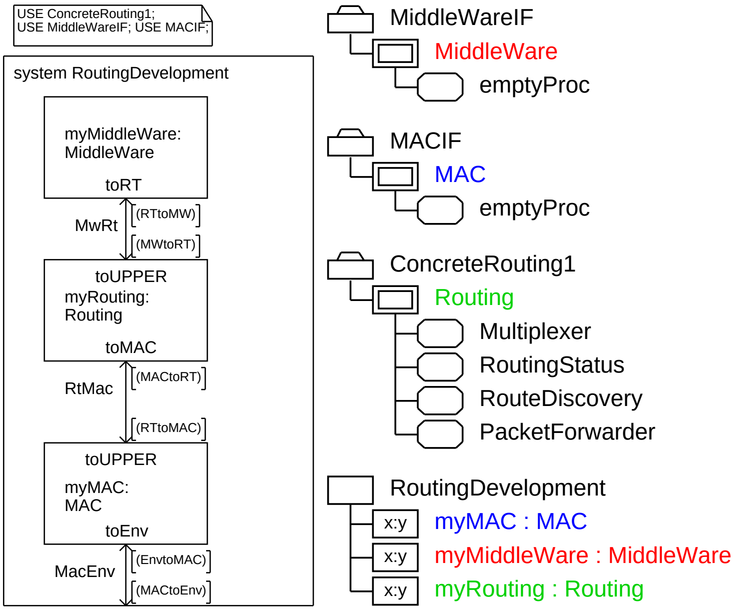


Abbildung 4.4.: Überblick auf ein System während der Entwicklung eines Routing-Protokolls

Modul. Wird diese Trennung nicht verwendet, zeigen die Werkzeuge häufig Fehler an Stellen an, die unverändert übernommen wurden und nicht fehlerhaft sind. Grund dafür ist, daß sowohl Scanner als auch Parser im Fehlerfall einen geeigneten Wiederaufsetzpunkt suchen [79] und diesen als Fehlerpunkt ausgeben. Abhängig von der Implementierung des Scanners / Parsers, der Sprache selbst und der Komplexität des zu analysierenden Systems kann der Fehler sehr weit von der angezeigten Stelle entfernt sein und die Suche des Problems verlängert sich. Auch für die SDL-Werkzeuge wirkt sich ein kleineres System positiv aus: Da die Werkzeuge kleinere Spezifikationen prüfen müssen, benötigen die Werkzeuge weniger Ressourcen und die Analyse- sowie Testkodeerstellung geht schneller.

Soll das Modul aus Abbildung 4.4 getestet werden, stehen zwei Vorgehensweisen zur Wahl: Das Routing-Protokoll wird in ein vollständig spezifiziertes System integriert und mittels Simulator (wie z.B. *ns*+SDL [68]) geprüft, ob alle Daten ihr Ziel finden. Die komplette Funktionsweise des Systems wird so über *Blackbox-Testing* überprüft. Kommen Daten nicht an ihrem Ziel an, kann der Fehler in jeder beteiligten Komponente vorhanden sein. Für die Fehlersuche ist man auf Debug-Ausgaben aus dem

SDL-System und des Simulators angewiesen. Eine Alternative stellt der Austausch der Module `myMiddleware` und `myMAC` durch Generator- bzw. Test-Module dar, die definierte Daten generieren und empfangen. Werden sowohl Generator als auch Testmodul speziell für die Implementierung geschrieben und decken damit einige Problemfälle ab, handelt es sich um einen *White-Box-Test*.[24] Durch eine automatisierte Ersetzung der SDL-Module in bestehende Systeme können viele verschiedene Tests mit einem geänderten Modul automatisch durchgeführt werden. In vielen Fällen können Fehler direkt auf die Komponente zurückgeführt und behoben werden, bevor sie in anderen Systemen Schaden anrichten.

Ist ein SDL-Modul soweit entwickelt, daß es auch in anderen SDL-Systemen eingesetzt werden kann bzw. soll, wird es in textueller SDL-Repräsentation (SDL-PR-Kode) auf einem zentralen Datei-Server oder in einem Versionskontrollsystem (z.B. Subversion) abgelegt. Werden die SDL-Module in einem Versionskontrollsystem verwaltet, lassen sich zum einen Änderungen zwischen den Versionen des Moduls nachverfolgen, zum anderen ist es aber auch einfach möglich, eine spezielle Version des Moduls zu verwenden oder zwischen Entwicklungs- und Finalversionen eines SDL-Moduls zu unterscheiden. Es ist somit immer möglich, bei Fehlern, die durch die Verwendung einer neuen SDL-Modul-Version entstanden sind, weiterhin die Vorgängerversion zu verwenden und damit die eigene Entwicklung nicht einzuschränken.

4.1.3. Anwendung in der Praxis

Um die praktische Anwendung der SDL-Module zu vereinfachen, wurde in Zusammenarbeit mit Philipp Becker ein plattformunabhängiges Werkzeug *SPaSs* (***S****DL* ***Pa****ckage* ***S****ubstitution Tool*) entwickelt, das direkt aus der SDL-Tau Suite aufgerufen werden kann. Bisher wurden nur einfache Ersetzungen betrachtet, bei denen es darum ging, ein oder zwei SDL-Module durch andere zu ersetzen. Mittels SPaSs ist es jedoch auch möglich, diesen Vorgang rekursiv anzuwenden. Zu jedem SDL-Modul gehört deshalb eine optionale Konfigurationsdatei, die ähnlich wie *Makefiles* aufgebaut ist und Ziele (*targets*) definiert. Jedes Ziel (z. B. release, debug, default, ...) klassifiziert dabei, welche Ausprägung eines SDL-Moduls erstellt werden soll. Das SPaSs-Werzeug erhält beim Aufruf das Ziel, worauf der Algorithmus rekursiv alle benötigten Pakete ermittelt und deren Abhängigkeiten auflöst. Alle doppelten sowie nicht referenzierten SDL-Packages werden dabei aus der Spezifikation entfernt.[25] Ein SDL-Package wird dabei als doppelt erkannt, sobald dessen Name mehr als einmal

---

[24] Eine vollständige Abdeckung mittels White-Box-Test ist sehr aufwändig und Laufzeitintensiv und deshalb nur in kleinen Einzelsystemen möglich.

[25] SDL verbietet es, einen Paketnamen mehrfach zu verwenden, selbst wenn die Spezifikation identisch ist.

in dem entstehenden System vorhanden ist. Zur Lösung dieses Konfliktes werden alle Pakete bis auf eines aus der Spezifikation entfernt, sofern diese bis auf darin enthaltene Kommentare gleich sind. Sollten die enthaltenen Pakete nicht gleich sein, wird unter Ausgabe einer Warnung versucht, die aktuellste Version des Pakets zu verwenden. Kommt es in Folge dessen zu Problemen bei der Übersetzung der Spezifikation, muß der Entwickler selbst diesen Konflikt auflösen, was sich durch eine Definition der spezifischen Version eines SDL-Moduls in der Konfigurationsdatei bewerkstelligen läßt. Die detaillierte Beschreibung des Ersetzungsalgorithmus und dessen Pseudokode ist in der Veröffentlichung [12] zu finden.

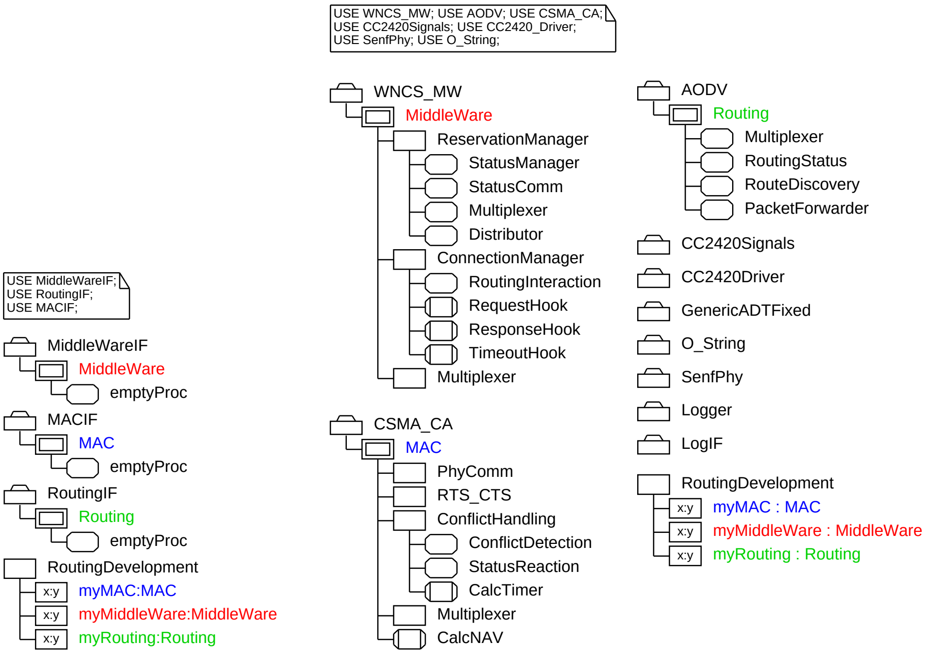
4.1.3.1. Beispiel

```
 1 <?xml version="1.0" standalone="yes"?>
 2 <Config>
 3   <Global>
 4     <Src name="svnSDL" type="svn" src="http://rns3/svn/SDL" branch="trunk" user="guest"
         pwd="guest" />
 5   </Global>
 6
 7   <Mapping>
 8     <Entry search="Synchronization" replace="BBS" file="bbs.pr" src="svnSDL"/>
 9     <Entry search="LoggerIF" replace="NullLogger" file="logger.pr" src="svnSDL"/>
10   </Mapping>
11
12   <Mapping target="verbose">
13     <Entry search="LoggerIF" replace="VerboseLogger" file="logger.pr" target="debug"
         src="svnSDL"/>
14   </Mapping>
15 </Config>
```

Listing 4.2: Konfiguration einer MAC-Schicht

Am Beispiel der MAC-Schicht soll nun der Aufbau und die Funktionsweise einer solchen Konfigurationsdatei erläutert werden. In Listing 4.2 ist diese Konfiguration, die im XML-Format erfolgt, gezeigt. Innerhalb des Wurzelelements `Config` darf einmalig das `Global`-Element (Zeile 3) stehen, in dem die Speicherorte der SDL-Module benannt sind. Das `Mapping`-Element (Zeile 7 und 12) darf mehrfach definiert sein und benennt über das `target`-Attribut das Ziel, für das die darin enthaltene weitere Definition gilt. Für die weitere Verwendung innerhalb der Konfigurationsdatei erhält das zentrale Subversion-Repository den Namen *svnSDL* (Zeile 4. Der Mapping-Eintrag in Zeile 7 ist der Standard-Mapping-Eintrag, der verwendet wird, sofern das Werkzeug beim Aufruf kein bestimmtes Ziel erhalten hat. Weiterhin erben alle anderen Mappings die Einstellungen des Standard-Eintrags, in diesem müssen somit nur noch die Änderungen angegeben werden. Innerhalb eines Mappings definieren die Einträge (`Entry`), welche SDL-Packages (`search`) durch andere (`replace`) aus der Datei `file`, die aus der Quelle (`Src`) zu beziehen ist, ersetzt werden soll (siehe Zeile 8). Das Attribut `src` muß dabei einer Definition, wie sie im `Global`-Bereich angelegt ist, entsprechen. Jede Modul-Datei kann zusätzlich noch eine Konfigurationsdatei mit gleichem Dateinamen aber der Endung `.cfg` besitzen. Ist diese Konfigurationsdatei vorhanden, werden hier ebenfalls alle Regeln entsprechend durchlaufen. Über das Attribut `target` in der `Entry`-Elementdefinition läßt sich steuern, welches Ziel innerhalb der Subkonfiguration verwendet werden soll (siehe Zeile 13).

In diesem Beispiel existieren zwei mögliche Ziele: das Standard-Ziel und ein „verbose"-Ziel. Während im Standardziel keine Ausgaben zur Verfolgung erfolgen (`replace="NullLogger"`), wird im „verbose"-Ziel ein spezieller Ausgabetreiber (`replace="VerboseLogger"`) verwendet.



(a) System mit Platzhaltern vor der Ersetzung

(b) System mit konkreten Spezifikationen nach der Ersetzung

Abbildung 4.5.: SPaSs-Ersetzungsalgorithmus am Beispiel

Zur Verdeutlichung der Arbeitsweise von SPaSs wird in Abbildung 4.5 die Struktur des Tau-Organizers gezeigt. Die linke Seite (Abbildung 4.5(a)) zeigt das ursprüngliche System unter Verwendung von SDL-Modulen. In diesem System befinden sich ausschließlich Schnittstellen-Module, die syntaktisch korrekt sind und als Platzhalter für für die jeweilige konkrete Funktionialität dienen. Rechts daneben zeigt Abbildung 4.5(b) das gleiche System, allerdings sind hier durch SPaSs alle Schnittstellen-Module durch konkrete Implementierungen ersetzt, und alle davon abhängigen Pakete ebenfalls zum System hinzugefügt worden. Es handelt sich hier im Vergleich zu dem in Abbildung 4.1 gezeigt Beispiel immer noch um ein kleines System, in dem bereits allein für die MAC-Schicht 35 Pakete benötigt wurden. Trotzdem wird bereits hier die Struktur groß und zunehmend unübersichtlich.

4.1.4. Fazit

Obwohl sich SDL sehr gut eignet um strukturierte, große Kommunikationssysteme zu entwickeln, fehlen noch Werkzeuge, die in der praktischen Anwendung für den komponentenbasierten Entwurf benötigt werden. Das Werkzeug SPaSs versucht mit Hilfe des Konzepts der SDL-Module, diesen Mißstand auszugleichen und wurde der SDL-Gemeinschaft frei zur Verfügung [85] gestellt. Der vorgestellte Ansatz und die Unterstützung des Werkzeugs fördert dabei die automatische Testbarkeit und verringert damit Fehler im finalen System. Die kleinere Spezifikation fördert die Übersichtlichkeit während der Entwicklung und verringert die Wartezeit einzelner Operationen der SDL-Werkzeuge. Idealerweise enthält das fertige Produkt keine während der Entwicklung benötigten, aber sonst überflüssigen Teile und profitiert damit von geringerem Ressourcenverbrauch (Programmgröße, Hauptspeicher, CPU), wodurch sich die Laufzeit von batteriebetriebenen Systemen verlängert.

4.2. Echtzeitsignalisierung in SDL

Mit der zunehmenden Umstellung von mechanischen Komponenten auf elektrische Mikrocontroller gesteuerte Komponenten, wie dies in der Automobil- oder Flugzeugindustrie geschieht, werden stabile echtzeitfähige Systeme immer wichtiger. Unter einem Echtzeitsystem versteht man ein System, welches das Ergebnis einer Berechnung bis zu einem vorher definierten Zeitpunkt garantieren muß [101]. Der Auslöser für die Berechnung kann ein abgelaufener Timer (*zeitgesteuert*) oder bei reaktiven Systemen ein externes Ereignis (*ereignisgesteuert*) sein. Die zeitliche Reihenfolge der Ereignisse muß eingehalten werden [60]. Eingebettete Systeme sind meist reaktive Systeme, die nur dann eine Reaktion zeigen, wenn ein externes Ereignis, wie beispielsweise ein Tastendruck eintritt. Vielfach werden zusätzlich periodische Vorgänge benötigt, es handelt sich damit nicht mehr um ein ausschließlich reaktives

System. Ein *Echtzeitsignal* definiert ein Signal, das von einem Prozess an einen anderen gesendet wird und zu einem angegebenen Zeitpunkt zugestellt werden soll.

Die ereignisgesteuerte Programmierung, erweitert um zeitgesteuerte Aufgaben, ist die Grundlage für Echtzeitsysteme. Beides ist in SDL bereits vorhanden, jedoch fehlen weitere Möglichkeiten, eben diese Grundlagen sinnvoll für Echtzeitsysteme einzusetzen. In SDL läßt sich bisher keine Aussage über die Ankunftszeit eines Signals treffen, im SDL-System läßt sich lediglich der Zeitpunkt der Signalerstellung (`now`) ermitteln. Für jede Form von Echtzeitsystemen ist aber gerade die Ankunftszeit einer Signalisierung entscheidend. Bereits 2009 hat P. Becker et al. [10] eine Erweiterung der Sprache SDL vorgeschlagen, um echtzeitfähige Protokolle mit SDL zu entwickeln.

Die folgenden Erweiterungen erlauben es, in SDL auf einfache Weise Echtzeitsignale zu definieren und ermöglichen es damit, Echtzeitspezifikationen zu erstellen.[26] Die Erweiterung ist auf der internationalen SDL-Konferenz 2011 in Toulouse vorgestellt und veröffentlicht [62] worden.

4.2.1. Konzeption

Ein System besteht aus Aufgaben (*Tasks*), die entweder periodisch oder ereignisgesteuert ausgeführt werden.. In SDL können *Tasks* entweder aus einer einzelnen Transition (*einfache Tasks*) oder auch aus mehreren Transitionen (*komplexe Tasks*) bestehen, die in anderen Prozessen liegen und durch Signale angestoßen werden. Die Aktivierung von Tasks kann dabei direkt durch ein Ereignis (normales SDL Signal), einen Timer oder indirekt durch einen in SDL spezifizierten *Scheduler* erfolgen. Bei der direkten Aktivierung der Tasks durch Timer wird der Ablaufplan (Schedule) der Tasks über das gesamte System verteilt, wodurch dessen Wartbarkeit sehr erschwert wird. Eine indirekte Aktivierung hingegen wird von einem oder wenigen zentralen Schedulern ausgeführt. Sieht man von externen Signalen, die Tasks direkt anstoßen ab, läßt sich so der Ablaufplan des Gesamtsystems einfach überprüfen und anpassen.

4.2.1.1. Zeitgesteuerter Scheduler in SDL

In der SDL Virtual Machine (SVM) ist bereits ein zentraler Scheduler vorhanden, der die Ausführung der einzelnen SDL-Prozesse steuert. Eine Integration der Ablaufsteuerung in diesen Scheduler ist zwar möglich, muß aber für jedes System erneut

---

[26] In der Spezifikation ist noch nicht festgelegt, ob es sich um ein weiches oder hartes Echtzeitsystem handelt.

angepaßt werden. Der Scheduler ist dabei nicht selbst in SDL spezifiziert, sondern abhängig vom jeweils verwendeten Werkzeug in einer höheren Programmiersprache geschrieben. Es besteht aber durchaus auch die Möglichkeit, den Scheduler selbst als Prozess in SDL zu spezifizieren. Wie jeder andere SDL-Prozess wird auch der Scheduler von der SVM gesteuert und aktiviert seinerseits durch Signale andere Prozesse.

Zunächst werden nun verschiedene Möglichkeiten gezeigt, wie ein solcher Scheduler in SDL spezifiziert werden kann, welche Anforderungen an die SDL-Prozesse dabei gestellt werden und wie präzise die Aktivierung der Tasks erfolgt. Gerade für Echtzeitsysteme stellt die präzise Aktivierung ein wichtiges Kriterium dar. Soll nun ein einfacher Task `T` regelmäßig, nach einer intialen Wartezeit von 0.5 s, von einem Scheduler mit der Periode 1 s ausgeführt werden. Die in SDL naheliegendste Lösung einen solchen Scheduler zu realisieren ist in Abbildung 4.6 gezeigt.

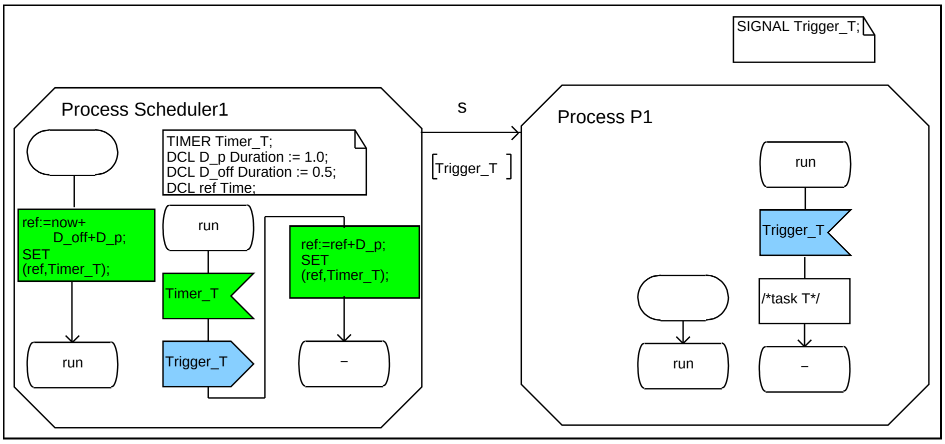


Abbildung 4.6.: `Timer_T` im Scheduler steuert die Ausführung der Task `T`.

Der Scheduler setzt seinen initialen Timer über die Periode `D_p` und die Verzögerung `D_off` auf den ersten Auslösezeitpunkt für den Task. Sobald dieser Timer nun auslöst, wird ein SDL-Signal `Trigger_T` an den Prozess `P` mit der auszuführenden Task geschickt. Für den nächsten Auslösezeitpunkt setzt der Scheduler den Timer um eine Periode weiter indem `D_p` auf den Referenzzeitpunkt addiert wird. Die Verwendung von `now` ist nicht präzise, da bereits Zeit vergangen ist, bis die Transition feuert. Damit würde bereits der Scheduler nicht die korrekte Periode verwenden. Sobald nun der Prozess `P` ausgeführt wird und das Signal empfängt, kann die zugehörige Transitions ausgeführt werden. Diese Lösung scheint genau das erwartete Verhalten zu zeigen. Bei genauer Betrachtung zeigt sich jedoch, daß diese Lösung viele Ressourcen benötigt und schließlich den Task durch Verzögerungen erst später als erwartet ausführt. Zu dem Zeitpunkt, zu dem eigentlich der Task `T` ausgeführt werden sollte, ist der Scheduler noch aktiv. Es muß zunächst ein Signal erzeugt, der Signalweg zum Zielprozess ermittelt und dieses darüber innerhalb des SDL-Systems transportiert werden. Es handelt sich hier um keinen verzögernden Kanal, trotzdem ist in der Praxis zu erwarten, daß für die Weiterleitung des Signals Zeit verbraucht wird. Schließlich muß der Zielprozess von dem Scheduler der SVM zur Ausführung gebracht werden. Der Prozess muß die Transition für das Signal auswählen, dieses schließlich konsumieren und die Transition ausführen. Der Task wird somit auf jeden Fall zu spät ausgeführt – sofern sich das System in Ruhe befindet, jedoch mit der richtigen Periodizität. Stehen weitere Tasks parallel zur Ausführung bereit, kann keine Aussage mehr über den Zeitpunkt der Ausführung getroffen werden, obwohl sich das SDL-System korrekt an die Spezifikation hält.

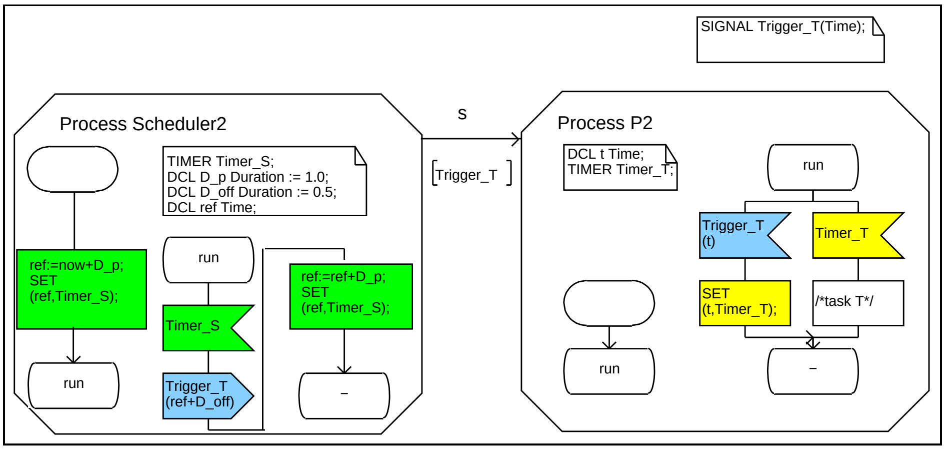


Abbildung 4.7.: Externer Trigger und lokaler Timer.

Mit Hilfe von lokalen Timern erhält man sehr präzise Ausführungszeiten der Tasks, wie dies in Abbildung 4.7 gezeigt ist. Ein Task bekommt a priori seine Ausführungszeit mit Hilfe eines Signals, das einen Zeitstempel als Parameter enthält, angekündigt. Der Prozess zieht sich darauf hin einen lokalen Timer auf und teilt diesen Zeitpunkt als Aufwachzeit dem zentralen Scheduler der SVM mit. Der Scheduler-Prozess `Scheduler2` wird dazu ebenfalls mit einer Periode `D_p` ausgeführt. Bei Ablaufen des Timers `S` wird dann ein Trigger-Signal `Trigger_T` an die entsprechende Task im Prozess `P2` versendet. Im Unterschied zu vorheriger Lösung wird dem Signal jedoch ein Zeitstempel, der um die Zeit `D_off` vom Triggerzeitpunkt in der Zukunft liegt, als Parameter mitgegeben. Der Zielprozess darf nun bei Signalempfang nicht sofort die eigentliche Task ausführen, sondern zieht sich einen lokalen Timer mit dem im Signal übergebenen Ablaufzeitpunkt auf. Durch das vorherige Übertragen des Triggersignals an den Zielprozess muß zum Zeitpunkt der Ausführung kein Signal erzeugt, kein Weg zum Zielprozess gesucht und auch keine erneute Auswahl des Prozesses durch den SVM-Scheduler erfolgen, wodurch diese Verzögerungen entfallen. Der Scheduler der SVM ist über lokale Timer informiert und kann die Ausführungswünsche der Prozesse geeignet ordnen, um damit bei Ablauf eines Timers sofort den richtigen Prozess zur Ausführung zu bringen.

4.2.1.2. Nachteile der SDL-Lösungen

Die in Abbildung 4.7 gezeigte Lösung läßt sich komplett mit vorhandenen SDL-Mitteln umsetzen und eliminiert die Verzögerung bei der Ausführung weitestgehend, hat jedoch auch Nachteile.

Jeder Task, der durch den Scheduler gesteuert werden soll, muß das Pattern, bestehend aus einem Signalempfang mit einem Zeitstempel sowie einem lokalen Timer, implementieren. Der Prozess, in dem der Task definiert ist, enthält zusätzliche Transitionen, die für die Verarbeitung der Triggersignale zuständig sind, und wird dadurch unübersichtlicher. Weiterhin läßt sich die Ausführung einer Task nicht mehr direkt als Folge eines Trigger-Signals ableiten, wodurch die Spezifikation schwerer verständlich wird. Der zusätzliche Zeitstempel als Parameter des Trigger-Signals stört zum einen die Logik, behindert aber vor allem die Wiederverwendung des Prozesses als SDL-Modul (bzw. Komponente) in anderen Projekten.

Betrachtet man den Ressourcen-Verbrauch der vorgestellten Lösungen, zeigen sich weitere Nachteile der reinen SDL-Lösungen: Im betrachteten System wird zunächst das Trigger-Signal erzeugt und versendet, eine Route vom Ausgangs- zum Zielprozess gesucht und schließlich das Signal an die Empfangswarteschlange des Zielprozesses angehängt. Der SDL-Scheduler muß zunächst den Zielprozess aus der Liste der ausführbaren Prozesses auswählen und aktivieren. Der Zielprozess prüft zunächst seine Empfangswarteschlange, führt nun jedoch nicht direkt den Task selbst aus, sondern startet einen lokalen Timer. Für die vorzeitige Aktivierung des Prozesses und den zusätzlichen Timer werden zusätzliche Ressourcen (Speicher und Zeit) verbraucht.[27] Die Ausführung erfolgt dann schließlich erst bei Ablauf des Timers, der eine erneute Aktivierung des Prozesses mit Kontextwechsel und Transitionsauswahl zur Folge hat. In Systemen mit beschränkten Ressourcen kann es durch die Zeitverzögerung zu einer Überlastsituation kommen.[28] Die Folge hiervon sind verspätete Taskausfüh-

---

[27] Sind weitere Signale in der Empfangswarteschlange des Prozesses, verzögert sich der Signalempfang für die verzögerte Ausführung weiter.

[28] Viele Prozesse sind gleichzeitig rechenbereit, weil deren Timer abgelaufen sind.

rung und schlimmsten Falls der Ausfall des Systems. Eine Umstellung des Designs ist nötig, um mehr Ressourcen zur Verfügung zu stellen. Alternativ bzw. zusätzlich kann ein Wechsel auf eine andere Hardware mit mehr Ressourcen und damit meist auch höherem Energiebedarf nötig werden. Da diese Situationen durch externe Ereignisse nicht immer vollständig vorhersehbar sind, kann bei geeignetem Design ein System auch durch das Verwerfen von Signalen wieder stabilisiert werden. Eine Lösung, die fast ohne zusätzliche Ressourcen auskommt und dadurch ein System wieder stabilisieren kann, wird in Kapitel 4.2.2.2 vorgestellt.

4.2.2. Echtzeitsignalisierung in SDL durch Erweiterung der Syntax und Semantik

Bisher wurde bereits gezeigt, daß alle Möglichkeiten, die SDL bietet, größere Nachteile haben, die zu Lasten der Ressourcen und der Ausfürungspräzision gehen. Wie sich zeigt kann durch geringere Anpassungen an der Sprache und der Semantik, die Echtzeitsignalisierung in SDL umgesetzt werden: Zunächst werden dazu die *Signalankunftzeit*, die *Signalablaufzeit* und einen Zeitstempel für jedes Signal durch Hinzufügen der Schlüsselwörter `at`, `expiry` und `sendtime` eingeführt. Die Änderungen der SDL-Syntax (Z.100 [55, 56]) sind in Listing 4.3 blau dargestellt. Die nötigen Anpassungen an der SDL-Semantik in Listing 4.4 fallen sehr gering aus, da einige Mechanismen, die zur Implementierung benötigt werden, bereits vorhanden sind. Als erstes wird dazu die Signalankunftzeit, die Signalablaufzeit und der *Signalzeitstempel* vorgestellt. Anschließend wird das vorherige Beispiel mit Hilfe der Änderungen sauber und effizient erneut modelliert.

```
<output statement> ::= output <output body> <end>
<output body> ::= <signal identifier> [<actual parameters>] ...
                  <communication constraints>
<communication constraints> ::=
                  {to <destination> | timer <timer identifier> | <via path>
                  | at <time expression> | expiry <time expression>}*
<imperative expression> ::= <sendtime expression> | ...
<sendtime expression> ::= sendtime
```

Listing 4.3: SDL-Syntax-Änderungen für `at`, `expiry` und `sendtime`

```
shared atArg: PlainSignalInst → Time
shared expiryArg: PlainSignalInst → Time
shared sendTime: PlainSignalInst → Time
controlled sendTime: SdlAgent → Time

maxTime(a:Time,b:Time):Time =def
```

```
  if $(b = undefined) \vee (a \geq b)$ then $a$ else $b$ endif

OUTPUT $=_{def}$ SIGNAL $\times$ VALUELABEL$^*$ $\times$ VALUELABEL $\times$ VIAARG $\times$ TIME $\times$ TIME $\times$ CONTINUELABEL

EVALOUTPUT($a$:OUTPUT) $\equiv$
  SIGNALOUTPUT($a.s-Signal$, $values(a.s-ValueLabel-seq, Self)$, $value(a.s-ValueLabel, Self)$, $a.$
        $s-ViaArg$, $a.s-Time$, $a.s2-Time$)
  $Self.currentLabel := a.s-ContinueLabel$

SIGNALOUTPUT($s$:SIGNAL, $vSeq$:VALUE$^*$, $toArg$:TOARG, $viaArg$:VIAARG, $atArg$:TIME, $expiryArg$:
      TIME) $\equiv$
  if $toArg \in$ PID $\wedge$ $s \notin toArg.s-Interface-definition$ then
    $RAISEEXCEPTION(InvalidReference, empty)$
  else
    choose $g$: $g \in Self.outgates \wedge Applicable(s, toArg, viaArg, g, undefined)$
      extend PLAINSIGNALINST with $si$
        $si.plainSignalType := s$
        $si.plainSignalValues := vSeq$
        $si.toArg := toArg$
        $si.viaArg := viaArg$
        $si.atArg := atArg$
        $si.expiryArg := expiryArg$
        if $atArg = undefined$ then $si.sendTime := now$
        else $si.sendTime := atArg$ endif
        $si.plainSignalSender := Self.self$
        $INSERT(si, now, g)$
      endextend
    endchoose
  endif

DELIVERSIGNALS $\equiv$
  choose $g$: $g \in Self.ingates \wedge g.queue \neq empty$
    let $si = g.queue.head$ in
      DELETE($si$,$g$)
      $si.arrival := maxTime(si.arrival, si.atArg)$
      if $si.toArg \in PID \wedge si.toArg \neq undefined$ then
        choose $sa$: $sa \in SDLAGENT \wedge sa.owner = Self \wedge sa.self = si.toArg$
          $INSERT(si, si.arrival, sa.inport)$
        endchoose
      else
        choose $sa$: $sa \in SDLAGENT \wedge sa.owner = Self$
          $INSERT(si, si.arrival, sa.inport)$
        endchoose
      endif
    endlet
  endchoose

SETTIMER($tm$:TIMER, $vSeq$ :VALUE$^*$, $t$:TIME) $\equiv$
  let $tmi = mk-TIMERINST(Self.self, tm, vSeq)$ in
    if $t = undefined$ then
      $Self.inport.schedule := insert(tmi, now + tm.duration, delete(tmi, Self.inport.schedule))$
        $si.arrival := now + tm.duration$
```

```
      else
        Self.inport.schedule := insert(tmi, t, delete(tmi, Self.inport.schedule))
        si.arrival := t
    endif
    si.sendTime := si.arrival
  endlet

SelectTransitionStartPhase ≡
  if Self.currentExceptionInst ≠ undefined then
    Self.agentMode3 := selectException
    Self.agentMode4 := startPhase
  elseif Self.currentStartNodes ≠ ∅ then
    Self.stateNodeChecked := undefined
    Self.agentMode3 := selectStartTransition
  elseif Self.currentExitStateNodes ≠ ∅ then
    Self.stateNodeChecked := undefined
    Self.agentMode3 := selectExitTransition
  elseif Self.currentConnector ≠ undefined then
    Self.agentMode3 := selectFreeAction−
  else
    Self.inport.schedule := cleanSchedule(Self.inport.schedule)
    Self.inputPortChecked := Self.inport.queue
    Self.agentMode3 := selectPriorityInput
    Self.agentMode4 := startPhase−
  endif

cleanSchedule(siSeq:SignalInst*):SignalInst* =def
  if siSeq = empty then empty
  elseif siSeq.head.expiryArg = undefined ∨ siSeq.head.expiryArg ≥ now then
    < siSeq.head > ∩cleanSchedule(siSeq.tail)
  else cleanSchedule(siSeq.tail)
  endif
```

Listing 4.4: Änderungen der SDL-Semantik für `at`, `expiry` und `sendtime`

4.2.2.1. Signalankunftzeit mit `at`

Bei der Echtzeitsignalisierung soll ein Signal zu einem in der Zukunft liegenden Zeitpunkt bei einem Zielprozess empfangen werden. Bisher ist es in SDL nur möglich Signale zu versenden – deren Ankunftszeit bleibt dabei jedoch unbekannt. Timer erfüllen die Eigenschaft der Ankunft zu einem bestimmten Zeitpunkt, können aber nur lokal in einem Prozess gesetzt und empfangen werden. Im Unterschied zu Signalen können Timer über das Schlüsselwort `reset` gelöscht werden, Signale jedoch nicht. Ein Echtzeitsignal erhält alle Eigenschaften eines Signals und zusätzlich die Eigenschaft des Timers, erst in der Zukunft zugestellt zu werden. Das Löschen eines Echtzeitsignals ist weiterhin nicht möglich, da dies eine essentielle Eigenschaft eines

SDL-Signals darstellt. Jedes SDL-Signal enthält einen Zeitstempel, der die Ankunftszeit des Signals bei dem Zielprozess angibt. Die Eingangswarteschlange des Zielprozesses sortiert die Signale nach diesen Zeitstempeln. Ein Echtzeitsignal bekommt bei Ankunft an den Zielprozess nicht die aktuelle Systemzeit als Zeitstempel, sondern die im Signal definierte Zeit gesetzt. Der Prozess verarbeitet jedoch nur Signale aus der Eingangswarteschlange deren Ankunftszeit bereits abgelaufen ist. Es ist somit möglich Signale, Echtzeitsignale und Timer über die gleiche Eingangswarteschlange zu verarbeiten.

Soll ein Echtzeitsignal versendet werden, muß beim Versenden des Signals lediglich das Schlüsselwort `at` und eine Zeit (die Zustellungszeit) angegeben werden (siehe Listing 4.3), wie z.B. **`output signal at now`** `+ 1.0`. Zusätzlich zu den Änderungen an der Syntax waren einige kleine Anpassungen an der formalen SDL-Semantik, wie in Listing 4.4 gezeigt, erforderlich. Die OUTPUT-Anweisung wurde dazu um einen Zeitparameter TIME (Zeile 9) sowie das ASM-Makro SIGNALOUTPUT um den Parameter *atArg* (Zeile 15) erweitert. Sobald ein Echtzeitsignal erstellt ist (Zeile 20), wird die angegebene Signalankunftszeit über die Funktion *atArg* (Zeile 1) mit dem Signal assoziiert. Die Weiterleitung eines Echtzeitsignals bleibt gegenüber einem normalen SDL-Signal unverändert, eine Unterscheidung erfolgt erst bei der Ankunft am Zielprozess. Das ASM-Makro DELIVERSIGNALS (Zeile 35), das für die Zustellung des Signals zum Zielprozess zuständig ist, ermittelt mit der Hilfsfunktion *maxTime* das Maximum aus der angegebenen Signalankunftszeit (*si.atArg*) und tatsächlichen Ankunftszeit (*si.arrival*). Wird das Signal auf einem verzögernden Kanal transportiert, wird so dessen Verzögerung korrekt berücksichtigt und gleichzeitig kann ein Echtzeitsignal nicht die Reihenfolge der konsumierbaren Signale in der Eingangswarteschlange verändern. Abhängig von der Implementierung der SDL-SVM, kann so die Signalweiterleitung entweder periodisch (z. B. über einen Round-Robin-Scheduler) oder zu Zeiten geringer Systemlast erfolgen.

4.2.2.2. Signalvergänglichkeit

Mit der Echtzeitsignalisierung ist es bereits möglich, Signale soweit zu verzögern, daß diese erst zu einem bestimmten Zeitpunkt in der Zukunft zugestellt werden. Die Gesamtlast des Systems sinkt hierbei zwar nicht, jedoch sollte bei entsprechendem Systementwurf die Zahl der Lastspitzen geringer ausfallen. Durch Benutzerinteraktionen oder externe Ereignisse kann selbst ein solches System zur Überlast gebracht werden: die Signalwarteschlangen füllen sich schneller als diese abgearbeitet werden können. In den Warteschlangen sammeln sich Signale, deren Information ungültig oder uninteressant geworden ist, was besonders bei periodisch auftretenden Signalen der Fall sein kann. Da jedes der Signale von der SDL-SVM und dem Prozess erst verarbeitet werden muß, wird das bereits ausgelastete System mit diesen Signalen belastet. Sobald man in einem Signal mittels zusätzlichem Parameter ein maximales Alter spezifizieren kann, würde das Signal nach dessen Überschreiten aus der Signalwarteschlange entfernt und hierdurch das System nicht zusätzlich belastet. In Kombination mit der Echtzeitsignalisierung entsteht ein fest definiertes Zeitfenster, in dem eine Information gültig ist und ausgeliefert werden soll. Bei der Erstellung von SDL-Systemen unter Zuhilfenahme von Signalvergänglichkeit muß beachtet werden, daß durch das Ausbleiben eines Signals die zugehörige Transition nicht mehr feuert. Wird dies beim Entwurf nicht berücksichtigt, kann es zu einem *Dead-Lock* kommen.

Analog zur Echtzeitsignalisierung, wird der Signalversand um einen weiteren Parameter **`expiry`** ergänzt, sodaß man das maximale Alter des Signals über eine Zeit wie folgt angeben kann: **`output signal expiry now`** `+ 2.0`. Für die Signalvergänglichkeit muß dazu die SDL-Semantik angepaßt werden: In der Subdomäne OUTPUT wird der zusätzliche Zeitwert hinzugefügt (Zeile 9), das ASM-Makro SIGNALOUTPUT wird um den Parameter *expiryArg* ergänzt (Zeile 15) sowie der Wert des Arguments `expiry` bei der Signalerzeugung (Zeile 20) und der Signalinstanz (Zeile 26) mit Hilfe der Funktion *expiryArg* (Zeile 2) hinzugefügt. Während des Signaltransports bleibt die Eigenschaft `expiry` unberücksichtigt und wird erst beim Zielprozess zum Zeitpunkt der Transitionsauswahl mit dem ASM-Makro SELECTTRANSITIONSTARTPHASE (Zeile 64) überprüft. Da während der Transitionsauswahl die Eingangswarteschlange unveränderlich ist (Zeile 78), werden direkt vor diesem Zeitpunkt alle bereits abgelaufenen Signale aus der Warteschlange entfernt (Zeile 77). Formal geschieht diese Säuberung der Warteschlange mit Hilfe der Funktion *cleanSchedule* (Zeile 83).

4.2.2.3. Signalzeitstempel

In SDL-Systemen ist es häufig wichtig zu wissen, wann ein Signal erstellt wurde, da dies unmittelbar den Zeitpunkt eines Ereignisses widerspiegelt. Externe SDL-Signale aus der Umgebung dienen häufig der Synchronisierung und benötigen einen sehr präzisen Zeitstempel. Um standardkonform zu bleiben, kann dieser Zeitstempel nur als Parameter mitgegeben werden, wie dies auch in der Arbeit von P. Becker et. al. [10] vorgeschlagen wurde. Bereits Álvarez et al. [1] schlagen vor, jedem Signal einen Zeitstempel zuzuordnen. Diese Idee läßt sich ebenfalls für die Echtzeitsignalisierung anwenden und nicht nur auf Simulationen anwenden. Die anonyme SDL-Variable `sendtime` (siehe Listing 4.3) gibt dabei den Sendezeitpunkt des zuletzt konsumierten SDL-Signals wieder. Bei normalen Signalen gibt die Variable damit den Zeitpunkt der `Output`-Anweisung wieder. Lokale Timer erstellen ihr Signal zum Zeitpunkt ihres Ablaufes, was ebenfalls ihrem Erstellungszeitpunkt entspricht. Echtzeitsignale hingegen werden vorzeitig erstellt und an den Zielprozess geleitet. Sie sollen ähnlich behandelt werden wie von außen aufgezogene Timer. Die Variable `sendtime` soll also in diesem Fall genau wie bei einem Timer den Zeitpunkt ihrer Ankunftszeit wiedergeben.

In der SDL-Semantik wird dies wie folgt umgesetzt: Bei normalen Signalen wird zum Zeitpunkt der Signalerstellung die aktuelle Zeit in die Variable `sendtime` (Zeile 27) geschrieben. Im Falle eines Timers wird der Zeitpunkt, zu dem dieser auslösen soll, in die Variable geschrieben (Zeile 61). Bei Echtzeitsignalen schließlich wird ihre angegebene Ankunftszeit *si.atArg* (Zeile 28) in diese Variable geschrieben.

Durch Auswerten des Ausdrucks `now - sendtime` kann nun bei normalen Signalen die Zeit für den Signaltransfer und die zusätzliche Wartezeit in der Signalwarteschlange ermittelt werden. Bei Timern und Echtzeitsignalen entfällt die Zeit für den Transfer und es wird lediglich die Wartezeit in der Signalwarteschlange ermittelt.[29]

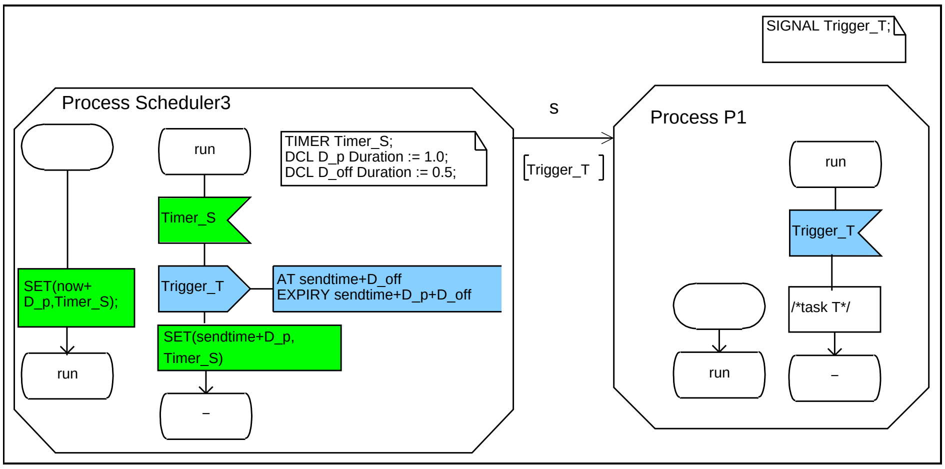


Abbildung 4.8.: *Echtzeitsignalisierung*

---

[29] Aus der globalen durchschnittlichen Wartezeit in der Signalwarteschlange läßt sich aktuelle Systemlast ermitteln. Lokal kann sie ein Indikator für fehlerhaft dimensionierte Zeiten in der Spezifikation anzeigen.

4.2.2.4. Beispiel für Echtzeitsignalisierung

Das Beispiel aus Abbildung 4.6 soll nun zeigen, wie sich Echtzeitsignalisierung mit den vorgestellten Mitteln umsetzen läßt. Abbildung 4.8 zeigt die fertige Lösung, die ohne Anpassungen an dem Prozess `P1` mit der Task auskommt und deshalb identisch zu dem Prozess aus Abbildung 4.6 ist. Genau wie es die Abstraktion fordert, wurde lediglich der Scheduler (und die SVM) auf die Echtzeiteigenschaften der Signale angepaßt. Der Scheduler (`scheduler2`), wird dabei, wie in Abbildung 4.7, periodisch vor der eigentlichen Taskausführung ausgeführt. Bei Aktivierung versendet dieser Echtzeitsignale an alle Tasks mit einem Aktivierungs- und einem Ablaufzeitpunkt in der Zukunft. Sind ausreichend Ressourcen verfügbar und die Ausführungszeit der Tasks bekannt (abschätzbar oder meßbar), kann man die Zeitpunkte so wählen, daß sich nie zwei Taskaktivierungen gleichzeitig im Zielprozess befinden. Die Weiterleitung der Signale vom Scheduler erfolgt zu einem Zeitpunkt mit geringer oder keiner Systemlast, wodurch eine präzise Ausführung der Task gewährleistet ist. Die Verwendung der Echtzeitsignalisierung stellt weiterhin sicher, daß veraltete Signale verworfen werden. Im Prozess `P1` ist keine Anpassung vorgenommen worden, was dessen Logik einfach hält und weniger Ressourcen benötigt. Insgesamt spart die geringere Zahl an Transitionen, Timern und Variablen zur Laufzeit Ressourcen (RAM, CPU) ein.

Eine weitere Verbesserung, die ebenfalls zur Übersichtlichkeit beiträgt, ist die Verwendung von `sendtime` sowohl zum periodischen Aufruf des Timers `Timer_S`, aber auch zur Definition der Echtzeitsignaleigenschaften (`AT` und `EXPIRY`). In den vorherigen Lösungen war dazu eine eigene Variable `ref` nötig, obwohl diese Information in der SDL-SVM bereits vorhanden, mit SDL-Mitteln jedoch nicht zugreifbar war. Die Verwendung von Signalen ohne spezielle Scheduler-Parameter macht diese einfacher verständlich und erhöht die Möglichkeit zu Wiederverwendbarkeit der Prozesse in Modulen.

4.2.3. Anpassungen an der Umgebung (SEnF)

Das SDL Environment Framework bildet die Schnittstelle zwischen dem SDL-System, dem Betriebssystem oder direkt mit der Hardware. Um auch beim Versand von Signalen in die Umgebung eine höhere Genauigkeit über Echtzeitsignale zu erreichen, müssen auch hier kleine Anpassungen vorgenommen werden.

Signalversand Zum Austausch von Signalen, die aus der Umgebung kommen oder an die Umgebung gesendet werden, wird SEnF, wie jeder andere SDL-Agent auch, regelmäßig ausgeführt. Signale an die Umgebung werden ebenfalls, wie bei jedem SDL-Agenten, in eine Eingangswarteschlange gestellt, der Reihe nach abgearbeitet und an die Hardwaretreiber weitergegeben. Für die Unterstützung der Echtzeitsignalisierung wurde dieses Verfahren so erweitert, daß angeschlossene Hardware sehr zeitgenau gesteuert werden kann. Im Unterschied zu normalen Prozessen werden hier alle Signale, also auch Echtzeitsignale mit einem Zeitstempel in der Zukunft sofort verarbeitet. Jedes Signal wird dazu zuerst an eine Offset-Korrektur-Funktion des Treibers weitergeleitet. Aufgabe dieser Funktion ist es, in Abhängigkeit des jeweiligen Treibers und den übergebenen Daten zu errechnen, wie früh dieses Signal verarbeitet werden muß, damit die übergebenen Daten zum geplanten Zeitpunkt der Hardware zur Verfügung stehen. Ziel hierbei ist es, innerhalb von SDL keine hardware- oder treiberspezifischen Spezifikationen zu verwenden und stattdessen die Berechnung in den Treiber zu verlagern. An einem Beispiel wird dieses Vorgehen schnell klar: Über den Transceiver des Imote2 sollen 100 Bytes übertragen werden. Die Offset-Korrektur errechnet dazu die benötigte Zeit zur Datenübertragung der CPU zum Transceiver und zum Befüllen der Puffer des Transceivers und addiert die Umschaltzeit des Transceivers vom Empfangs- zum Sendebetrieb sowie eventuelle weitere treiberspezifische Verzögerungen. Zur Ermittlung der Zeiten können Messungen, Abschätzungen und die Datenblätter herangezogen werden. Die ermittelte Zeit wird vom Zeitstempel des Signals abgezogen und das Signal in eine zentrale Warteschlange der Umgebung einsortiert. Um die Genauigkeit der Signalauslösung weiter zu erhöhen, wird ein Hardware-Timer (oder ein möglichst genauer Betriebssystem-Timer) für die Steuerung der zentralen Warteschlange verwendet. Löst der Timer aus, wird das erste Signal aus der Warteschlange entfernt, die Signalgültigkeit überprüft und das Signal bzw. die im Signal enthaltenen Daten an den Treiber weitergeleitet.

Signalempfang Viele Hardware-Komponenten signalisieren über einen Hardware-Interrupt, sobald ein Ereignis vorliegt, was zur sofortigen Ausführung der sog. *Interrupt Service Routine* (ISR) führt. Während der Ausführung der ISR ist der normale Programmfluß angehalten, deshalb sollte diese Routinen sehr klein gehalten werden. Es wird meist lediglich ein Marker gesetzt, der bei Ausführung des SEnF die eigentlichen Daten aus der Hardwarekomponente ausliest. Zwischen dem Auftreten des Hardware-Ereignisses, dem Transfer der Daten und schließlich der SDL-Signalerzeugung kann einige Zeit vergehen. In der ISR wird deshalb zusätzlich zu dem Marker noch der Zeitstempel ermittelt und dem erzeugten SDL-Signal als Erstellungszeitpunkt hinzugefügt. Innerhalb von SDL läßt sich so der exakte Zeitpunkt des Hardware-Ereignisses ermitteln.

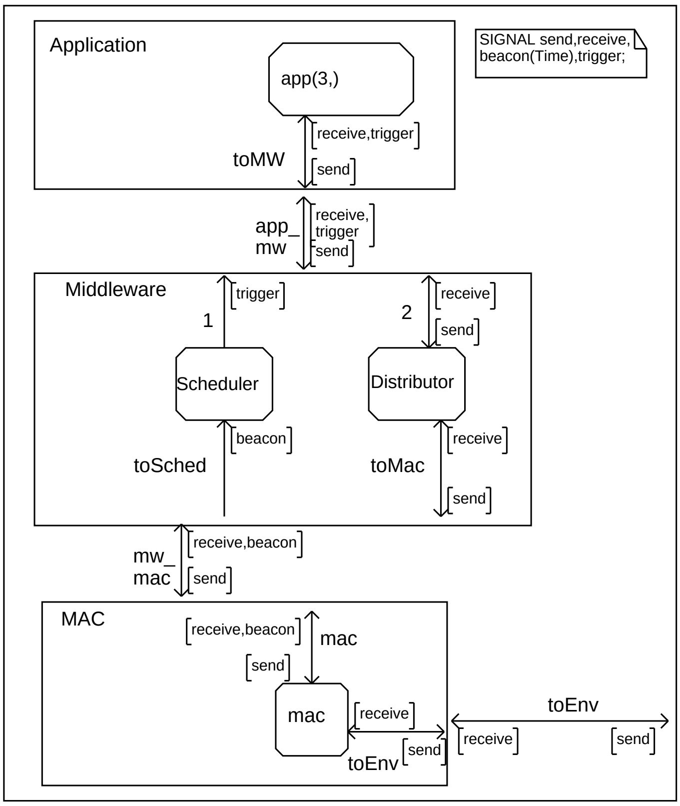


Abbildung 4.9.: Vereinfachte Sicht auf die NCS-CoM

4.2.4. Evaluation

Da die *Network Control Systems-Communication Middleware* (NCS-CoM), wie sie in Kapitel 5.2.2 vorgestellt wird, zu komplex ist, wurden die Echtzeitsignale an einem vereinfachten Modell, dessen Kommunikationseigenschaften ähnlich sind, evaluiert. Abbildung 4.9 zeigt den Aufbau der NCS-CoM und entspricht dem Evaluationssystem. Auf unterster Ebene befindet sich die MAC-Schicht, die direkt mit den Treibern des Transceivers im SEnF kommuniziert. Die `MAC`-Schicht ist sowohl für den Zugriff auf das Kommunikationsmedium, aber auch für die Tick-Synchronisation der Knoten zuständig. Im Master-Knoten wird dazu periodisch das Synchronisationssignal in der MAC-Schicht erzeugt, über den Transceiver verschickt und von allen anderen Knoten empfangen. Ankommende Daten und das Synchronisationssignal werden an die darüber liegende `Middleware`-Schicht weitergeleitet, die ihrerseits zu versendende Daten an die MAC-Schicht leitet. Der `Distributor` innerhalb der Middleware leitet ankommende Daten an die zugehörige Applikation (`app`) im Block `Application` weiter. Das Synchronisationssignal wird vom `Scheduler` der Middleware verarbeitet, der auf dieser Grundlage die Applikation steuert. Nach Aktivierung einer Applikation durch den Scheduler versendet diese Daten, die durch die Middleware an die MAC-Schicht weitergeleitet werden.

Die gezeigte Architektur entspricht hierbei einem klassischen Modellierungsansatz eines Kommunikationssystems.

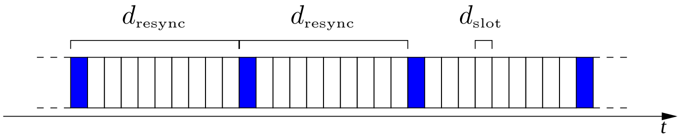
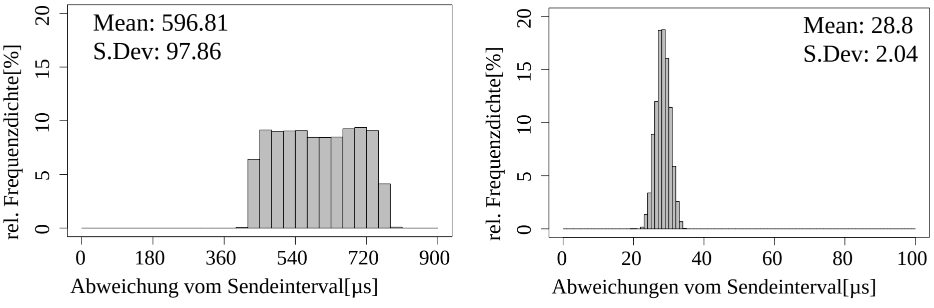


Abbildung 4.10.: MAC-Schicht mit TDMA-Zugriffsverfahren

Die Aufteilung des Kommunikationsmediums unter Verwendung eine TDMA-Zugriffsverfahrens ist in Abbildung 4.10 gezeigt. In regelmäßigen Abständen (d_{resync}) sendet ein Knoten ein Synchronisationssignal. Die Zeit dazwischen wird in gleich lange Slots (d_{slot}) aufgeteilt. Die Kommunikation innerhalb eines Slots ist exklusiv für einen Knoten reserviert, der dann zu diesem Zeitpunkt senden darf. Weicht die Synchronisation der beteiligten Knoten zu stark voneinander ab, sodaß ein Knoten zu spät mit seiner Übertragung beginnt, reicht diese noch in den nächsten Slot hinein, was zur Zerstörung der laufenden und der nachfolgenden Übertragung führen kann. Durch die Verwendung der Signalzeitstempel, die im SEnF sehr früh geschieht, wird der Zeitpunkt des Synchronisationssignals sehr exakt ermittelt. Sobald der Transceiver den vollständigen Empfang eines Datenrahmens per Hardwareinterrupt dem Mikrocontroller mitteilt, wird die aktuelle Systemzeit zu dem Signalempfang gespeichert. Ist das Signal vollständig empfangen, wird aus der Größe des Datenpakets und den ermittelten Zeiten aus dem Datenblatt der Sendebeginn zurückgerechnet.

Die Bestimmung der exakten Ankunftszeit ist eine Grundvoraussetzung zur exakten Synchronisation, jedoch wird vom Master-Knoten ebenfalls ein exakter Versand des Synchronisationssignals benötigt. Auf einem Imote2-Knoten wurden 6000 Meßwerte mit einem Resynchronisationsintervall von $d_{\text{resync}} = 100\,\text{ms}$ gesammelt. Betrachtet man nun die Verzögerung beim Versand des direkt in der MAC-Schicht erzeugten Sync-Signals, mittels normalem Output-Signal (Abbildung 4.11(a)) und mittels Echtzeitsignalisierung (Abbildung 4.11(b)), zeigen sich deutliche Abweichungen.



(a) normaler Signalversand

(b) *Echtzeitsignalisierung*

Abbildung 4.11.: Tick-Synchronisierung mittels normalem Output und Echtzeitsignalisierung über **at** im SEnF

Der normale Signalversand aus Abbildung 4.11(a) zeigt dabei Abweichungen zwischen $\sim 400\,\mu$s und $\sim 800\,\mu$s. Im Vergleich dazu zeigt die Echtzeitsignalisierung aus Abbildung 4.11(b), bei der das Signal bereits vorher schon an den Zielprozess gesendet wurde, lediglich ein Maximum von $35\,\mu$s und ein Minimum von $25\,\mu$s. Die Genauigkeit des Signals könnte so, ohne eine Offset-Korrektur innerhalb von SEnF, bereits um den Faktor 22 gesteigert werden. Stellt man diese Korrektur über Messungen sehr genau ein, sind lediglich Schwankungen zwischen $5\,\mu$s und $15\,\mu$s zu erwarten. Bei der normalen Signalisierung kann durch Optimierungen der SVM die minimale Verzögerung von $400\,\mu$s zwar optimiert werden, für die Genauigkeit ist die maximale Verzögerung entscheidend, die im Unterschied zur Echtzeitsignalisierung von der aktuellen Systemlast abhängt.

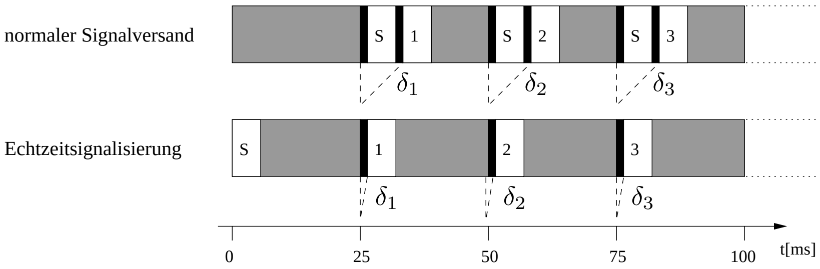
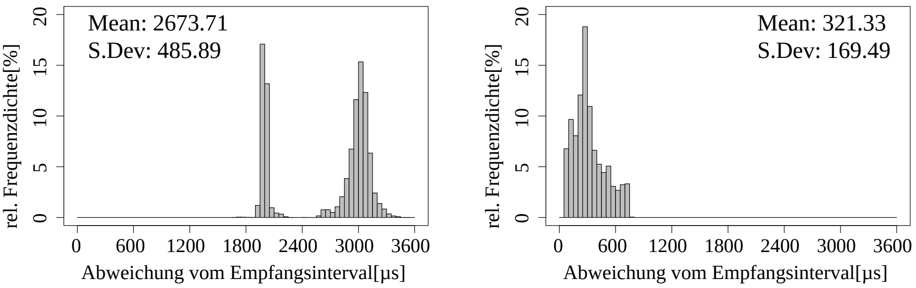


Abbildung 4.12.: Schedule der Anwendung

Zur Aktivierung der Anwendung kann, wie durch die Beispiele in diesem Kapitel beschrieben, ein zentraler Scheduler `Scheduler` eingesetzt werden, der dann über Echtzeitsignale die Anwendung (`app`) aktiviert (Vergleiche Abbildung 4.9). Für die Evaluation aktiviert der Scheduler dazu die Anwendungen `app(1)`, `app(2)` und `app(3)` regelmäßig alle 100 ms. Abbildung 4.12 zeigt, wie der Ablaufplan dabei unter Verwendung normaler SDL-Signale und mit Hilfe von Echtzeitsignalisierung aussieht. Bei normaler Signalisierung wird zunächst der Scheduler-Process aktiviert, der nun erst das Aktivierungssignal an die jeweilige Applikation sendet. Die Verzögerungen $\delta_1, \delta_2, \delta_3$ geben jeweils die Differenz vom geplanten Zeitpunkt bis zur Aktivierung der jeweiligen Applikation an und beinhalten dabei die Ausführungszeit des Schedulers selbst. Im Vergleich dazu wird im Falle der Echtzeitsignalisierung der Scheduler nur einmal ausgeführt, die Aktivierung der Applikation erfolgt ausschließlich nur noch durch den Scheduler der SVM.



(a) normaler output (siehe Abbildung 4.6) (b) *Echtzeitsignalisierung* (siehe Abbildung 4.8)

Abbildung 4.13.: Scheduler mit normalem Output und Output `at`

Für den Schedule aus Abbildung 4.12 wurden auf dem Imote2 ebenfalls 6000 Einzelmessungen durchgeführt. Die Ergebnisse sind sowohl für die normale Signalisierung Abbildung 4.13(a) aber auch für die Echtzeitsignalisierung in Abbildung 4.13(b) gezeigt. Bei normaler Signalerzeugung dauert es zwischen 1600 μs und 3600 μs, bis die jeweilige Applikation aktiviert ist. Im Falle der Echtzeitsignalisierung geht die Zeit auf 0 μs bis maximal 800 μs zurück. Um den Vergleich zwischen beiden Ansätzen einigermaßen fair zu gestalten, verwendet der Scheduler der SVM ein Round-Robin Verfahren und aktiviert jeden Prozess, ohne vorher zu überprüfen, ob dieser konsumierbare Signale in der Eingangswarteschlange enthält. Wird diese Optimierung aktiviert, enthält der Scheduler der SVM nur die ausführungsbereiten Tasks, d.h. der SDL-Prozess wird sofort ausgeführt, sobald das Echtzeitsignal in dem Prozess ankommen soll.

4.2.5. Fazit

Mit dem Thema Echtzeit und der Umsetzung in SDL haben sich bereits andere Autoren beschäftigt. Die meisten Arbeiten in diesem Bereich beschäftigen sich jedoch ausschließlich mit der Simulation von SDL-Systemen, wobei die Simulation nicht in Echtzeit ablaufen muß. So wird beispielsweise in den Arbeiten von Bozga et. al. [16, 17] zwar die simulierte Zeit beeinflußt, dieser Ansatz ist jedoch nicht zur Ausführung auf echter Hardware geeignet. Im Unterschied zur Ausführung auf echter, eventuell sogar Ressourcen-limitierter Hardware, spielen in reinen Simulationsansätzen sowohl Speicher als auch Laufzeit keine große Rolle. Praktische Ansätze zur Realisierung von Echtzeitsystemen werden in den Arbeiten von Bræk, Mitschele-Thiel und Sanders [18, 84, 93] gegeben.

In den Arbeiten von Mitschele-Thiel [84] und Gotzhein et. al. [46] wird versucht die Systemlast durch Begrenzung der Anzahl der Signale in der Eingangswarteschlange gering zu halten. Im Unterschied zum vorgestellten Ansatz ist nur die Anzahl selbst und nicht die Signalverfallszeit, das Kriterium, ob ein Signal verworfen wird. Durch die Limitierung der Signalzahl läßt sich der Speicherverbrauch eines Systems einschränken und auch die Systemlast durch das Verwerfen von Signalen reduzieren. Sollen jedoch Duplikate von Signalen verworfen werden, weil deren Information veraltet ist, muß die Signalwarteschlange vor dem Einfügen aufwändig erst nach bereits vorhandenen gleichen Signalen durchsucht werden und diese aus der Eingangswarteschlange des Prozesses entfernt werden.

Álvarez et al. [1] schlägt ebenfalls vor, SDL-Signalen Zeitstempel mitzugeben und diesen für das Scheduling zu verwenden. Im Unterschied zum vorgestellten Ansatz wird die Warteschlange an Hand dieser Zeiten für das Scheduling umsortiert. Die von SDL geforderte FIFO-Eigenschaft der Signalwarteschlange gilt hier nicht mehr. Signale die früher erzeugt wurden und im SDL-System durch Weiterleitung von SDL-Agenten mehr Zeit benötigt haben, als ein anderes Signal werden jedoch vor diesem in der Warteschlange einsortiert.

Die in dieser Arbeit ermittelten Ergebnisse zeigen, daß die Echtzeitsignalisierung, wo sie sinnvoll angewandt werden kann, deutlich weniger Ressourcen benötigt, präziser die erwarteten Zeitpunkte trifft und durch die geringere Ausführungszeit weniger Energie verbraucht. Die Einführung von Zeitstempeln für jedes Signal eröffnet neue Möglichkeiten bei der Modellierung von SDL-Systemen. Im Unterschied zuÁlvarez et al., verhält sich der hier vorgestellten Ansatz Konform zur SDL-Semantik, da Signale über den `AT`-Zeitstempel nur eine definierte Verzögerung erhalten, nicht aber andere Signale im System überholen können. Die Erweiterung der Semantik um die Signalvergänglichkeit hilft, leistungsschwache Systeme zu stabilisieren und verhindert zusätzlich auch den Empfang von Signalen, deren Information längst nicht mehr gültig ist.

Die Echtzeitsignalisierung und Signalzeitstempel sind essentielle Bestandteile für jedes Kommunikationssystem und können mit Standardmitteln in SDL nicht umgesetzt werden. Kombiniert man zusätzlich die Energiesignalisierung, wie sie in Kapitel 3.3 vorgestellt wurde, mit den Echtzeitsignalen und der Signalvergänglichkeit, ist es möglich, sehr präzise den Energieverbrauch zu steuern. Gerade für die Energiesignalisierung ist die Signalvergänglichkeit wichtig, da ein zu spät verarbeitetes Signal zu einem Fehlverhalten führen würde.

5

Kapitel 5.

Kommunikationsplattform für Regelungssysteme

Regelungssysteme bestehen aus Sensoren zur Meßwerterfassung, einer Regelstrecke, einem Regler und Aktuatoren. Aktuatoren beeinflussen die meßbaren Größen der Regelungsstrecke, an der die Sensoren angebracht sind, welche Meßwerte an den Regler liefern (vergleiche Abbildung 5.1). Die Sensoren und Aktuatoren können direkt an den Regler angeschlossen sein, wodurch die Verzögerungen in der Kommunikation mit Sensoren und Aktuatoren sich auf die elektrische Signalausbreitung beschränken und somit sehr gering sind. Besitzt eine Regelstrecke viele Sensoren und Aktuatoren oder ist die Entfernung zum Regler sehr groß, ist die direkte Verbindung häufig nicht praktikabel. Alternativ kann ein Regelungssystem über ein drahtgebundenes oder drahtloses Kommunikationssystem mit seinen Komponenten verbunden sein. Teilen sich jedoch mehrere Regelungen ein und das selbe Kommunikationssystem, steht es somit einer Regelung nicht exklusiv zur Verfügung und es kann durch konkurrierenden Zugriff zu Verzögerungen und Datenverlusten kommen (siehe Kapitel 2.4.2).

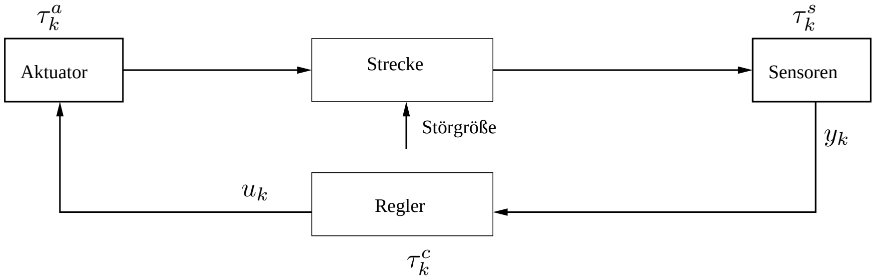


Abbildung 5.1.: Regelungssystem

In vielen Fällen können die Regelungsaufgaben durch sehr kleine und energiesparsame Mikrocontroller realisiert werden. Je nach Komplexität des Regelungsalgorithmus kann dieser direkt implementiert werden oder die Werte müssen à priori vorberechnet und in Nachschlagetabellen abgelegt werden, was jedoch auf Grund der Größe des Zustandsraums nicht für jeden Algorithmus möglich ist. Mit steigender Komplexität der Berechnung und der damit verbundenen erhöhten Anforderungen an die Leistungsfähigkeit der CPU oder der Größe des Speichers steigt im Allgemeinen der Energieverbrauch. Bei batteriebetriebenen Geräten muß somit großer Wert auf die Auswahl der Hardware gelegt und bei der Programmierung besonders auf den Ressourcenverbrauch geachtet werden.

Durch die Verwendung von Funkkommunikation zur Vernetzung der Sensoren oder Aktuatoren mit dem Regler können Komponenten ohne vorheriges Legen von Kabeln miteinander verbunden werden. Dies ist gerade an den Stellen von Interesse, an denen Kabel schwer oder gar nicht zu verlegen sind. Andererseits haben Funknetzwerke meist eine geringere Bandbreite als kabelgebundene Kommunikationsnetze, sind vielen Störungen ausgesetzt und stehen selten exklusiv zur Verfügung. Anders als kabelgebundene Netzwerke besitzt ein Funknetzwerk in der Regel eine *Rundruf-Eigenschaft* (Broadcast), d.h. alle Knoten in Reichweite des Senders können die Daten empfangen. Verwenden zwei miteinander kommunizierende Knoten die gleiche Trägerfrequenz, so kommunizieren sie unidirektional, da das Senden eines Signals den eignen Empfang von Signalen entfernter Sender stört.[30] Für die Entwicklung von netzwerkbasierten Regelungssystemen müssen daher sowohl alle Aspekte des Netzwerkes als auch alle Anforderungen der Regelung berücksichtigt werden.

Cross-Layer-Design Der häufigste Ansatz in der Regelungstechnik für den Entwurf einer netzwerkbasierten Regelung besteht darin, ein Standard-Netzwerkprotokoll und dessen Hardware (beispielsweise WLAN oder IEEE 802.15.4) [27] zu verwenden und den Regler unabhängig von dessen spezifischen Eigenschaften zu entwickeln. Die Regelungs-Applikation setzt dabei meist direkt auf einem Kommunikationsprotkoll (beispielsweise TCP/IP) oder einer Basistechnologie (IEEE 802.15.4) auf und verzichtet auf eine Abstraktion durch eine Middleware, weshalb bei Austausch eines der Netzwerkknoten oft das ganze Netz neu konfiguriert werden muß. Aus Sicht der Regelungstechnik wird ein Netzwerk meist stark abstrahiert als Transportmedium mit einer zeitlichen Verzögerung und sporadischen Rahmenausfällen einer definierten Häufigkeit betrachtet. Das Ergebnis dieses Regelungsentwurfs wird entweder simulativ unter der Annahme von deterministisch gleichverteilten Rahmenausfällen oder einfach nur einer fixen Netzwerkverzögerung bestimmt [74]. In Folge dessen werden viele Netzwerkregelungen weiter so entworfen, daß diese eine konstante Abtastzeit erwarten, was jedoch zu Problemen führt, wenn das Netzwerk dies nicht unterstützt. Drahtlose Regelungsnetzwerke zeigen jedoch eine variable Gesamtverzö-

---

[30] Bidirektionale Verbindungen sind nur möglich, wenn die beiden Kommunikationsrichtungen auf getrennten Trägerfrequenzen und einem eigenen Transceiver erfolgt.

gerung zwischen aufeinanderfolgenden Datenübertragungen durch Rahmenverluste und Neuübertragungen oder durch Zeitabweichungen der Uhren zwischen den Netzwerkknoten. Die Optimierung des Energieverbrauchs erfolgt oft nur statisch über die Auswahl der verwendeten Hardware, wird aber beim Protokoll- und Software-Design meist nicht weiter berücksichtigt.

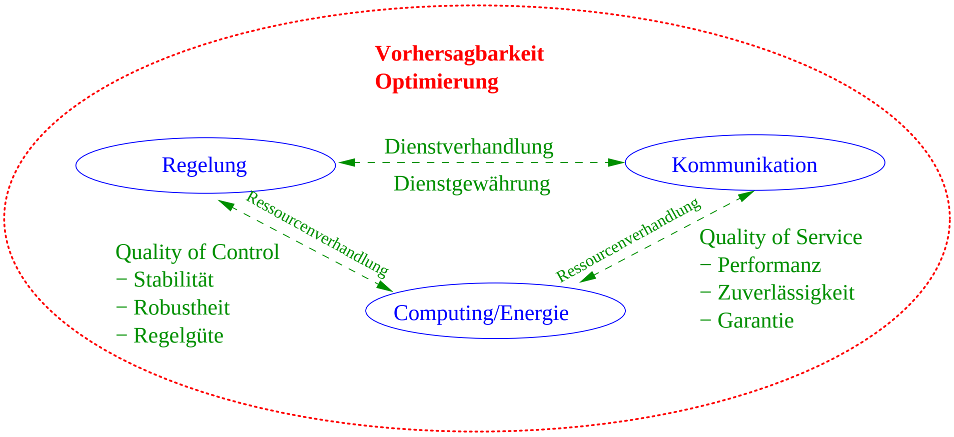


Abbildung 5.2.: C³-Cross-Design

Ziel des *Cross-Design* ist es hingegen, alle Komponenten auf das Regelungssystem angemessen abzustimmen (siehe Abbildung 5.2). Beim Cross-Design wird sowohl die Regelung (*Quality of Control*) als auch die Kommunikation optimal (*Quality of Service*) auf die Regelungsstrecke und die verwendete Hardware angepaßt, es erfolgt so ein ganzheitlicher Entwurf. Die Wechselwirkung zwischen der Regelung, der Middleware und dem Netzwerk kann dabei soweit verfeinert werden, daß z. B. die Regelung ihren Algorithmus dynamisch an die Qualität des Netzwerks, die CPU-Auslastung oder die Restkapazität der Batterie anpaßt. Zunächst werden nun alle Komponenten einer Netzwerk-Regelung einzeln und danach das inverse Pendel als Anwendung betrachtet.

Netzwerk Grundsätzlich unterscheidet man bei Netzwerkprotokollen zwischen wettbewerbsbasierten und Multiplex-Verfahren.

Sollen Daten über ein wettbewerbsbasiertes Medienzugriffsverfahren übertragen werden, wird zunächst überprüft, ob das Medium frei ist. Ist dies der Fall, können die Daten direkt übertragen werden. Ist das Medium belegt, wartet der Sender und kontrolliert in regelmäßigen Abständen die Belegung, bis das Medium schließlich frei ist und die Sendung beginnen kann. Da es bei mehreren wartenden Sendern auch zu Kollisionen in Folge von gleichzeitigen Übertragungen kommen kann, führen viele Protokolle eine zusätzliche zufällige Wartezeit innerhalb eines vorgegebenen Intervalls ein, wenn das Medium als frei erkannt wurde. Durch die zufällige Wahl der Wartezeit kommt es seltener zu den mit Kollisionen verbundenen Rahmenausfällen, ausschließen lassen sie sich damit jedoch nicht.

Multiplexverfahren unterteilen ein Medium z.B. in disjunkte Zeitslots oder Frequenzbänder und garantieren dem Sender, daß keine andere Station die Übertragung stört. Die Unterteilung kann dabei à priori erfolgen und damit fest kodiert, oder regelmäßig zwischen allen beteiligten Knoten ausgehandelt werden. Bei den Zugriffsverfahren unterscheidet man üblicherweise zwischen Frequenz- und Zeitmultiplexverfahren und der Kombination aus beiden. Frequenzmultiplexverfahren (FDMA) unterteilen das Medium in verschiedene Frequenzbereiche, in denen zwei Knoten zeitgleich senden können, ohne die Übertragung des anderen zu stören. Zeitmultiplexverfahren (TDMA) unterteilen das Medium in Zeitabschnitte, in denen der Sender diese Frequenz exklusiv belegt. Bei den Zeitmultiplexverfahren müssen alle Sender und Empfänger regelmäßig synchronisiert werden, um Kollisionen zu vermeiden, die durch Gangungenauigkeiten der lokalen Uhren hervorgerufen werden. Protokolle, die Multiplexverfahren implementieren, besitzen häufig auch einen wettbewerbsbasierten Zeitabschnitt, der entweder für dynamische Nachrichten oder zur Aushandlung der nächsten Reservierung benötigt wird. Weitere Verfahren sind das *Code Division Multiple Access* (CDMA) bei dem für unterschiedliche Übertragungen auf dem gleichen Medium unterschiedliche Kodierungen verwendet werden und das emphSpace Division Multiple Access (SDMA) das durch die Trennung der Übertragungswege (beispielsweise Richtfunk) auf der gleichen Frequenz mehrere zeitgleiche Übertragungen ermöglicht.

Für den Entwurf von energieeffizienten Protokollen, die gerade für batteriebetriebene Sensornetze wichtig sind, sind folgende Punkte besonders zu beachten [113]:

Untätiges Zuhören (*Idle Listening*) Ein Knoten hört einen Kanal auf mögliche Übertragungen ab, ohne selbst eine Übertragung zu erwarten oder einen Sendewunsch zu besitzen.

Überhören (*Overhearing*) Empfängt ein Knoten eine Nachricht, die nicht für ihn bestimmt ist, wird diese Nachricht dennoch empfangen und verarbeitet, nach der Erkennung des Ziels aber dennoch verworfen.

Kollisionen (*Collisions*) Zwei Knoten senden in Empfangsreichweite zum Zielknoten gleichzeitig bzw. überlappend. Beide Übertragungen sind beim Empfänger gestört und müssen erneut erfolgen. Ursache kann hierbei beispielsweise das

Hidden-Station-Problem sein, bei dem zwar der Empfänger die Signale beider Sender hört, die Sender sich gegenseitig jedoch nicht hören können.

Overhead Hierzu zählen alle Kontrolldaten und Rahmen die zusätzlich zu den Nutzdaten übertragen werden (beispielsweise RTS/CTS, ACK, ...)

Zur Reduzierung des Energieverbrauchs von Kommunikationssystemen existieren dazu sogenannte Betriebszyklusprotokolle (*duty cycle protocols*), deren Aufgabe es ist, den Transceiver und den Knoten selbst so kurz wie möglich aktiv zu halten. Man unterscheidet bei den Protokollen zwischen zentral gesteuerten (mit einem dedizierten Koordinator-Knoten), wie beispielsweise Bluetooth [14] und ZigBee (IEEE 802.15.4) [54, 106] und nicht zentral gesteuerten Protokollen. Nicht zentral gesteuerte Protokolle lassen sich in zwei Gruppen unterteilen: Protokolle mit periodischem globalem Synchronisationszeitpunkt (beispielsweise S-MAC, T-MAC, RMAC, DW-MAC und MacZ [11, 25, 28, 99, 113]) und Protokolle mit einer Signalisierungspräambel (beispielsweise STEM, B-MAC und WiseMAC [30, 88, 97]).

Die Signalisierungspräambel wird von einem Sender vor den Nutzdaten übertragen und muß so lang sein, daß ein schlafender Knoten diese erkennt und aufwacht. Dies kann entweder über spezielle Hardware erreicht werden, die trotz weitestgehend abgeschaltetem Transceiver eine Kanalbelegung erkennt, oder durch periodisches kurzzeitiges Aufwachen der einzelnen Knoten und Prüfung auf eine vorhandene Kanalbelegung. Der Empfänger-Knoten muß dazu seinen Transceiver aktivieren und summiert die erkannte Energie auf dem Medium über einen bestimmten Zeitraum auf. Liegt die ermittelte Energiemenge unter einem Schwellenwert, wird der Kanal als frei erkannt. Dieses Verfahren wird als *Clear Channel Assessment* (CCA) bezeichnet. Der Einsatz von Spezialhardware ist nicht überall möglich oder gewünscht und im Falle von gegebener Hardware sogar unmöglich, ist jedoch bezogen auf den Energieverbrauch die beste Variante bei sporadischem Datentransfer. Der periodische Aufwachvorgang zur Prüfung einer Kanalbelegung kostet regelmäßig Energie, weshalb sich diese Art der Übertragung nur eignet, wenn die Periode ausreichend groß gewählt werden kann und eine Modifikation der Hardware ausgeschlossen ist.

Bei Protokollen mit einem globalen Synchronisationszeitpunkt existiert ein Zeitpunkt, zu dem alle Knoten wach und empfangsbereit sind. Dieser Zeitpunkt kann entweder dazu verwendet werden, die Kommunikation bis zum nächsten Synchronisationszeitpunkt auszuhandeln oder dies als einzigen allgemeinen Kommunikationszeitpunkt (wettbewerbsbasiert oder über *Token-passing*) der Knoten im Netz zu verwenden. Damit alle Knoten immer zum gleichen Zeitpunkt aufwachen und kommunikationsbereit sind, müssen die internen Uhren der Knoten regelmäßig auf den Anfang der nächsten Periode synchronisiert werden. Ist es dabei ausreichend, nur den Anfang einer neuen Periode zu signalisieren, spricht man von der *Ticksynchronisati-*

on, bei der kein absoluter Zeitstempel übertragen wird. Für die Ticksynchronisation kann ein einzelner Tickframe eingesetzt werden. Sobald es sich jedoch um ein Netzwerk mit mehreren Hops handelt, erfordert die genaue Synchronisation der Knoten ein aufwändigeres Protokoll, wie z.B. die *Black Burst Synchronization* [48].

Middleware Sind an einer Regelstrecke viele Sensoren und Aktuatoren beteiligt, steigt die Wahrscheinlichkeit für den Ausfall eines Knotens. Aber auch durch den Austausch defekter Knoten oder das Anbringen neuer Sensoren ändert sich das Netz. Durch Einsatz einer Middleware, die von den einzelnen Knoten abstrahiert und stattdessen abstraktere Dienste zur Verfügung stellt, kann die Rekonfiguration des Netzes in diesen Fällen oft vermieden werden. Dies ist besonders dann interessant, wenn ein Sensorwert von mehr als einem Knoten verwendet wird, also keine eindeutige Zuordnung zwischen Sensoren, Reglern und Aktuatoren besteht. In diesem Fall kann eine Middleware diese Daten über eine Dienstabstraktion liefern und dabei auch eine Mehrfachübertragung des Sensorwertes aufgrund unterschiedlicher Zieladressaten vermeiden. Die Verwaltung der Dienste im Netzwerk kann dabei zentral auf einem Knoten erfolgen, oder dezentral über das Netzwerk verteilt werden, wie dies z. B. bei der *AmiCom* [37, 59] der Fall ist. Hier werden alle An- und Abmeldungen von Diensten im Netzwerk von jedem Knoten registriert bzw. über einen Suchalgorithmus Dienste mit bestimmten Eigenschaften gesucht und abonniert.

Regelung In einem Regelungssystem werden Sensoren verwendet, um den aktuellen Systemzustand zu einem Zeitpunkt t konsistent zu erfassen. Aufgrund dieser Meßwerte werden dann Stellwerte errechnet und mit geringer Verzögerung an die Akuatoren weiter gegeben. Sofern alle Komponenten fest mit dem Regler verdrahtet sind, signalisiert der Regler den Sensoren den Beginn einer Meßwerterfassung und friert danach seine Eingänge ein (*Sample & Hold-Verfahren*), sodaß die Werte nacheinander verarbeitet werden können, ohne daß sich diese noch ändern. Im verteilten System arbeiten die Sensorknoten, die zur Meßwerterfassung verwendet werden, zunächst unabhängig vom Regler. Um sicherzustellen, daß alle Sensoren gleichzeitig ihre Meßwerte erfassen, wird eine Ticksynchronisation benötigt, über die der gemeinsame Erfassungszeitpunkt signalisiert wird.

Sowohl die Meß- als auch die berechneten Stellwerte besitzen eine zeitlich beschränkte Gültigkeit und müssen in einer definierten Zeit zuverlässig beim Empfänger ankommen. Ist eine Übertragung nicht erfolgreich, so ist es in den meisten Fällen wegen der beschränkten Gültigkeit nicht sinnvoll, die Daten erneut zu übertragen.

Inverses Pendel Als Anwendungsfall für den Cross-Design-Ansatz wählten wir die Regelung eines inversen Pendels (siehe auch Kapitel 2.3), da dieses aufgrund seiner hohen Ansprüche an die Genauigkeit und Reaktionsgeschwindigkeit der Regelung eine große Herausforderung an verteilte Regelsysteme darstellt. Es handelt sich beim inversen Pendel um ein nicht stabiles, offenes System, das nicht minimalphasig [76] ist und mathematisch nicht durch ein lineares Modell beschrieben werden kann. In der Regelungstechnik wird das inverse Pendel häufig als Benchmark für ein nichtlineares Regelungssystem [4] verwendet, um damit die Performanz und die Güte der Regelung vergleichen zu können. Aufgrund äußerer Störeinflüsse, die auf das Pendel wirken, muß das Regelungssystem fast ständig den Führungsschlitten bewegen, um damit ein Umfallen des Pendels zu verhindern. Wird die Kommunikation zwischen Sensoren, Regler und Aktuator nicht mehr über eine exklusive Kabelverbindung realisiert (siehe Abbildung 5.3), ergeben sich hohe Anforderungen an die Zuverlässigkeit des Kommunikationssystems. Besitzt die Kommunikationsstrecke zu große Verzögerungen, muß der Führungsschlitten große Ausgleichsbewegungen fahren und das Pendel wirkt schnell instabil. Ist die Kommunikation nicht verlässlich und fallen in Folge dessen viele Rahmen nacheinander aus, eskaliert die Regelstrecke und das Pendel fällt um.

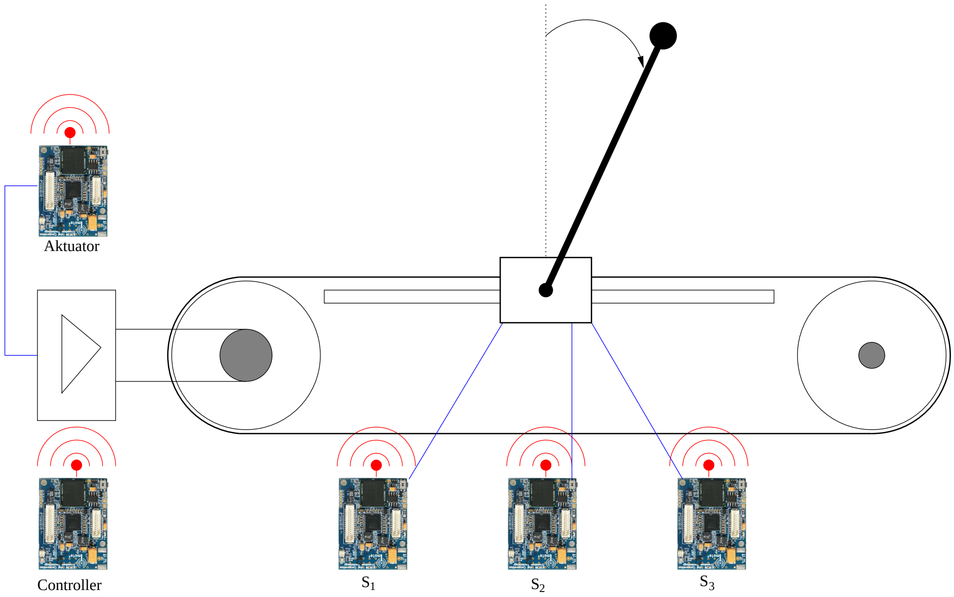


Abbildung 5.3.: Kommunikationsstrecken in der inversen Pendelregelung

5.1. Modellgestütztes Control-Computation-Communication-Cross-Design für funkbasierte Regelungssysteme

Regelungssysteme werden häufig über bereits bestehende Kommunikationsinfrastruktur und -protokolle miteinander vernetzt. Diese Protokolle bieten meist keine Garantien bezüglich Verlust und Verzögerung und werden aus Sicht der Regelung, als zufällig angesehen. Grund hierfür ist, daß es sich hierbei meist um Protokolle ohne Echtzeitfähigkeiten handelt, die keine Reservierungen beherrschen und stattdessen den Medienzugriff per Wettbewerb entscheiden. Es ist somit nötig, das gesamte Kommunikationssystem auf die Regelung abzustimmen bzw. nach dessen Vorgaben zu entwickeln. Dazu müssen zunächst die Anforderungen der Regelung bekannt sein und diese dann schrittweise umgesetzt werden.

5.1.1. Kommunikationsanforderungen

Am Beispiel eines physisch vorhandenen inversen Pendels und dessen Modell wurden die Anforderungen der Regelung an ein Kommunikationssystem ermittelt (Tabelle 5.1). Für die Stabilität des inversen Pendels ist die Abtastzeit h und die Gesamtverzögerung entscheidend: Die Abtastzeit bezeichnet dabei die Periode, in der die Sensorwerte an das Regelungssystem weitergegeben werden. Als Gesamtverzögerung bezeichnet man demgegenüber die Zeitspanne vom Beginn der Abtastung bis zu dem Zeitpunkt, an dem der Aktuator auf den resultierenden neuen Stellwert reagiert.

	min	opt
Abtastzeit h	80 ms	20 ms
Gesamtverzögerung	40 ms	8 ms
Datenrate	$4 \cdot 4$ Byte pro Abtastintervall	
Verlust (max)	1 pro $5\,h$	0

Tabelle 5.1.: Netzwerkdienstgüte für die inverse Pendelregelung

Wird die Abtastzeit zu groß gewählt, ist es nicht mehr möglich ein stark destabilisiertes Pendel vor dem Umfallen zu bewahren. Auch die Gesamtverzögerung sollte nicht zu groß werden, da der jeweilige Stellwert auf Grundlage der Meßwerte berechnet wird und bei zu großer Verzögerung die Abweichung der gemessenen Werte zur zwischenzeitlich veränderten Situation zu einer Reduktion der Regelgüte führt. Experimentell wurde für das vorliegende inverse Pendel ermittelt, daß es bis zu einer

Gesamtverzögerung von 120 ms stabilisierbar ist, sofern zu diesem Zeitpunkt keine starke externe Störung auf die Regelungsstrecke einwirkt. Bei einer Abtastzeit von 80 ms und einer Gesamtverzögerung von 40 ms ist somit ein einzelner Rahmenverlust gerade noch auszugleichen. Der Ausfall mehrerer aufeinander folgender Rahmen kann im allgemeinen nicht durch die Regelung ausgeglichen werden und führt fast immer zum Umfallen des Pendels.

5.1.2. Entwurf

Die Realisierung eines maßgeschneiderten Kommunikationssystems für die inverse Pendelregelung findet in mehreren Schritten statt und wird auf der Imote2-Plattform praktisch evaluiert. Die Spezifikation des Kommunikationssystems erfolgt in SDL, und über den in Kapitel 2.1 beschriebenen SDL-MDD-Ansatz wird hieraus direkt das fertige Programm generiert. Aufgrund der Aufgabentrennung zwischen Kommunikation und Regelung wird die Regelungsapplikation unabhängig in C++ entwickelt und greift nur auf die Kommunikationsfunktionen zu.

Erster Entwicklungsschritt Im ersten Entwicklungsschritt soll die Regelungsapplikation die volle Kontrolle über den Knoten erhalten, in Vorbereitung weiterer Schritte werden für die zur Ansteuerung benötigten Funktionen bereits Schnittstellen angeboten. Die Initialisierung und der Zugriff auf die Hardwarekomponenten wird von SEnF übernommen und die Schnittstelle dazwischen in SDL spezifiziert. Neben den Funktionen, die zum Zweck der Meßwerterfassung oder Steuerung des Aktuators direkten Zugriff auf die Hardware bieten, existieren auch Funktionen für den Zugriff auf den Transceiver. Im ersten Entwicklungsschritt existiert noch keine spezielle MAC-Schicht und es wird bei einem Sendeversuch der Regelungsapplikation lediglich die Kanalbelegung über CCA (*Clear Channel Assessment*) berücksichtigt und im Falle eines belegten Kanals der Sendeversuch ein weiteres Mal wiederholt. Dieser Schritt entspricht weitestgehend dem Vorgehen eines klassischen Entwurfs in der Regelungstechnik, bei dem ein Betriebssystem ähnliche Funktionen zur Verfügung stellt. Zunächst ist es bei diesem Entwurf ebenfalls einfach möglich, die komplette Regelung, Meßwerterfassung und Ansteuerung des Aktuators auf einem Knoten durchzuführen und damit zunächst erst einmal ein funktionierendes System zu erhalten, ohne Schwierigkeiten mit Netzwerkeffekten berücksichtigen zu müssen. Im folgenden kann die Netzwerkfunktionalität schrittweise hinzugefügt werden und so von einer Ein-Knoten-Lösung bis zu einer Fünf-Knoten-Lösung erweitert werden.

Abbildung 5.4.

Aufgrund des Fehlens von Synchronisation und spezifischen Netzwerkprotokollen wird die Netzwerkkommunikation durch den Regler-Knoten initiiert, indem dieser die Daten der Sensoren explizit anfordert.

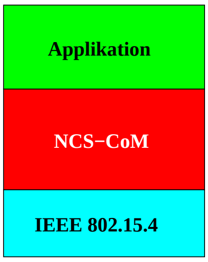


Abbildung 5.5.

Zweiter Entwicklungsschritt Im zweiten Entwicklungsschritt erfolgt der Übergang zu einer dienstorientierten Architektur, in der eine *Middleware*, die NCS-CoM (*Network Control Systems-Communication Middleware*) zwischen Hardware und Regelungsapplikation eingefügt wird. Jede Applikation registriert dazu die von ihr angebotenen Dienste bei der Middleware bzw. abonniert die Dienste, die sie benötigt. Die NCS-CoM übernimmt daraufhin die Verwaltung der Dienste, die Verteilung der Information über vorhandene Dienste im Netzwerk und die Übermittlung der Daten. Zu jedem Abonnement teilt der Dienstnutzer die gewünschte Periode mit, mit der er die Daten erhalten will. Die Middleware des Diensterbringers verwaltet alle Anforderungen, deren Periode und fragt bei der lokalen Applikation rechtzeitig dazu die Daten an. Diese Daten werden dann an alle Abonnenten weitergegeben, sowohl auf dem aktuellen, als auch auf entfernten Knoten. Durch die Realisierung der Applikationen als einzelne Dienste können hier die Dienste z. B. einfach von einer Ein-Knoten-Lösung auf eine Fünf-Knoten-Lösung erweitert werden, ohne daß Anpassungen an den anderen Diensten und Anwendungen nötig sind. In diesem Entwicklungsschritt ist lediglich eine rudimentäre MAC-Schicht vorhanden, die für die entfernte Kommunikation lediglich die aktuelle Kanalbelegung über das CCA-Signal beachtet und dann mittels *Best-Effort*-Strategie den Zugriff auf das Medium zuläßt. Es handelt sich hierbei um eine einfache Form des Kanalzugriffs, das so genannte *Carrier Sense Multiple Access/Collision Avoidance* (CSMA/CA) Verfahren. Aufgrund der rudimentären MAC-Schicht und der fehlenden Synchronisation ist bei nicht lokalem Dienstabonnement weiterhin mit Rahmenausfällen oder asynchroner Meßwerterfassung und somit schlechter Regelungsperformance zu rechnen.

Dritter Entwicklungsschritt Der dritte Entwicklungsschritt dient der Entwicklung einer eigenen MAC-Schicht (MacZ light) und sorgt mittels TDMA-Verfahren für eine zuverlässige Kommunikation zwischen den Knoten. Das Kommunikationsmedium wird dazu in gleich große Zeitslots eingeteilt die zur exklusiven Kommunikation den jeweiligen Knoten zugeteilt wurden. Eine Weiterleitung (Routing) von Daten ist in diesem Entwicklungsschritt noch nicht geplant, sodaß alle Knoten in Kommunikationsreichweite des Regelungsknotens sein müssen. Damit alle Knoten zum richtigen Zeitpunkt sende- bzw. empfangsbereit sind, wurde eine Tick-Synchronisation in die MAC-Schicht integriert. Die Synchronisation wird jedoch nicht nur für die Slot-Einteilung genutzt, sondern zusätzlich in der NCS-CoM publiziert. Die NCS-CoM verwendet diesen Tick-Zeitpunkt, um von allen verteilten Applikationen synchron den Meßwert erfassen zu lassen. Durch die Verwendung des TDMA-Protokolls und die Kontrolle der Middleware läßt sich sehr genau voraussagen, zu welchem Zeitpunkt welche Komponente des Knotens aktiviert werden muß. Über die Energiesignalisierung (siehe Kapitel 3.3) können die jeweiligen Komponenten explizit und implizit gesteuert werden, ohne daß die Applikation hierfür Vorkehrungen treffen muß. In diesem Entwicklungsschritt sind aufgrund der Dienstabstraktion nur Anpassungen aufgrund der neu hinzugekommenen Tick-Synchronisation nötig.

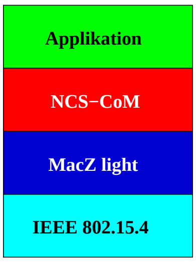


Abbildung 5.6.

Vierter Entwicklungsschritt Der vierte und letzte Entwicklungsschritt soll eine Regelung auch über mehrere Knoten und Hops hinweg zuverlässig ermöglichen, ist aber nicht mehr Gegenstand dieser Arbeit. In Abbildung 5.7 sind der gesamte Entwicklungsablauf und auch die geplanten Erweiterungen und Änderungen für diesen letzten Schritt dargestellt. Das für die Datenverteilung und Weiterleitung zuständige Routingprotokoll wird in dem Stack zwischen MAC-Schicht und Middleware positioniert. Da die Topologie trotz Routing eher statisch ist, dennoch möglichst wenige Daten im Netzwerk ausgetauscht werden sollen, kommt hier *BBQR* [9] als Routingprotokoll zum Einsatz. Aufgrund der fehlenden dynamischen Reservierung in der einfach gehaltenen MAC-Schicht *MacZ light* wird diese schließlich durch das universellere *MacZ* [11, 69, 70] und das *Black Burst Synchronisation-Protokoll* (BBS) [49] ausgetauscht. Der Einsatz von BBS garantiert hierbei eine sehr präzise Synchronisation auch über mehrere Hops hinweg.

5.2. Middleware

Bei einer Middleware handelt es sich um eine Vermittlungssoftware in einem verteilten Netzwerk, die einer Anwendung Zugriff auf Dienste im Netzwerk gewährt und dabei von dem Netzwerkzugriff abstrahiert. Eine Applikation, die auf einer Middleware aufbaut, kann Dienste publizieren, um diese netzwerkweit zugreifbar zu machen, und fremde Dienste über einen Aufruf nutzen. Die Middleware sorgt dabei für die Abstraktion, sodaß es für die Anwendung unerheblich ist, ob der Diensterbringer auf dem gleichen Knoten läuft, oder über ein Netzwerk angebunden ist. Aufgrund des

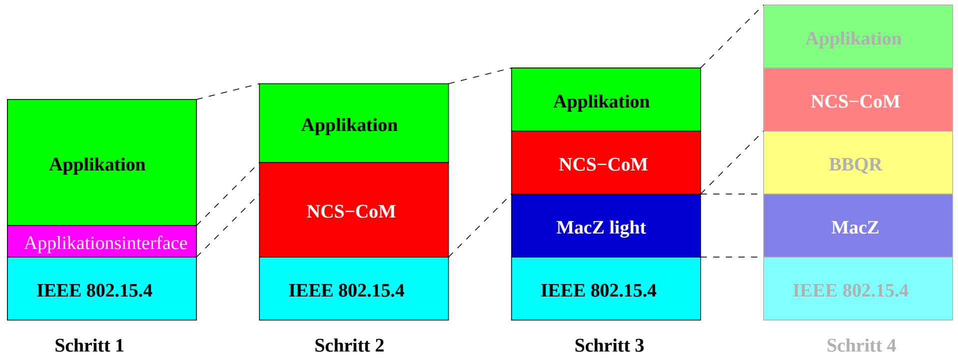


Abbildung 5.7.: Entwicklungsschritte der Pendelregelung

objektorientierten Zugriffsverfahrens wird diese Form der Middleware auch als *objektorientierte Middleware* [101] bezeichnet, die ihre Dienste als Objekte mit darauf definierten Funktionen anbietet. Zur Datenübertragung über ein Netzwerk spezifiziert die Middleware kein eigenes Protokoll, sondern verpackt ihre Daten in einen Datenstrom, der über jedes Standardprotokoll, wie beispielsweise TCP/IP oder UDP, transportiert werden kann. Die meisten Middleware-Implementierungen sind nicht für die Nutzung auf Sensorknoten konzipiert und würden zu viel Ressourcen benötigen. Deshalb wurde von I. Fliege die *AmICoM* [33, 37], eine Middleware speziell für Sensornetze entwickelt, die als Grundlage für die NCS-CoM dient. Die Middleware benötigt sowohl Schnittstellen zur Applikation, als auch zur MAC-Schicht. Dem Entwicklungsverlauf folgend wird nun zunächst die Schnittstelle der Applikation ohne eine Middleware besprochen.

5.2.1. Applikationsschnittstelle

Für die spätere Anbindung der Anwendung an eine Middleware und zur Anbindung der Anwendung an SDL muß zunächst eine gemeinsame Schnittstelle festgelegt werden. Die hierauf aufbauende Middleware soll im weiteren Verlauf in SDL spezifiziert werden und die Funktionen von SdlRE und SEnF zum Scheduling und zur Anbindung der Hardware benutzt werden. Da SEnF zur Verwendung mit SDL-Signalen aus SDL spezialisiert ist und um unerwünschte Nebeneffekte durch falschen Hardware-Zugriff auszuschließen, werden alle Hardwarefunktionen ebenfalls über diese Schnittstelle als Aufrufe aus SDL heraus definiert.

```
/*** Application functions ***/
//called on start, if the underlying system is ready
void Init();

//set a periodical (ms) event in miliseconds, that is identified by no. set ms=0 to deactivate
extern void SetPeriod(int no,unsigned int ms);

//periodical event no has expired
void Execute(int no);

/*** I/O functions ***/
//set (activate=true) or unset (actviate=false) an I/O interrupt
extern void IoSetInterrupt(unsigned int no, bool activate);

//called if I/O interrupt no has been captured
void IoInterruptCaptured(int no);

//Config Spi port to send bits with 13Mhz/(speed+1), if master: generate clock, set clock polarity with phase
extern void SpiConfig(int port,int bits, int speed, bool master, bool polarity, bool phase, bool frm_polarity);

//Send data through Spi port, return value is the received data (synchronous transfer)
extern int SpiSend(int port, int data);

/*** direct Communication functions ***/
//send data of length with transceiver
extern void Send(const char* data,unsigned int length);

//data of length received from transceiver
void Receive(const char* data, unsigned int length);

/*** Debug functions ***/
//set Imote LED status of led no to val (bool)
extern void SetLed(unsigned int no,unsigned int val);

//toggle LED no
extern void ToggleLed(unsigned int no);

//Send data of length to UART port
extern SendUart(unsigned int port, const char* data, unsigned int length);

//data from UART port with length received
void ReceiveUart(unsigned int port,const char* data,unsigned int length);
```

Listing 5.1: Applikationsschnittstelle

Im ersten Entwicklungsschritt stellt die Schnittstelle (Listing 5.1) für die Ablaufsteuerung der Anwendung drei zentrale Funktionen bereit. Die `Init`-Funktion ist die initiale Funktion der Applikationsschnittstelle, die von der Anwendung implementiert werden muß und von der Middleware aufgerufen wird. Diese Funktion wird erst aufgerufen, nachdem die SDL-Laufzeitumgebung komplett initialisiert ist und alle SDL-Prozesse ihre Starttransition abgearbeitet haben. Alle Hardwarefunktionen sind somit vollständig initialisiert, und im SDL-System sind alle SDL-Prozesse, Kanäle und Datenstrukturen erzeugt. Sie läßt sich mit dem vollständigen Starten eines Betriebssystems vergleichen. Die Anwendung sollte hier Variablen anlegen und die Hardware über die angebotenen Funktionen initialisieren. Anders als dies in der Regelungstechnik sonst üblich ist, darf die Anwendung hier nicht die Kontrolle behalten, sondern wird wie alle anderen SDL-Prozesse auch nur über Ereignisse gesteuert. Sollen periodische Ereignisse realisiert werden, wie das Auslesen eines Sensorwertes, kann mit Hilfe der Funktion `SetPeriod` ein periodisches Timer-Ereignis registriert werden. Sobald der Timer abläuft, wird die Applikation aktiviert und die Funktion `Execute` der Applikationsschnittstelle aufgerufen. Um verschiedene Timer-Ereignisse unterscheiden zu können, kann beim Setzen des Timers eine ID vergeben werden, die dann beim Aufruf der Execute-Funktion mit übergeben wird.

Die Laufzeitumgebung SdlRE unterstützt derzeit nur kooperatives Multitasking, und kann damit einer Anwendung nicht die Kontrolle entziehen. Bei der Anwendungsentwicklung muß dieser Sachverhalt berücksichtigt werden, und ein ereignisgesteuertes Design verwendet werden, bei dem der Prozessor nur zur Abarbeitung des Ereignisses benötigt wird. Dementsprechend sind *Busy-Waiting*-Schleifen in der Regelungsanwendung verboten und sind durch Ereignisse, Timer oder den Aufruf von Hardwarefunktionen zu ersetzen. Neben den periodischen Ereignissen können mit Hilfe der `IoSetInterrupt`-Funktion Hardware-Interrupts registriert werden, um damit auf spezielle Hardware-Bedingungen zu reagieren. Ist die Interrupt-Bedingung eingetreten, wird dies der Applikation durch Aufruf der Funktion `IoInterruptCap_tured` gemeldet und die eingetretene Bedingung mit angegeben. Da das Warten auf Änderungen eines Hardware-Zustands die häufigste Ursache für *Busy-Waiting* darstellt, sollte hierfür möglichst immer dieses Interface verwendet werden, da in der Zwischenzeit die Kontrolle zurück an die Laufzeitumgebung gegeben werden kann. Dort können dann entweder andere Prozesse rechnen oder der Knoten kann ohne weitere lauffähige Prozesse in den Schlafzustand gehen und damit Energie sparen.

Neben den vorgestellten Funktionen zur Ablaufsteuerung stehen noch weitere Funktionen zur Ansteuerung der auf dem Imote2 vorhandenen Hardware zur Verfügung, wie beispielsweise des SPI-Busses oder der LED und des UART-Ports, die zur Diagnose und Fehlersuche verwendet werden können. Der Zugriff auf den Transceiver zur Kommunikation ist hier ebenfalls noch direkt und ohne MAC-Protkoll realisiert. Es wird lediglich beim Sendeversuch die aktuelle Kanalbelegung mittels hardwaregestütztem Clear Channel Assessment geprüft, und sofern das Medium nicht frei ist, dieser Versuch selbständig ein weiteres Mal wiederholt.

5.2.2. Network-Control-Systems-Communication-Middleware

Nachdem die Regelungsapplikation über die Applikationsschnittstelle bereits auf die Imote2 Hardware zugreifen konnte und damit zunächst bei lokaler Anbindung die Stabilisierung des inversen Pendels möglich war, folgt nun im zweiten Entwicklungsschritt die Verteilung der Anwendung. Jede Sensor- und Aktuator-Applikation registriert dazu ihren Dienst bei der lokalen MiddlewareInstanz, die diesen dann netzwerkweit propagiert. Über die lokale Middleware-Instanz fragt die Regler-Applikationen alle im Netzwerk verfügbaren Dienste ab und abonnieren die für ihre Funktion benötigten Dienste mit der gewünschten bestimmten Periode. Abbildung 5.8 zeigt dazu einen Ausschnitt des Kommunikationsablaufs innerhalb der NCS-CoM des inversen Pendels mit drei Knoten als MSC.

In der Start-Phase registriert die Sensor-Anwendung zunächst den von ihm angebotenen Dienst `angle`, der den aktuellen Neigungswinkel des Pendels bereitstellt. Die Middleware publiziert diesen Dienst dann im Netzwerk mittels `alive`. Die lokale Middleware-Instanz überwacht die Anwendung und sendet, solange diese vorhanden ist, in regelmäßigen Abständen erneut eine `alive`-Nachricht in das Netzwerk. Empfängt eine entfernte Middleware-Instanz über längere Zeit keinen *Heart-Beat* oder eine explizite Deregistrierung des Dienstes, löscht diese den Dienst aus der lokalen Liste und informiert die dort registrierten Applikationen hierüber. Im nächsten Schritt registriert der Aktuator-Knoten auf gleiche Weise einen Dienst, über den sich der Wagen des Pendels steuern läßt. Nachdem alle Dienste für den Regler im Netzwerk bereitgestellt wurden, abonniert dieser sowohl den Dienst des Winkelsensors mit einer Periode von 30 ms als auch den Motor-Dienst, jedoch ohne eine Periode anzugeben. Beide Dienste bestätigen darauf das Abonnement und fügen den Knoten ihrer lokale Liste der Abonnenten hinzu. Sollte der Dienstanbieter die gewünschte Periode nicht erfüllen können, wird dies ebenfalls beim Abonnieren des Dienstes mitgeteilt. Nach dem erfolgreichen Abonnement eines Dienstes mit periodischer Signalisierung veranlaßt die Middleware die Applikation zum regelmäßigen Versand der Daten. Die Middleware adressiert jedoch die Daten nicht explizit an einen Abonnementen, sondern verteilt die Information per Broadcast im Netzwerk, sodaß diese jeder Middleware zur Verfügung steht. Erst die lokale Middleware-Instanz entscheidet, ob dieser Dienst abonniert wurde und leitet die Daten an die jeweilige Applikation auf dem Knoten weiter, oder verwirft sie, sofern es keine Applikation gibt die diese Daten abonniert hat. Der Regler selbst zieht sich unabhängig von der Middleware einen Timer auf, die bei Ausbleiben des Sensorwertes einen berechneten Modellwert an

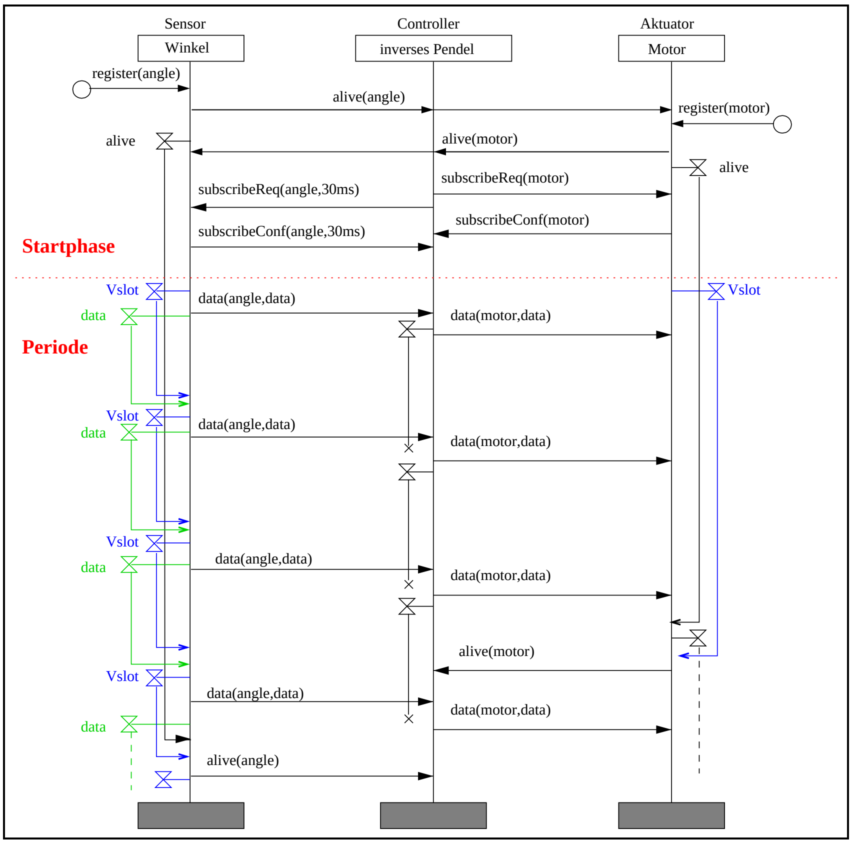


Abbildung 5.8.: MSC für NCS-CoM des inversen Pendel

den Aktuator-Knoten sendet. Erreicht der Sensorwert den Regler rechtzeitig, bricht dieser den Timer ab, berechnet aus dem Sensorwert einen neuen Stellwert und sendet diesen an den Aktuator-Knoten und zieht die Uhr erneut auf.

Die Architektur der NCS-CoM mit der zugehörigen Applikationsschnittstelle, wie sie in SDL modelliert ist, zeigt Abbildung 5.9. Die Applikationsschnittstelle ist im SDL-Prozess `Application` innerhalb des SDL-Blocks `ApplicationAdaption` realisiert. Die NCS-CoM wird durch den gleichnamigen Block repräsentiert und besteht aus mehreren Prozessen: Der `ServiceObserver` ist für die Überwachung der Applikation innerhalb der Middleware zuständig, prüft ob diese noch auf Signale reagiert und versendet daraufhin periodisch ein `alive`-Signal. Jeder Knoten enthält für jeden Dienst, der von der Applikation angeboten wird, eine Instanz des SDL-Prozesses

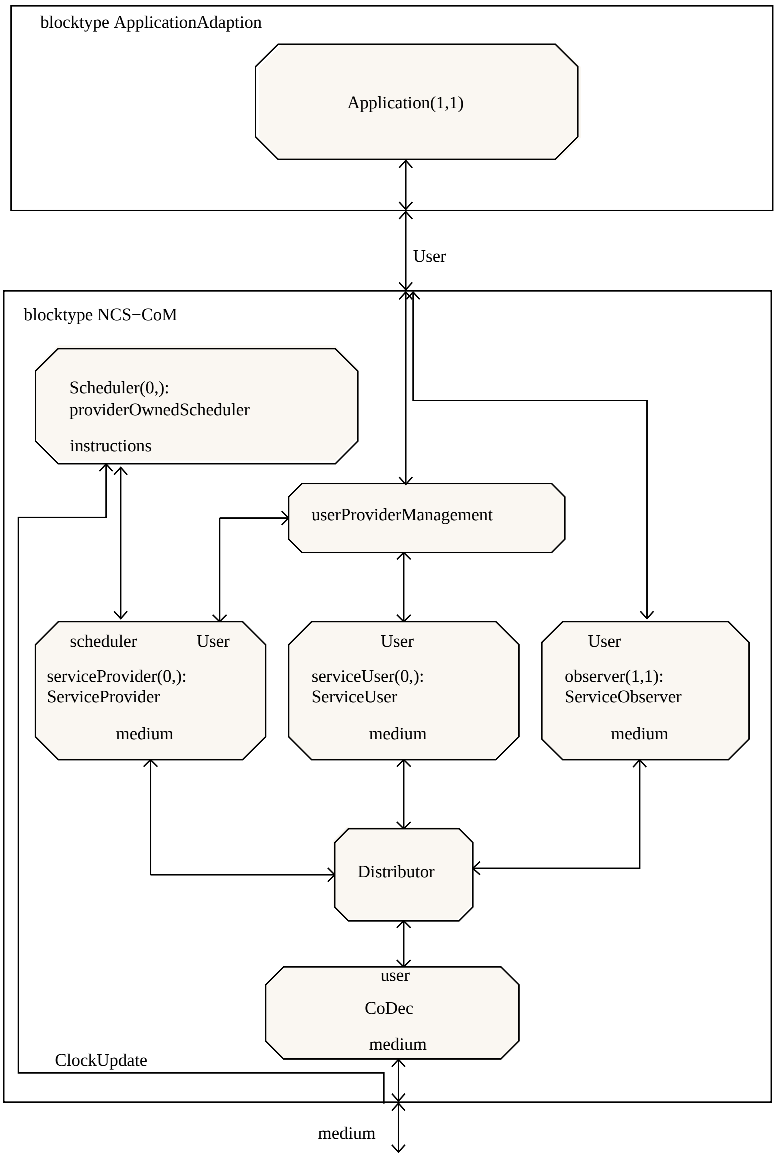


Abbildung 5.9.: NCS-CoM

`ServiceProvider` und für jeden abonnierten Dienst eine Instanz von `ServiceUser`. Die Repräsentation und Verwaltung gegenüber der Applikation wird transparent durch den SDL-Prozess `userProviderManagement` dargestellt. Der Distributor verbindet alle drei Prozesse und entscheidet, welche Anfragen an das Netzwerk weitergeleitet werden müssen oder von einem lokalen Repräsentanten beantwortet werden können. Bevor die Daten jedoch an das Netzwerk weitergeleitet werden können, sorgt der SDL-Prozess `CoDec` für eine ASN.1-BER-Kodierung der Daten zu einer Bytefolge und gibt diese zusammen mit dem Dienstnamen an die MAC-Schicht weiter. Die aus dem Netzwerk stammenden Daten werden nach gleichem Schema dekodiert und über den `Distributor` an den richtigen lokalen Repräsentanten weitergeleitet. Der SDL-Prozess `providerOwnedScheduler` [111] ist für den periodischen Versand von Daten eines Diensterbringers zuständig. Registriert eine Applikation einen Dienst, teilt sie nicht nur den Dienstnamen mit, sondern ebenfalls die minimale Anfragehäufigkeit sowie die benötigte Verarbeitungszeit. Der Scheduler verwaltet nun einen Zeitplan, in dem er alle Dienstabonnements mit ihren periodischen Anfragen unter Berücksichtigung der Totzeiten des Diensterbringers einplant. Unter Berücksichtigung der Verarbeitungszeit werden die Anfragen an die Applikation so früh gestellt, daß diese ebenfalls rechtzeitig in der MAC-Schicht zur Verarbeitung ankommen.

Durch die Einführung einer Middleware mußte die Schnittstelle für die Applikation erweitert und an die neuen Möglichkeiten angepaßt werden. In Listing 5.2 sind die neuen und geänderten Funktionen für die Middleware gezeigt. Neu hinzugekommen sind Funktionen zur Registrierung und zum Abonnement von Diensten, sowie eine Abfrage aller verfügbaren Dienste.

Bei der Registrierung eines Dienstes durch die Applikation wird neben dem Dienstnamen auch die Verarbeitungszeit und die minimale Zeit zwischen zwei Abfragen angegeben. Beim Abonnieren eines Dienstes mittels `subscribeService` muß der zu abonnierende Dienstname, die gewünschte Periode des Abonnements und eine maximal akzeptierte Verzögerung angegeben werden. Der für den Dienst zuständige Scheduler [111] kann aus diesen Angaben einen Plan erstellen, der ähnlich zum Scheduler für Jitter aus Kapitel 3.4 ist, mit dem Ziel, möglichst selten den Dienst zu aktivieren und und dabei Daten zu versenden. Ein Diensterbringer muß die Funktion `requestDataForService` bereitstellen, über die ihm die Applikationsschnittstelle mitteilt, daß Daten für einen zuvor registrierten Dienst bereitgestellt werden müssen. Die berechneten oder ausgelesenen Daten werden der `setRequestedData`-Funktion übergeben und dann von der Middleware weiterverarbeitet. Ein Dienstnutzer empfängt alle so versendeten Daten über die `Receive`-Funktion, die nun nicht mehr nur die Daten, sondern auch einen Verweis auf deren Herkunft enthält. Die beim Empfang ebenfalls vorhandene Sequenznummer (`seqNo`) enthält den Zähler des aktuellen

```
/**** Service Mangement Functions *****/
/*register Service with name, and a reactionDelay in ms, pollTime is the least time this
    service can be polled */
extern void registerService(const char* name, const int reactionDelay, const int pollTime);

/*deRegister a prior registered Service*/
extern void deregisterService(int id);

/*subscribe to Service name with a period and a maximum delay*/
extern int subscribeService(const char* name, const int period, const int delay);

/*desubscribe a prior subscribed service*/
extern void unsubscribeService(int id);

/**** Service Communication Functions ****/
/* data for service is requested, period is the subscribed period. This method gets called
    every 'period' milliseconds. After it was called, the application has the (at registration
     time) specified time to sample and/or calculate the data, call 'setRequestedData' and
    have it transmitted to the subscriber as soon as the specified sampling/calculation time is
     over*/
void requestDataForService(const int serviceID,const int seqNo);

/* set data according to the requestDataForService call */
extern void setRequestedData(const int serviceID, const char* data, int length);

/* received a data for this application with serviceId */
void Receive(int serviceId, const char* data, unsigned int length,int seqNo);

//Send data of length to service (via transceiver or local application),
extern void Send(const int service,const char* data, unsigned int length, bool reserved);

/**** CHANGED Interface ****/
//void Send, void Receive
```

Listing 5.2: Erweiterung der Applikationsschnittstelle für die NCS-CoM

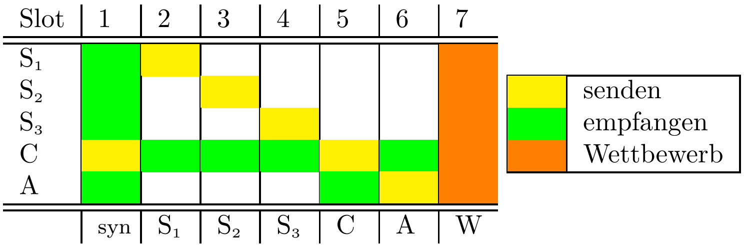
Kommunikationszyklus[31] und dient der Zuordnung von Nachrichten mit gleichem Zeitstempel, wie dies beim Reglerknoten nötig ist. Die Sequenznummer wird erst mit Einführen einer Ticksynchronisation im dritten Entwicklungsschritt verwendet. Für den Versand von nicht abonnierten, nicht periodischen Daten, kann die Funktion `Send` zusätzlich genutzt werden. Diese Funktion umgeht die Mechanismen der Middleware und interagiert direkt mit der MAC-Schicht, wodurch es möglich ist, Daten der Applikation direkt an Knoten ohne die Middleware zu versenden.

---

[31] Ein Kommunikationszyklus stellt eine vorher festgelegte Zeitspanne dar. Ist diese abgelaufen, erhöht sich der Zähler. Eine genaue Beschreibung ist in Kapitel 5.3 zu finden.

5.3. MacZ light

Für die Funkkommunikation beim inversen Pendel zwischen allen beteiligten Knoten wurde ein TDMA-Verfahren implementiert und das Medium in Slots gleicher Länge unterteilt. Die Zuordnung der Slots zu den Kommunikationspartnern S_1, S_2, S_3, Regler und Aktuator erfolgt nicht über ein dynamisches Reservierungsprotokoll, sondern wird à priori statisch festgelegt und wiederholt sich periodisch.[32] Der Zugriff auf das Medium erfolgt somit deterministisch ohne Wettbewerb. Zusätzlich zu den regulären Kommunikationsslots, die für die Meß- und Stellwerte benötigt werden, ist ein weiterer Slot zur Synchronisation und Slots zum Austausch von Statusinformationen wie dem Abonnement nötig. Die minimale Slotkonfiguration ist in Abbildung 5.10 für jeden Knoten gezeigt. Slot 1 dient der Synchronisation des Mediums und um die Abweichung der lokalen Uhren auszugleichen. Nachfolgend erhält jeder Knoten einmal exklusiven Zugriff auf das Medium, um seine Daten zu versenden. Der letzte Slot (7) dient dem Austausch von Verwaltungsinformationen der NCS-CoM und beinhaltet Abonnements und Heartbeats und findet in gegenseitigem Wettbewerb statt. Die Slotlänge wird für alle Daten auf die gleiche Länge gesetzt und muß damit groß genug sein, um ein Datenpaket maximaler Länge aus der Applikation aufzunehmen. Um diese Länge nun zu bestimmen, wird aufgrund der Angaben aus dem Datenblatt des Transceivers der CPU des Knotens und der Datenlänge die Zeit ermittelt.



Slot	1	2	3	4	5	6	7
S_1	🟩	🟨					🟧
S_2	🟩		🟨				🟧
S_3	🟩			🟨			🟧
C	🟨	🟩	🟩	🟩	🟨	🟩	🟧
A	🟩				🟩	🟨	🟧
	syn	S_1	S_2	S_3	C	A	W

🟨	senden
🟩	empfangen
🟧	Wettbewerb

Abbildung 5.10.: Minimal benötigte Slots für die Kommunikation der NCS-CoM mittels TDMA

5.3.1. Berechnung der Slotzeiten

Die Slotzeit könnte zwar auch rein experimentell ermittelt werden, hier soll jedoch die rechnerische Bestimmung der Länge über die Datenblätter als untere Schranke

---

[32] Die statische Reservierung ist als Zwischenschritt zu betrachten und wird mit dem vierten Entwicklungsschritt durch eine dynamische Reservierung ersetzt.

dienen. Auf der Imote2 Plattform mit dem CC2420-Transceiver ergeben sich dabei die Zeiten aus dem Datenblatt wie in Tabelle 5.2 aufgelistet. Jedes Byte Nutzdaten sowie das Längenbyte und die Steuerkommandos müssen zunächst zum Transceiver über den internen SPI-Bus übertragen werden. Ist der Bus nicht belegt, benötigt jedes Byte dafür τ_{SPI}. Für die Funkübertragung mittels *Clear Channel Assessment* (CCA), benötigt der Transceiver maximal die Zeit τ_{CCAmax}, um zunächst zu prüfen, ob der Kanal frei ist.

Transfer CPU → CC2420 1 Byte	τ_{SPI}	$= 1{,}4\,\mu\mathrm{s}$
CCA valide	τ_{CCAmax}	$= 128\,\mu\mathrm{s}$
Umschaltzeit (Empfang→ Senden)	τ_U	$= 192\,\mu\mathrm{s}$
Präambel	τ_{Pre}	$= 128\,\mu\mathrm{s}$
Übertragung 1 Byte	τ_B	$= 32\,\mu\mathrm{s}$
Hardware-Rahmen: SFD (1 Byte), Länge (1 Byte), CRC (2 Byte)	$4 \cdot \tau_B$	$= 128\,\mu\mathrm{s}$
Hardware-Rahmen ohne Längenbyte	$\tau_{\mathrm{Rl}} = 3 \cdot \tau_B$	$= 96\ \mu\mathrm{s}$

Tabelle 5.2.: Hardwarezeiten für Datenversand

Hardwarezeiten Transceiver Ist der Kanal frei, schaltet der Transceiver innerhalb von τ_U vom Empfangs- in den Sendemodus um und überträgt zunächst eine Präambel, die für τ_{Pre} das Medium belegt. Der Empfangsknoten erkennt hierbei zunächst den Kanal als belegt und schaltet, sobald er eine Präambel vollständig erkannt hat, seinen Dekoder ein und versucht die nachfolgenden Daten zu dekodieren. Den Nutzdaten eines Datenrahmens wird auf dem CC2420 zusätzlich ein Startbyte (SFD) und ein Längenbyte vorangestellt, sowie eine zwei Byte Prüfsumme als CRC-16 (*Cyclic Redundancy Check*) angefügt. Nach Ende des Sendevorgangs schaltet der Transceiver automatisch zurück in den Empfangsmodus und benötigt dafür τ_U, bis er erneut bereit ist, Daten zu empfangen.

Ein minimal für den Transceiver gültiger Datenrahmen besteht lediglich aus dem Längenbyte, das dem Transceiver mitteilt, daß keine Daten enthalten sind. Ohne Beachtung von Steuerkommandos an den Transceiver läßt sich die hierfür benötigte Zeit berechnen:

$$\begin{aligned} &\tau_{\mathrm{send}}(n) = (n+1)\cdot\tau_{\mathrm{SPI}} + \tau_{\mathrm{CCAmax}} + \tau_U + \tau_{\mathrm{Pre}} + (4+n)\cdot\tau_B \\ &\tau_{\mathrm{send}}(1) = 578{,}8\,\mu\mathrm{s}. \end{aligned}$$

Soll der Transceiver danach wieder Daten empfangen, erhöht sich die Zeit um τ_U auf 769,4 μs und stellt damit die untere Schranke für den Datentransfer zwischen zwei Knoten dar. Auf der Empfänger-Seite ist noch nicht berücksichtigt, daß die Daten zwar im Puffer des Transceivers gespeichert sind, jedoch aus diesem noch nicht per SPI an die CPU und damit in den lokalen Speicher des Prozessors transferiert sind. Folgt eine Datenübertragung, die von diesem Transceiver empfangen wird, bevor die Daten aus dem Empfangspuffer gelesen wurden, werden diese überschrieben. Es muß somit immer darauf geachtet werden, daß zwischen zwei aufeinander folgenden Transfers ausreichend Zeit vergeht, daß die Daten aus dem Transceiver gelesen werden können. Auf Empfängerseite stehen die Daten im optimalen Fall nach

$$\begin{aligned}\tau_{\text{receive}}(n) &= \tau_{\text{send}}(n) + (n+1)\cdot\tau_{\text{SPI}}\\ \tau_{\text{receive}}(1) &= 580,2\,\mu\text{s}\end{aligned}$$

zur Verfügung. Hierbei sind ebenfalls keine Steuerkommandos berücksichtigt, die benötigt werden, um die Daten zu empfangen oder auch die Zeit, die zur Ausführung der Interrupt-Routine benötigt wird.

Name	Typ	ASN.1-Größe
Applikations-ID (≤255)	Integer	3 Byte
Knotenname (IMXX)	Octet_string	2 + 4 = 6 Byte
Servicename	Octet_string	2 + 2 = 4 Byte
Periode (≤255)	Integer	3 Byte
Zeitstempel	Integer	2 + 4 = 6 Byte
Payload Anwendung		(siehe Tabelle 5.4)
Σ		22 Byte

Tabelle 5.3.: Nachrichtenformat Middleware

Nachrichtenformat Middleware Außer den Daten, die durch die Hardware den Datenrahmen hinzugefügt werden, benötigt die MAC-Schicht keine weiteren Daten zur Adressierung. Es werden somit direkt die als ASN.1 kodierten Nutzdaten der Middleware in den Hardware-Rahmen eingesetzt. Tabelle 5.3 listet alle anfallenden Daten der Middleware und deren Größe in ASN.1 auf. Das Paar aus Knotenname und Applikations-ID kennzeichnet die Applikation, von der die Daten stammen, im Netzwerk eindeutig und kann für ein Routing Protokoll benutzt werden. Für die Middleware ist der Servicename, der bei der Registrierung vergeben wurde, entscheidend für die Zuordnung zwischen Applikation und Dienst. Beim Abonnement eines Dienstes wird ein Dienst mit einer bestimmten Periode abonniert. Beim Datenversand fügt die Middleware des Diensterbringers den Wert *Periode* in den Datenrahmen ein, um damit zu kennzeichnen, zu welcher Periode der ermittelte Wert gehört. Damit die Middleware entscheiden kann, ob ein Wert an den Dienstabonnenten übergeben wird, überprüft sie nicht nur den Dienstnamen, sondern auch ob der übermittelte Wert zu der abonnierten Periode gehört. Stimmt die Periode nicht überein, verwirft die Middleware den Wert, um der Applikation keine unerwarteten Daten zu übermitteln. Ein ebenfalls von der Middleware eingefügter Zeitstempel ist für den Empfangsknoten wichtig, um das Alter der Daten zu bestimmen und um Werte, die zu einem konsistenten Systemzustand verteilt ermittelt wurden, als zueinander gehörig zu erkennen. In Summe ergibt sich für die Übertragung der Middleware-Daten ein Datenstrom der Länge von 22 Byte.

Name	Typ	ASN.1-Größe
ID des Knotens	Integer	$2 + 4 = 6$ Byte
Sensorwert	Integer	$2 + 4 = 6$ Byte
Stellwert für Aktuator	Integer	$2 + 4 = 6$ Byte
Σ		12 Byte

Tabelle 5.4.: Nachrichtenformat Anwendung

Nachrichtenformat Anwendung Im Vergleich zu den anfallenden Daten auf Middleware-Ebene, werden auf Anwendungsebene fallen auf Anwendungsebene deutlich weniger Daten an, wie Tabelle 5.4 zeigt. Neben den Sensorwerten bzw. dem Stellwert für den Aktuator fügt die Applikationen zusätzlich noch die Knoten-ID ein. Diese redundante Information dient ausschließlich der Fehlersuche und könnte ebenfalls über die Schnittstelle der Middleware geliefert werden. Auf Anwendungsebene entstehen somit in Summe 12 Byte Nutzdaten.

Zeit Nachricht Anwendung	$\tau_{\mathrm{PA}} = 12 \cdot (\tau_B + \tau_{\mathrm{SPI}})$	$= 400.8\,\mu\mathrm{s}$
Zeit Nachricht Middleware (ohne Payload Anwendung)	$\tau_{\mathrm{PM}} = 22 \cdot (\tau_B + \tau_{\mathrm{SPI}})$	$= 734.7\ \mu\mathrm{s}$
Zeit Längenbyte	$1 \cdot (\tau_B + \tau_{\mathrm{SPI}})$	$= 33.4\,\mu\mathrm{s}$
Transceiver (Daten)	$\tau_{\mathrm{Pre}} + \tau_{\mathrm{Rl}}$	$= 224\,\mu\mathrm{s}$
Transceiver Umschaltzeit	τ_U	$= 192\,\mu\mathrm{s}$
Σ	τ_{TxFrame}	$= 1584.9\,\mu\mathrm{s}$

Tabelle 5.5.: Übertragungszeit eines Datenrahmens

Übertragungszeit eines Datenrahmens Die gesamte Übertragungszeit für einen Datenrahmen ist in Tabelle 5.5 dargestellt und ergibt sich aus der Summe der Daten für Anwendung, Middleware und dem Längenbyte. Jedes Byte wird dabei zum einen über den SPI-Bus transferiert und danach erst vom Transceiver übertragen. Mit Hilfe der berechneten Werte aus Tabelle 5.2 kann die Zeit für die reine Datenübertragung mit $1584.9\,\mu$s berechnet werden. Die minimale untere Schranke für die Kommunikation ist somit mit $1584.9\,\mu$s gegeben und kann für alle weiteren Abschätzungen verwendet werden. Diese Zeit wird in der Praxis jedoch nicht zu erreichen sein, da hier nicht die Zeit für die Ansteuerung des Transceivers berücksichtigt ist und ebenfalls davon ausgegangen ist, daß alle Busse zu jedem Zeitpunkt sofort verfügbar sind. Für einen Mikroslot wurde deshalb die Slotlänge zunächst auf 4 ms festgelegt.

5.3.2. Synchronisation

Bei TDMA-Verfahren müssen alle Knoten regelmäßig synchronisiert werden, um die Abweichungen der internen Uhren anzugleichen, damit alle an der Kommunikation beteiligten Knoten den Mikroslotanfang treffen. Ein verspäteter Sendebeginn resultiert darin, daß die Übertragung bis in den nächsten Mikroslot reicht und damit die Übertragung dort zerstört. Knoten, die nicht an einer Übertragung beteiligt sind, können in der Zwischenzeit ihren Transceiver deaktivieren, müssen jedoch wieder rechtzeitig empfangsbereit sein, wofür auch sie eine exakte Zeit benötigen. Da die Uhren aller Knoten Gangungenauigkeiten besitzen, müssen die Uhren von allen Knoten in regelmäßigen Abständen neu synchronisiert werden. Zu diesem Zweck wird im Vorfeld ein Knoten als Zeitgeber bestimmt. Dieser sendet regelmäßig innerhalb des hierfür reservierten Mikroslots einen Synchronisations-Rahmen über das Medium. Alle Knoten verwenden den möglichst genauen Empfangszeitpunkt des Rahmens als neuen Referenzzeitpunkt. Die Verzögerung, die zwischen Senden und Empfang der Daten entsteht, kann entweder von dem Synchronisationsknoten oder den Empfängern ausgeglichen werden.

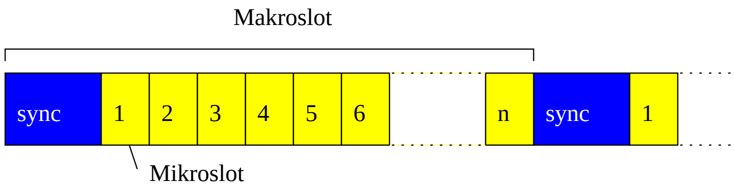


Abbildung 5.11.: Zeitliche Strukturierung des Mediums

Die hierbei entstehende Aufteilung des Mediums ist in Abbildung 5.11 gezeigt. Nach jedem Synchronisationszeitpunkt beginnt die Zählung der Mikroslots wieder von

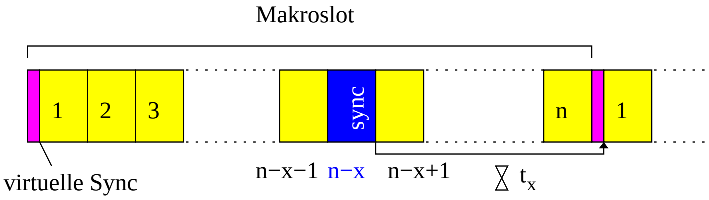


Abbildung 5.12.: Synchronisation in der NCS-CoM

vorne. Als Makroslot wird dabei die Synchronisation und alle Mikroslots bis zum nächsten Synchronisationszeitpunkt bezeichnet. Die Aufteilung eines Makroslots in Mikroslots wird dabei entweder so gewählt, daß jede mögliche Übertragung genau in einen Mikroslot paßt oder bei der Verwendung von mehreren Mikroslots für eine Übertragung möglichst wenig Verschnitt entsteht, das Medium also optimal ausgenutzt werden kann. Die Länge des Makroslots hängt zunächst von der Genauigkeit der verwendeten Uhren ab. Jeder Mikroslot muß aufgrund des Clockoffsets einen Sicherheitsabstand (*Guardtime*) einfügen, damit Übertragungen, die aufgrund von ungenauen Uhren zu früh bzw. zu spät begonnen wurden, nicht die Übertragung in nachfolgenden oder vorherigen Mikroslot zerstören.

Den Synchronisationszeitpunkt zu Beginn eines Makroslots zu stellen und damit vor den ersten Mikroslot, wie in Abbildung 5.11 dargestellt ist, hat entscheidende Nachteile: Nach der Übertragung der Synchronisation muß der darauf folgende Slot bereits synchron zu dem übertragenen Synchronisationssignal sein. Läuft die lokale Zeit eines Knotens schneller als die des Synchronisationsknotens, wird die Zeit „zurückgedreht“ oder angehalten, wodurch es auf dem Knoten zu Problemen in der Ablaufsteuerung kommen kann.[33] Ist die lokale Zeit zu langsam fortgeschritten, wird diese vorgestellt, was für die Abläufe meist weniger problematisch ist. Problematisch ist es jedoch, wenn aufgrund einer zu langsam laufenden Uhr die Daten für den Versand im nächsten Mikroslot noch gar nicht vorliegen und somit dieser Slot dann nicht benutzt werden kann, bzw. dadurch der Synchronisationsslot sehr viel größer gewählt werden muß als nötig. Soll das Synchronisationssignal nicht nur für die MAC-Schicht benutzt werden, sondern ebenfalls zur Synchronisation der Middleware und der Applikation dienen, vergeht ebenfalls Zeit, bis diese Information im System verbreitet ist.

In der NCS-CoM soll der Synchronisationszeitpunkt sowohl für die MAC-Schicht, als auch für die Middleware verwendet werden, weshalb hier nicht der erste Mikroslot des jeweiligen Makroslots zur Synchronisation verwendet wird, wie in Abbildung 5.12 zu

---

[33] Um Probleme bei internen Abläufen zu vermeiden, wird die lokale Zeit nie verändert und nur ein Offset korrigiert.

sehen ist. Der Synchronisationsslot wird dazu x-Slots vor Ende eines Makroslots gesetzt. Jeder Empfänger-Knoten berechnet die Zeit für die verbleibenden x Mikroslots und verteilt diese Information frühzeitig innerhalb des Knotens. Jede Schicht kann nun frühzeitig diese Information verarbeiten und ggfs. auch die Datenbereitstellung für den ersten Mikroslot des nächsten Makroslots darauf anpassen. Die MAC-Schicht setzt sich lediglich einen Timer t_x, der zu Beginn des nächsten Makroslots auslöst und die Nummerierung der Slots beginnt wieder von vorne. Die Mikroslots $n-x+1$ bis n sind von der Synchronisation in Slot $n-x$ nicht betroffen. Durch diese Wahl des Synchronisationsslots ist es nicht nötig, daß der darauf folgende Mikroslot $n-x+1$ bereits synchron zum Synchronisationszeitpunkt ist und der Synchronisationsslot fällt kleiner aus. Vorteil dieser Lösung ist, daß es durch das Synchronisationssignal im System aufgrund der Verteilung des Signals und der evtl. daraus resultierenden verkürzten Zeit für die Datenermittlung nicht zu einer lokalen Lastspitze kommt, die sich nur durch einen längeren Sicherheitsabstand kompensieren ließe. Der von der MAC-Schicht errechnete virtuelle Synchronisationszeitpunkt kann außerdem rechtzeitig in die Anwendung verteilt und dort ebenfalls zur Synchronisation herangezogen werden. Die Middleware mit dem Scheduler für den periodischen Versand kümmert sich selbständig darum, die Zeit anzupassen und die Daten entsprechend früher von der Anwendung anzufordern. Gerade für den periodischen Versand von Daten, der von der Middleware Daten aus der Applikation anfordert, kommt es hier zu weniger Ungenauigkeiten bzw. Ausfällen, wenn die Daten nicht rechtzeitig in die MAC-Schicht weitergereicht wurden.

5.3.3. Slotaufteilung

Die Slotaufteilung des Mediums muß vollständig unter Berücksichtigung der Erfordernisse der Applikation erfolgen und auf diese angepaßt sein. Ebenfalls sind die Kommunikationsslots der Middleware für den Austausch von Daten zu berücksichtigen. Bei der Aufteilung müssen auch die Verarbeitungs- und Weiterleitungszeiten innerhalb des SDL-Systems beachtet werden, da die Ankunft eines Datenrahmens in der MAC-Schicht nicht gleichbedeutend mit der Verarbeitung in der Applikation ist. Für das inverse Pendel wurde aufgrund der Anforderungen der Applikation die Slotaufteilung, wie in Abbildung 5.13 gezeigt, gewählt.

Ein Makroslot wird bei der NCS-CoM zusätzlich in eine feste Anzahl Kommunikationszyklen unterteilt. Ein Kommunikationszyklus entspricht einer reservierten Folge von Mikroslots, die immer gleich aufgebaut sind. In Abbildung 5.13 ist der Kommunikationszyklus der NCS-CoM gezeigt, der mit Slot 1 beginnt und mit Slot 11 endet. Der Kommunikationszyklus beginnt mit dem ersten Slot, der gleichzeitig für Verwaltungsinformationen wie Registrierung, Abonnement und Alive-Signale der

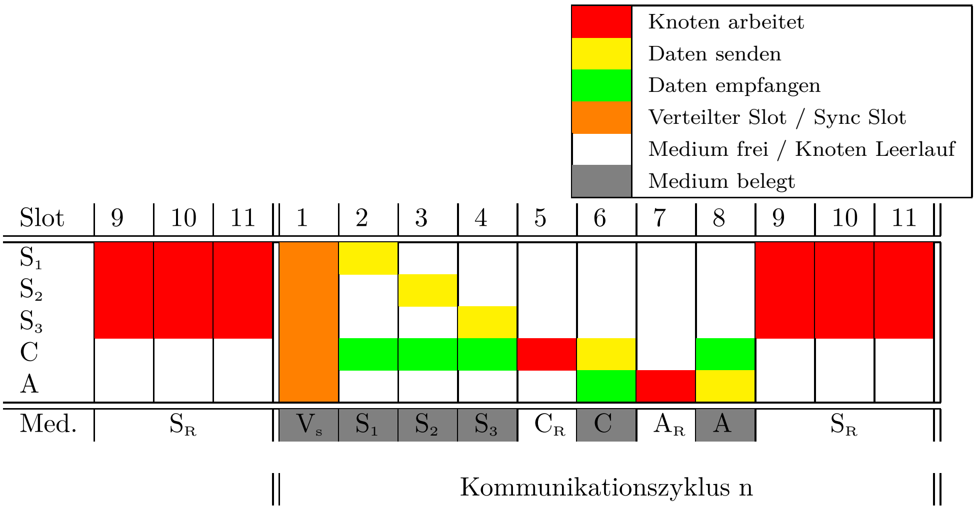


Abbildung 5.13.: Kommunikationszyklus NCS-CoM

Middleware reserviert ist. Damit es in diesem Slot nicht zu einem Wettbewerb und Kollisionen kommt, steht dieser Slot in jedem Kommunikationszyklus einem anderen Knoten exklusiv zur Verfügung. Im letzten Kommunikationszyklus eines Makroslots wird der verteilte Slot zur Synchronisation verwendet. Die Synchronisation wirkt sich jedoch erst, wie im letzten Abschnitt beschrieben, zu Anfang des nächsten Makroslots aus. In den Slots 2 – 4 versenden die Sensor-Knoten S_1, S_2 und S_3 die Daten, die sie synchron in den Slots 9 – 11 des vorherigen Kommunikationszyklus gesammelt und gewandelt haben (S_R), an den Regler-Knoten C. In Slot 5 bleibt das Medium frei, da hier alle empfangenen Sensorwerte von der Regelungsapplikation verarbeitet werden müssen (C_R) und daraus der Stellwert für den Aktuator errechnet wird, der dann in Slot 6 vom Regler an den Aktuator A versendet wird. Der Aktuator benötigt Slot 7 (A_R), um den empfangenen Stellwert in eine Spannung zu verwandeln und damit den Motor anzusteuern. Slot 8 ist für den Aktuator-Knoten reserviert, um beispielsweise Fehler zu signalisieren. Die Sensoren werden aufgrund der Trägheit der Regelstrecke erst in Slot 9 synchron angesprochen. Nach Slot 11 beginnt der Kommunikationszyklus erneut mit dem Verwaltungsslot und dem anschließenden Datenversand der in Slot 9 ermittelten Werte. Die Zykluszeit dieser MAC-Schicht liegt, bei einer Mikroslotzeit von 4 ms, bei $11 \cdot 4\,\text{ms} = 44\,\text{ms}$ und die Gesamtverzögerung bei $9 \cdot 4\,\text{ms} = 36\,\text{ms}$.

Um die Länge eines Makroslots zu bestimmen, muß neben der Ungenauigkeit der Uhr noch die Nebenbedingung für den Verwaltungs- und Synchronisationsslot beachtet werden: Innerhalb eines Makroslots sollte dieser Slot jedem der 5 Knoten mindestens einmal für Verwaltungsinformationen und einem weiteren Knoten einmal zur Synchronisation exklusiv zur Verfügung stehen – somit sind mindestens 6 Kommunikationszyklen pro Makroslot nötig. Für das inverse Pendel ist eine Genauigkeit im Bereich um eine Millisekunde ausreichend genau, für die MAC-Schicht ist eine höhere Genauigkeit erforderlich, um den Sicherheitsabstand zwischen den Slots nicht zu groß werden zu lassen. Für die Pendelsteuerung wurde ein Makroslot mit 35 Kommunikationszyklen (7·Anzahl Knoten) willkürlich festgelegt und der letzte verteilte Slot (374) in diesem Makroslot für die Synchronisation reserviert. Damit beginnt ein neuer Makroslot alle

$$35 \cdot 11 \cdot 4\,\mathrm{ms} = 1,54\,\mathrm{s}.$$

Die zu erwartende Zeitabweichung der Knoten von einer idealen Uhr, läßt sich bei der für Quarze üblichen Ungenauigkeit von ±80 ppm, errechnen und liegt bei

$$1,54\,\mathrm{s} \cdot \pm 80\,\mathrm{ppm} = \pm 123\,\mu\mathrm{s}.$$

Bei einer Slotgröße von 4 ms und einer Übertragungszeit von 1585 μs stellt eine Abweichung der Uhren zwischen zwei Knoten von 246 μs noch kein Problem dar, was sich in der praktischen Evaluation auch später bestätigte. Die komplette Konfiguration der MAC-Schicht für alle Knoten, wie sie für die Pendelregelung verwendet wurde, ist im Anhang A.2 zu finden.

5.4. Energie-Aspekte in der NCS-CoM

Ziel der Pendelregelung ist es, eine hohe Regelgüte und deterministisch zuverlässigen Datentransfer zu erhalten, dennoch sollte ebenfalls der Energieverbrauch möglichst gering gehalten werden. Aus Abbildung 5.13 lassen sich bereits für jeden Knoten Bereiche identifizieren, in denen Teilkomponenten, wie der Transceiver, über die Energiesignalisierung aus Kapitel 3.3 deaktiviert werden können. Für den Winkel-Sensor sind im MSC in Abbildung 5.14 einige Zeitpunkte hervorgehoben, bei denen über die Energiesignalisierung der Stromverbrauch gesenkt werden kann. Tabelle 5.6 listet dazu zu jedem der Zeitpunkte den nötigen Energiezustand der CPU und des Transceivers auf.

Während der Start-Phase, also solange noch keine Dienste abonniert wurden, befindet sich der Knoten im Schlafzustand und aktiviert nur während des Verwaltungsslots seinen Transceiver zum Senden bzw. zum Empfangen, um ihre Dienste zu

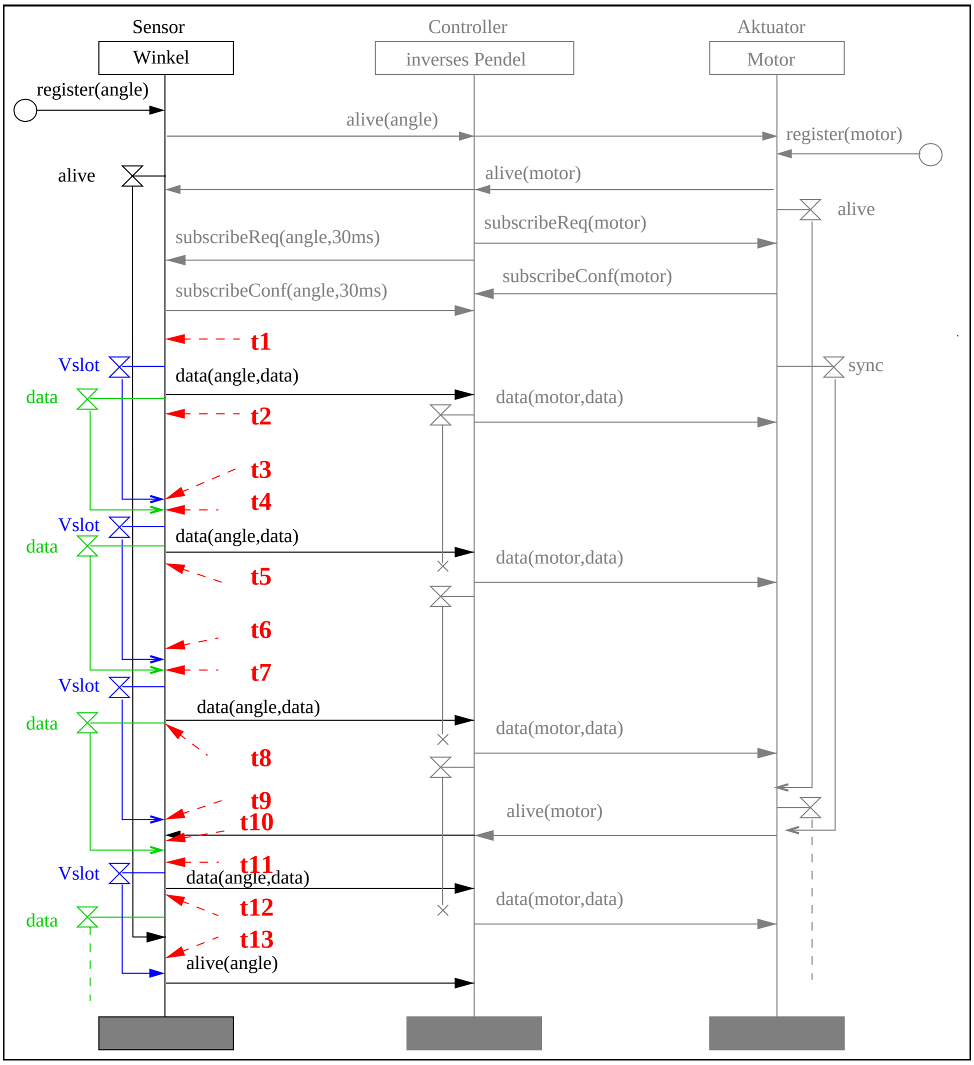


Abbildung 5.14.: MSC des Inversen Pendels

registrieren und andere zu abonnieren. Nach Abschluß der Start-Phase beginnt der normale Kommunikationszyklus (t_1) und der Winkelsensor versendet das erste Mal Daten und versetzt den Transceiver im Anschluß sofort wieder in den *idle* und die CPU in den Schlafzustand (t_2). Läuft der Timer `Vslot` für den nächsten gemeinsamen Kommunikationsslot aus (t_3), wechselt der Transceiver in den Empfangsmodus. Nach Ende des Slots kann der Transceiver zurück in den Schlafzustand wechseln. Mit Ablaufen des Timers `data` (t_4) wird der Sensorwert erfaßt und anschließend mit dem Transceiver versandt. Die Zeitpunkte t_5 – t_7 sind entsprechend den Zuständen t_2 – t_4 gleich. Zum Zeitpunkt t_9 läuft Timer `Vslot` aus und signalisiert damit einen neuen gemeinsamen Mikroslot. Der Transceiver wechselt wieder in den Empfangsmodus und empfängt diesmal Daten und signalisiert dies über einen Interrupt der CPU (t_{10}). Nachdem die Daten vom Transceiver an die CPU übertragen wurden, wechselt der Transceiver zurück in den *idle*-Zustand. Mit erneutem Auslaufen von Timer `Vslot` (t_{13}) sendet auch der Winkelsensor in dem ihm exklusiv zugeteilten Verwaltungsslot V_s sein alive-Signal und wechselt danach zurück in den *idle*-Zustand.

Zeitpunkt	CPU Status	Transceiver Modus
t_1	13 MHz	idle/send
t_2	sleep	idle
t_3	sleep	receive
t_4	104 MHz	idle/send
t_5	sleep	idle
t_6	sleep	receive
t_7	104 MHz	idle/send
t_8	sleep	idle
t_9	sleep	receive
t_{10}	Transceiver-Interrupt, 104 MHz	receive/idle
t_{11}	104 MHz	idle/send
t_{12}	sleep	idle
t_{13}	104 MHz	idle/send

Tabelle 5.6.: Energiemodi der NCS-CoM für Abbildung 5.14

Über die explizite und implizite Energiesignalisierung können nun die in Tabelle 5.6 angegebenen Energiezustände über SDL-Signale modelliert und zeitgenau gesteuert werden. Während die Steuerung des CPU-Schlafmodus in der NCS-CoM ausschließlich über implizite Signalisierung direkt über den SDL-Scheduler erfolgt, erhöht sich der CPU Takt in der Middleware, sobald ein Abonnent für einen auf dem Knoten registrierten Dienst existiert. Die Steuerung des Transceivers hingegen wird direkt in der MAC-Schicht mit Hilfe der statischen Reservierung realisiert. Die sich daraus ergebenden Einsparungen der CPU sind in Tabelle 5.7 für einen Sensor-, den Regler- und den Aktuatorknoten angegeben.

Ressource		Sensor	Regler	Aktuator
Laufzeit		330,14 s	316,68 s	337,10 s
CPU-Zeit	SDL-Kernel	4,8 %	6,7 %	4,8 %
	SDL-Agenten	14,3 %	27,0 %	10,9 %
	SDL-Umgebung	2,5 %	2,5 %	2,9 %
	HW-Interrupt	1,6 %	1,8 %	1,5 %
	Energiesparen	**76,8 %**	**62,2 %**	**80,0 %**
RAM	max. Reserviert	102.732 Byte	115.076 Byte	102.860 Byte
	in Benutzung	100.580 Byte	109.516 Byte	100.332 Byte
	Auslastung	39,2 %	43,9 %	39,2 %
SDL-Signale	Gesamt	151.832	266.900	154.946
	Frequenz	459,9 Sig/s	842,8 Sig/s	459,7 Sig/s
SDL-Cache	Treffer	223.040	349.984	227.704
	Fehler	5.807	16.596	5.900
	Effizienz	97,4 %	95,2 %	97,4 %

Tabelle 5.7.: Aufbereitete Status-Ausgabe der NCS-CoM nach einer Laufzeit von 5 Minuten bei Abtastung alle 60 ms, CPU max. 104 MHz

In den Laufzeitinformationen zeigt sich die geringste Einsparung (62 %) beim Regler-Knoten und die höchste (80 %) beim Aktuator-Knoten. Im Vergleich zu Sensor und Aktuator muß der Regler-Knoten alle Sensorwerte empfangen, diese an die Applikation weiterleiten und einen neuen Stellwert berechnen. Trotz der hohen Anforderungen an die Regelungsperformance ist keiner der Knoten mehr als 38 % der Zeit aktiv und spart so mit der impliziten Energiesignalisierung Strom ein.

5.5. Fazit

Das Ziel, über den modellgetriebenen Ansatz mit Hilfe von SDL ein Kommunikationssystem und eine Middleware für ein schwieriges Regelungssystem durchzuführen ist gelungen, und konnte in einer praktischen Evaluation ebenfalls gezeigt werden. Zunächst wurden dazu die Anforderungen des Regelungssystems an das Kommunikationssystem bestimmt und dann die Entwicklung in mehreren Phasen durchgeführt. War zunächst ein einzelner Knoten für die komplette Regelung verantwortlich, wurde dies im Verlauf auf immer mehr Knoten verteilt und der Zugriff generisch über eine dienstorientierte Middleware abgebildet. Die speziell für diesen Anwendungsfall entwickelte MAC-Schicht garantiert jedem Dienst die exklusive Nutzung des Mediums in seinem Kommunikationsslot und stellt gleichzeitig eine Synchronisation bereit, die auf allen Ebenen bis in die Anwendung genutzt werden kann. Durch die exklusive Reservierung können Verbraucher mittels expliziter Signalisierung gezielt ein- und ausgeschaltet werden und damit verbunden Energie eingespart werden. Die Entwicklung von zeitkritischen Regelungsaufgaben in einem modellgetriebenen Ansatz in SDL zu entwickeln, stellt somit keinen Widerspruch dar. Die Einsparung von Energie wird durch die Verwendung von SDL ebenfalls begünstigt, da es in vielen Fällen ausreichend ist, bei der Generierung von Kode aus der Spezifikation die implizite Energiesignalisierung zu aktivieren.

Für die Weiterentwicklung des Systems sollte das SDL-System noch genauer nach Ressourcen-Engpässen untersucht werden und weitere Optimierungen an der SDL-Laufzeitumgebung durchgeführt werden. In Folge dessen kann die gewählte Mikroslotzeit verkürzt werden womit sich die Gesamtverzögerung verringert. Die Umsetzung des vierten Entwicklungsschritts und damit die Erweiterung der Ein-Hop-Lösung zu einer Multi-Hop-Lösung erfordert zunächst den Austausch der Synchronisation durch die deterministische Multi-Hop Blackburst-Synchronisation. Im Folgenden steht dann die Erweiterung um eine Routing-Schicht und der Austausch der MAC-Schicht an.

6

Kapitel 6.

Werkzeuge & Hardware

Ohne geeignete Werkzeuge und deren volle Kontrolle wären viele der in den vorherigen Kapiteln vorgeschlagenen Erweiterungen an SDL nicht möglich gewesen. Erweiterungen des Schedulers für die Realisierung von Jitter (Kapitel 3.4), die Echtzeitsignalisierung (Kapitel 4.2) und schließlich die Regelung des inversen Pendels (Kapitel 5) haben Eingriffe am Parser, Generator, der Laufzeitumgebung und den Hardwaretreibern nötig gemacht. Für die Sprache SDL lagen alle wichtigen Werkzeuge wie Transpiler (*ConTraST* [35]), Laufzeitumgebung (*SdlRE* [36]) und die Hardwareabstraktion (*SEnF* [35]) als Eigenentwicklungen der AG Vernetzte Systeme im Quellkode vor. Neben den bereits vorgestellten wurden ebenfalls Optimierungen und Fehlerbehebungen vorgenommen, die in Kapitel 6.1 beschrieben sind.

Neben den Erweiterungen an bestehenden Werkzeugen ist auch die SDL-Konfigurationsschnittstelle (Kapitel 6.2) entstanden, die es erlaubt, Einstellungen an bestehenden Programmen vorzunehmen, ohne diese neu zu übersetzen. Erst die Entwicklung eines neuen Imote2-Bootloaders [32, 63] (Kapitel 6.4) und die Implementierung von Fehlerbehandlungsroutinen haben die Imote2-Plattform für größere Systeme benutzbar gemacht.

Doch nicht nur Software ist nötig, um eingebettete Systeme erforschen, entwickeln und anwenden zu können. Für die Energiemessungen aus Kapitel 3.2 und die Überwachung der eigenen Akkuspannung des MICAz mußte eine Hardware-Erweiterung gefertigt werden. Für den Einsatz des Imote2 zur Regelung des inversen Pendels und im *AmSys*-Projekt [102] fehlten dem Imote2 analoge Schnittstellen und der Zugriff auf digitale Ein- und Ausgänge, weshalb eigens ein selbstentwickeltes Sensorboard zum Einsatz kommt. Alle Hardware-Erweiterungen, die für diese Arbeit nötig waren, sind in Kapitel 6.3 zusammengefaßt.

6.1. Die SDL-Entwicklungswerkzeuge

ConTraST (*Configurable SDL Transpiler*) und die Laufzeitumgebung SdlRE (*SDL Runtime Environment*) [36] sind im Rahmen einer Diplomarbeit von C. Weber [110] entstanden und danach kontinuierlich weiterentwickelt worden. Der Transpiler ConTraST verarbeitet SDL-PR-Kode,[34] der von einem grafischen Editor generiert oder von Hand geschrieben werden kann, und erzeugt daraus C++-Klassen. Jede Klasse repräsentiert entweder einen SDL-Datentyp, ein SDL-Signal oder einen SDL-Zustandsautomaten, wie er in dem SDL-System, den SDL-Blöcken oder den SDL-Prozessen enthalten ist. Die Laufzeitumgebung SdlRE instantiiert die Klassen, erzeugt somit das SDL-System und stellt zusätzlich allgemeine Routinen zur Signalverarbeitung/-zustellung, Bearbeitung von Warteschlangen und zum Scheduling der einzelnen SDL-Agenten bereit. Das *SDL Environment Framework* (SEnF) ist in SdlRE ein eigener SDL-Agent und ist für die Anbindung von SDL an externe Ressourcen, wie Betriebssystemaufrufe, Anbindung an andere Anwendung oder den Hardware-Zugriff zuständig.

6.1.1. ConTraST

6.1.1.1. Entstehung

Die Sprache SDL [55] hat seit 1992 und dem Standard SDL'92 erste objektorientierte Eigenschaften erhalten. SDL'96 behebt hauptsächlich Fehler und Inkonsistenzen von SDL'92. Mit Einführung von SDL-2000 wurde zusätzlich eine formale Semantik mit *Abstrakten Zustandsautomaten* (ASM) definiert, die das Laufzeitverhalten von SDL spezifiziert. Die letzte Änderung hat die Sprache 2010 mit dem Standard SDL-2010 erfahren, mit dem einige Erweiterung von SDL-2000 wegen zu hoher Komplexität wieder entfernt wurden. Aufgrund der fehlenden Werkzeugunterstützung von SDL-2000 bzw. SDL-2010 werden diese Standards im folgenden nicht weiter beachtet.

In SDL erfolgt die Spezifikation des Verhaltens in den SDL-Prozessen. Jeder SDL-Prozess stellt einen kommunizierenden Automaten im SDL-System dar, der über Kanäle mit den anderen SDL-Prozessen im System Signale austauschen kann. Die Transformation von Automaten in objektorientierte Sprachen wie C++ läßt sich leicht auf Klassen, Methoden und Variablen abbilden. Während die formale Spezifikation von Hand in der Laufzeitumgebung (SdlRE) implementiert wurde, wurde zur

---

[34] SDL-PR-Kode ist die textuelle Repräsentation von SDL. Fast jedes SDL-Werkzeug kann diese Repräsentation erstellen und viele verwenden sie als Zwischensprache bei der Kodegenerierung.

Transformation der Automaten der SDL-Spezifikation ein Transpiler (ConTraST) entwickelt, der lesbaren, objektorientierten Kode erzeugt. Die spezifizierten Datenstrukturen und Signale werden ebenfalls auf Klassenstrukturen abgebildet und von ConTraST automatisch generiert. Die Lesbarkeit und Nachverfolgbarkeit des generierten Kodes ist gerade dann wichtig, wenn in einem System ein Fehler vorliegt und die Ursache mit einem Debugger eingegrenzt werden muß. Ausführliche Beispiele für die Generierung von Klassen aus SDL-PR mittels ConTraST sind in der Dissertation von I. Fliege [33] zu finden. Durch ConTraST ist es möglich, Erweiterungen an der Sprache SDL zu definieren und diese zu testen. Um die Unterstützung der grafischen Werkzeuge, die diese neuen Sprachmittel (noch) nicht kennen, nicht zu verlieren, behilft man sich meist mit Annotationen, d.h. „qualifizierten“ Kommentaren, die vom grafischen Werkzeug ignoriert, von ConTraST jedoch verarbeitet werden.

6.1.1.2. Optimierungen

Doch nicht nur Strukturen und Automaten müssen von SDL-PR in C++-Kode umgewandelt werden, auch neue Datentypen, Signale und deren Signalwege müssen generiert werden. Viele Funktionen, die in SDL natürlich und leichtgewichtig wirken, stellen sich bei genauerer Analyse als sehr aufwändig dar, wie beispielsweise der Signalversand oder das Ausführen von Prozeduren. Um die Laufzeiten von SDL zu veringern, konnten einige Optimierungen bereits auf den Zeitpunkt der Kodegenerierung verlagert werden, ohne dabei die Semantik der Sprache zu verletzen:

Signalrouten Im SDL-Standard wird für das Finden eines Signalweges das Prädikat `applicable` auf jeden voll expandierten Pfad angewandt, bis eine gültige Route gefunden wurde. Allein die Tiefensuche über alle Pfade ist zur Laufzeit aufwändig. Zusätzlich müssen in jedem Expansionsschritt viele Bedingungen erfüllt sein (siehe Z100F3 [56], Seite 14), wie die Existenz des Signals explizit oder innerhalb einer Signalliste auf jedem Teilabschnitt. Durch die Verwendung von `via` kann die Wahl der Signalroute zusätzlich eingeschränkt werden. Das Signal wird anschließend nicht direkt in die Signalwarteschlange des Agenten für den SDL-Prozess eingefügt, sondern dem Agenten (SDL-Agentset) des umgebenden Blocks. Erst bei der Ausführung des Agenten für den SDL-Block, wird das Signal sukzessive über bestehende Links weitergeleitet und schließlich in die Warteschlange des SDL-Agenten eingefügt. Nach der Generierung eines SDL Systems können sich Signalrouten bis zum umgebenden Block nicht mehr ändern und, sofern kein dynamisches Erzeugen von Instanzen verwendet wird, auch nicht die Routen zu den SDL-Prozessen.

Die Berechnung der Signalrouten kann in vielen Fällen statisch zum Zeitpunkt des Transpilerlaufs oder beim Start des Systems erfolgen. Nur in wenigen Fällen ist eine Ermittlung dynamisch zur Laufzeit notwendig, sodaß diese Information in Caches abgelegt werden kann. Für den Transpiler waren Erweiterungen notwendig, damit alle Informationen, die zur Laufzeit zur Verfügung stehen, ebenfalls während seiner Ausführung zur Verfügung stehen. Der Transpiler benötigt diese Informationen, um darauf vorab eine Signalwegeermittlung durchführen zu können.

Prozeduren SDL-Prozeduren sind vergleichbar mit Funktionsaufrufen in anderen Programmiersprachen. Bei ihrem Aufruf wird der Programmfluß unterbrochen, eine Teilfunktionalität berechnet und danach der Programmfluß dort fortgesetzt, wo er zuvor unterbrochen wurde. In einer prozeduralen oder objektorientierten Sprache ist die Zeit für den Sprung/Rücksprung sowie das Sichern und Wiederherstellen des Zustands gering oder wird sogar anschließend vom Compiler eliminiert.[35] Funktionen bzw. Prozeduren werden einerseits verwendet, um große Programmteile logisch zu strukturieren und damit übersichtlicher zu machen, andererseits auch, um wiederkehrende oder ähnliche Programmabschnitte generisch und einmalig vorzuhalten. In SDL kann man zwischen einfachen und komplexen Prozeduren unterscheiden: Einfache Prozeduren bestehen aus einer oder mehreren Anweisungen, enthalten aber keine SDL-Zustände. Komplexe Prozeduren sind entweder externe Prozeduren (zur Anbindung von nativem Kode) oder Prozeduren, die SDL-Zustände enthalten. Die SDL-Spezifikation sieht vor, daß bei jedem Prozedur-Aufruf der Zustandsgraph um alle Anweisungen, Transitionen und Zustände der Prozedur sowie alle darin enthaltenen Variablen erweitert wird. Bei Verlassen der Prozedur wird der ursprüngliche Zustandsgraph durch Entfernen aller eingefügter Elemente wiederhergestellt. Alle Variablen, die nicht explizit als Parameter mit `IN/OUT OUT` angeben sind, müssen nach der Rückkehr der Prozedur wieder ihren ursprünglichen Wert enthalten.

Bereits zur Ausführungszeit des Transpilers läßt sich überprüfen, ob eine Prozedur einfach oder komplex ist. Handelt es sich um eine einfache Prozedur, läßt sich diese auf einen C++-Funktionsaufruf abbilden, wodurch die aufwändige Sicherung des Kontextes dem C++-Compiler übertragen wird. Da sich alle Anweisungen in der Funktion kapseln lassen, ist eine Veränderung des Zustandsgraphen unnötig. Die Unterscheidung, ob Variablenwerte erhalten bleiben oder nicht, läßt sich ebenfalls bei der Übergabe der Parameter als Referenz oder als Wert steuern. Da vielfach Prozeduren nur dazu eingesetzt werden, um einen SDL-Prozess zu strukturieren oder wiederkehrende Funktionalität auszulagern, werden einfache Prozeduren bevorzugt eingesetzt.

---

[35] In C/C++ läßt sich dies explizit durch einfügen des Schlüsselwortes `inline` ausdrücken, und erlaubt damit dem Compiler die Funktion, sofern er kann, anstelle des Funktionsaufrufs direkt einzubetten

Durch die Erkennung bereits während des Transpilerlaufs entsteht kein zusätzlicher Aufwand zur Ausführungszeit. Während der Ausführung profitiert jede einfache Prozedur von der schnelleren Ausführung bei gleichbleibender SDL-Semantik. Die Auslagerung von Funktionalität aus der Spezifikation als einfache Prozeduren hat somit nur sehr geringe, in der Regel zu vernachlässigende, Auswirkungen auf die Laufzeit.

Signalverarbeitung im SDL-Prozess Wurde ein SDL-Agent eines Prozesses mit Signalen in seiner Eingangswarteschlange zur Ausführung durch den Scheduler ausgewählt, werden in diesem Prozess alle ausgehenden Transitionen des aktuellen Zustand ermittelt.[36] Zunächst werden nur Transitionen mit priorisiertem Signalempfang berücksichtigt. Unter Berücksichtigung von `Save`-Anweisungen wird geprüft, ob eine der Transitionen das erste Signal in der Eingangswarteschlange empfangen kann. Wurde keine Transition gefunden, wird die Liste aller Transitionen nicht mehr auf priorisierten Empfang eingeschränkt und die Bedingung erneut geprüft. Wurde auch hier kein Signal gefunden, wird das erste Signal in der Eingangswarteschlange verworfen[37]. Sofern noch Signale in der Eingangswarteschlange enthalten sind, beginnt die Suche erneut mit dem priorisierten Empfang. Sind keine Signale mehr vorhanden, werden alle Transitionen geprüft, die ein sogenanntes *Continuous Signal* empfangen. Ein Continuous Signal ist eine Bedingung, die erfüllt sein muß, damit eine Transition ausgeführt wird. Enthält ein SDL-Prozess weder einen priorisierten Signalempfang noch ein Continuous-Signal, entsteht trotzdem in jedem Prozess-Agenten zunächst der Suchaufwand für die entsprechenden Transitionen.

Auch hier kann der Transpiler bereits statisch entscheiden, ob ein SDL-Prozess, unter Berücksichtigung von Prozeduraufrufen, überhaupt einen priorisierten Signalempfang oder Continuous Signale enthält. Ist dies nicht der Fall, kann diese Information der Laufzeitumgebung beim Start des Agenten für des SDL-Prozesses mitgeteilt werden, sodaß die Überprüfung dieser Signalarten entfallen kann. Um komplexe SDL-Systeme weiter zu beschleunigen, wäre es denkbar, bereits alle Zustände berechnen zu lassen, in denen der priorisierte Empfang möglich ist und nur in diesen die Prüfung auf priorisierten Empfang zu aktivieren. Entsprechendes gilt für das Continuous Signal.

---

[36] Der *-Zustand ist eine syntaktische Abkürzung und wird bereits durch den Transpiler aufgelöst

[37] Sofern dieses nicht durch eine Save-Anweisung gespeichert wurde, sonst das nächste Signal der Liste

6.1.2. SdlRE (SDL Runtime Environment)

6.1.2.1. Entstehung

Während ConTraST das spezifizierte Verhalten der Automaten in C++-Kode umwandelt, wird eine Laufzeitumgebung (*SDL-Virtual-Machine*) benötigt, die für die Initialisierung und Ausführung des Systems und aller Komponenten verantwortlich ist. Zusätzlich stellt sie auch Signalwarteschlangen für jeden Agenten zur Verfügung, leitet Signale an die Agenten der Empfängerprozesse weiter, wählt Transitionen aus und implementiert ein Scheduling zur fairen Auswahl aller SDL-Agenten im System. Bis auf wenige Ausnahmen und Implementierungsdetails entspricht SdlRE der formalen ASM-Definition der SDL-Semantik. Zusätzlich dazu implementiert SdlRE die Basisdatentypen von SDL.

6.1.2.2. Optimierungen

Sprünge Da die formale Semantik auf ASMs basiert und in ASMs nach jedem Schritt ein konsistenter Änderungssatz auf den Automaten angewandt wird, erfolgen alle Änderungen über Zustände und viele Ein- und Rücksprünge. Im Gegensatz zum theoretischen Modell benötigen die vielen Sprünge Laufzeit und die Speicherung der Zustände zusätzlich Speicher. Nach einem Zustandswechsel der ASM müssen teilweise Listen, die beispielsweise für die Transitionsauswahl benötigt werden, mit gleichem Inhalt erneut aufgebaut werden. Der Einsatz von Schleifen, persistenten Listen und Caches konnte hier das Laufzeitverhalten und die Transitionsauswahl, bei der viele dieser Sprünge vorkommen, deutlich verbessern.

Mengenoperationen Operationen auf Mengen und die Auswahl von Teilmengen über Prädikate werden in der SDL-Semantik exzessiv verwendet (vergleiche beispielsweise die Transitionsauswahl). Die Implementierung dieser Funktionen in einer Programmiersprache wie C++ ist aufwändig und benötigt viel Zeit und Speicher. Im ersten Schritt wurden viele der Mengen zunächst in Caches zwischengespeichert und danach sukzessive durch effizientere Implementierungen ausgetauscht. Dieser Vorgang ist in SdlRE für die Agenten der Prozesse bereits erledigt – da es sich um einen sehr fehleranfälligen Vorgang handelt, jedoch noch nicht an allen Stellen fertiggestellt.

6.1.2.3. Fehlerbehebung/-sicherung

Eine der wichtigsten Änderungen an der Laufzeitumgebung war es, Fehler der alten Implementierung zu finden, zu beheben und sicher zu stellen, daß diese Fehler nicht erneut auftreten. Die Größe der Laufzeitumgebung inklusive der Datenstrukturen von mehr als 25.000 Quellkodezeilen lassen vermuten, daß hier auch einige Fehler in dem Kode stecken, die erst gefunden werden müssen.

gcc Bei der Übersetzung von SDL-Systemen zu ausführbaren Dateien kommt nach ConTraST der C++-Compiler (gcc/g++) der GNU Compiler Collection zum Einsatz. Vielfach unbeachtet ist, daß der Compiler neben Fehlern auch mittels Warnungen auf potenzielle Fehler hinweisen kann. Grund hierfür ist sicherlich, daß bei der Beachtung aller Warnungen mehr und auch im Auge der Entwickler aufwändigerer Quellkode geschrieben werden muß und viele der automatisch generierten Funktionen explizit implementiert werden müssen. Beispiele hierfür sind Standard-Konstruktoren, Kopierkonstruktor und der Zuweisungsoperator. Ohne spezielle Aufrufparameter des Compilers werden zudem keine Warnungen ausgegeben; stattdessen muß man sie einzeln aktivieren. Bei der Überarbeitung von ConTraST und SdlRE war das Ziel, mit möglichst vielen aktiven Parametern keine Warnungen, weder bei der Übersetzung von ConTraST, noch bei der Übersetzung des komplett generierten SDL-Systems durch den C++-Compiler zu erhalten. Am Ende der Umstrukturierung wurden selbst mit Aktivierung vieler Optionen, keine Warnung mehr ausgegeben:

```
−Wall −Weffc++ −Wold−style−cast −Wswitch−default −Winit−self
−Wunused−parameter −Wstrict−overflow=5 −Wpointer−arith
−Wcast−qual −Wcast−align −Wmissing−declarations −Wmissing−field−initializers
```

Unit-Tests Ein Compiler kann bereits viele Fehler und potentielle Fehler aufdecken, nicht jedoch logische Fehler, bei denen der Kode semantisch falsch, syntaktisch aber fehlerfreie ist. Unit-Tests unterstützen bei der Untersuchung einzelner Funktionen oder Operationen auf ihre korrekte Funktionalität und stellen gleichzeitig sicher, daß die Funktionen auch bei späteren Änderungen der Implementierung bei gleicher Eingabe die gleichen Ergebnisse liefern. In SdlRE wurden dazu Testklassen für alle SDL- und ASM-Datentypen hinzugefügt.

Listing 6.1 zeigt einen kleinen gekürzten Ausschnitt aus einem Test für Ganzzahlen, der mit dem *cxxtest*-Framework [24] erstellt wurde. Jede Testsuite besteht aus einer Klasse, die von einer Basisklasse des Frameworks abgeleitet ist. Alle darin enthaltenen Funktionen stellen einen einzelnen Testfall dar, der wiederum aus Einzeltests für die möglichen Operationen zusammengesetzt ist. In der Klasse aus dem

```
include <cxxtest/TestSuite.h>include <asm_datatypes.h>
class asm_datatypeTests:public CxxTest::TestSuite{
  public:
  void testInteger(){
    TS_ASSERT_EQUALS(Integer(2)+Integer(−3),Integer(−1));
    TS_ASSERT_EQUALS(Integer(−2)∗Integer(−5),Integer(10));
    TS_ASSERT_EQUALS(Integer(−10)/Integer(3),Integer(−3));
    TS_ASSERT_EQUALS(Integer(2)<Integer(5),true);
  }
}
```

Listing 6.1: gekürzter Ausschnitt aus einem Testfall für Ganzzahlen

Beispiel werden alle ASM-Datentypen auf Korrektheit getestet und dazu die Klasse `asm_datatypeTests` angelegt. Jeder Datentyp wird in einer einzelnen Funktion im Beispiel `testInteger` überprüft. In dieser Funktion wird jede Operation des Datentyps mit mehreren Eingaben getestet.[38] Ein Lauf des Unit-Test-Werzeugs für die Datentypen sieht wie folgt aus:

```
make
/usr/bin/cxxtestgen −−part −o out/genericadtfixed.cpp src/genericadtfixed.h
cxxtestgen  −−root "−−error−printer" −o out/runner.cpp
cxxtestgen −−part −o out/sdl_datatypesTest.cpp src/sdl_datatypesTest.h
cxxtestgen −−part −o out/asm_datatypesTest.cpp src/asm_datatypesTest.h
g++ −g −o out/runner out/runner.o out/sdlreEssentials.o out/asm_datatypes.o out/
    sdl_datatypes.o out/sdl_coder.o out/sdl_datatypesTest.o out/asm_datatypesTest.o out/
    asm_agentQueueTest.o out/util_cow_cowUtilTest.o out/Set_tcc_tests.o out/
    Octet_string_cow.o out/sdl_coderTest.o
out/runner
Running 58 tests    ................    OK!
```

Blackbox-Test Unit-Tests sind eine gute Basis, um Datentypen und essentielle Operationen auf ihre Funktion zu testen. Komplexe Vorgänge, wie sie in SdlRE bei der Ausführung eines Agenten geschehen, lassen sich damit nur teilweise testen. Das Problem besteht dabei darin, die Ausgangssituation genau so herzustellen, wie es bei vollständiger Initialisierung geschehen wäre. Aus diesem Grund ist in Zusammenarbeit mit M. Winkler ein Testsystem als SDL-Spezifikation entstanden, das Testfälle direkt in einer SDL-Spezifikation definiert. Dieses Testsystem erfaßt dabei sowohl Fehler von ConTraST und SdlRE als auch solche des GNU-Compilers. Abbildung 6.1 zeigt den SDL-Block `TestcaseExecution`, der neben der Ausführungskontrolle 10 Blöcke mit Testfällen enthält. Die Kontrollinstanz führt jeden Testfall einzeln aus und wartet auf dessen Ergebnis. Liefert der Testfall innerhalb einer bestimmten Zeit

---

[38] Im Beispiel ist nur eine Auswahl vorhanden.

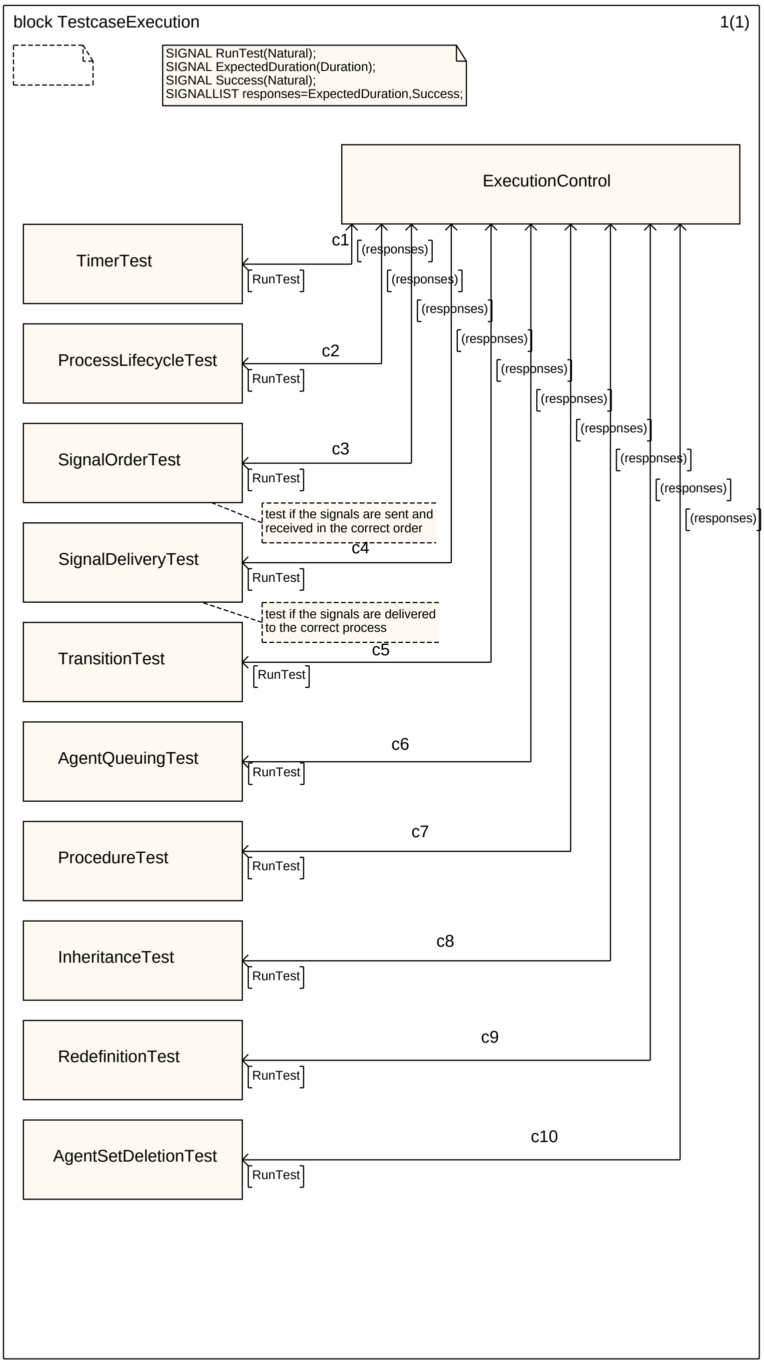


Abbildung 6.1.: System-Test für SdlRE

kein oder ein negatives Ergebnis, gibt die Kontrollinstanz dies als fehlgeschlagenen Test aus. Anhang A.1 zeigt die Bildschirmausgabe eines kompletten Systemtests.

6.1.3. SDL Environment Framework

Die Anbindung von SDL-Systemen an vorhandene Anwendungen, das Betriebssystem oder direkt an die Hardware erfolgt über die „SDL-Umgebung“. Alle Signale, die aus dem SDL-System über Kanäle an die Systemgrenze geführt werden, gelangen in die Umgebung. SdlRE gibt diese SDL-Signale an einen eigenen *Environment*-Agenten weiter, der kompatibel zu den Generatoren aus Tau die Signale an die C-Funktion `xOutEnv` übergibt. Umgekehrt fragt der *Environment*-Agent regelmäßig die Funktion `xInEnv` ab, um Signale aus der Umgebung in das SDL-System zu übertragen. Da diese Funktion außerhalb des SDL-Systems definiert ist, kann jedes Sprachmittel von C/C++ verwendet werden. Über diesen Mechanismus ist es möglich, externe Applikationen anzubinden, Zugriff auf Betriebssystemfunktionen zu erhalten oder auch direkt auf die Hardware zuzugreifen.

Viele der Funktionen einer Umgebung, wozu auch die Ermittlung der aktuellen Systemzeit gehört, werden in jedem Projekt erneut gebraucht. Deshalb wurde von Fliege et al. das S*DL* En*vironment* F*ramework* (SEnF) [35] entwickelt, das für verschiedene Hardwareplattformen und Betriebssysteme Implementierungen zur Ausführung von SDL anbietet. Wird ein SDL-System zusammen mit SEnF übersetzt, entscheidet der Compiler aufgrund von Pre-Compiler-Konstanten, welche Teile des SEnF benötigt werden und bindet nur diese ein. Erfolgt im SDL-System weder der Signalversand noch der Signalempfang an bzw. von einer speziellen Hardwarekomponente, enthält der erzeugte Maschinenkode auch nicht den Treiber aus dem SEnF. Das SEnF ist modular aufgebaut und unterstützt die Betriebssysteme Linux, Windows und auf MICAz [66, 67] und Imote2 [94] den direkten Zugriff ohne Betriebssystem (bare). Zur Simulation von Netzwerken gibt es eine spezielle Anbindung an den Netzwerksimulator ns-2 [68, 108]. Je nach verwendeter Plattform und Betriebssystem werden Funktionen für die Initialisierung der Hardware und Treiber für die Schnittstellen LAN, WLAN, RS-232, CC2420, LED, Bluetooth etc. von SEnF bereitgestellt. Für einige zeitkritische Vorgänge, wie die Ansteuerung des FlexRay-Controllers [19], oder die *Black-Burst-Synchronisierung* (BBS) [48], sind zusätzliche Treiber integriert worden.

Neben der Erweiterung und Pflege bestehender Plattformen wurden die MICAz- und die Imote2-Plattform integriert. Für beide Plattformen wurde zudem eine Möglichkeit benötigt, Laufzeitinformationen anzuzeigen und die Systemauslastung zu ermitteln. Mit Hilfe der Laufzeitinformationen konnten auf einigen Plattformen spezielle

Funktionen zur Energiekontrolle, wie sie in Kapitel 3.3 beschrieben sind, eingeführt werden. Die Einführung der Echtzeitsignalisierung (siehe Kapitel 4.2) hat ihren Ursprung bei den Signalen von und an die Umgebung und stellt ebenfalls eine große Erweiterung dar, die einige Änderungen am Framework erforderlich machte.

6.1.4. Fazit

Die Ergebnisse der vorherigen Kapitel, bei denen die hier beschriebenen Änderungen und Optimierungen zum Einsatz kamen, zeigen, daß sich zeitkritische und energieeffiziente Systeme in SDL spezifizieren lassen. Erst durch diese Optimierungen konnte die Laufzeit des SDL-Systems des inversen Pendels soweit gesenkt werden, daß dieses auch bei größeren Störeinflüssen von außen stabil blieb. Für das dort verwendete SDL-System zeigt Tabelle 5.7 aus Kapitel 5.4 die Laufzeitinformationen, bei denen zum einen eine hohe Cache Trefferrate von 95 % (bei 200.000 Zugriffen) erzielt wurde, zum anderen der Knoten sich in $60 - 80\,\%$ der Zeit im Schlafzustand befindet, also nicht ausgelastet ist. Eine genauere Evaluation der Effizienzsteigerung im Vergleich zur Arbeit von I. Fliege [33] erfolgt hier nicht, da dies nicht Gegenstand dieser Arbeit war. Es sei jedoch bemerkt, daß I. Fliege in seinem Ping-Pong-Benchmark die Transitionsauswahl mit 40 % angegeben hat, weshalb die meisten Optimierungen auch in diesem Bereich erfolgten.

6.2. SDL-Konfigurationsschnittstelle

Konstanten zur Konfiguration werden in Software verwendet, um Einstellungen zentral an bestimmte Gegebenheiten anpassen zu können. Da Konstanten zur Laufzeit nicht verändert werden können, muß hierfür kein veränderlicher Speicher reserviert werden und der Wert der Konstante kann direkt im Quelltext eingebettet werden. In SDL werden Konstanten, im Unterschied zu Variablen, wie in Listing 6.2 gezeigt, definiert. Eingeleitet wird jede Konstantendefinition von dem Schlüsselwort `SYNONYM` gefolgt von dem Namen und dem Typ, sowie der einmaligen Zuweisung des Wertes. Müssen diese Konstanten aufgrund geänderter Bedingungen angepaßt werden, ist es nötig, das SDL-System erneut komplett zu übersetzen und auf die Hardware aufzuspielen. Dieser Prozess ist, abhängig von der Systemgröße, aufwändig und erfordert die SDL-Spezifikation sowie alle Tools zur Übersetzung für die Zielplattform. Wird die Konfiguration von der Spezifikation bzw. dem Kompilat getrennt, können Parameter einfach und schnell angepaßt werden, ohne eine Neuübersetzung des gesamten Systems. Es ist somit möglich, initial alle Systeme mit dem gleichen SDL-System zu bespielen und eine Differenzierung der Funktion ausschließlich über die spätere Konfiguration zu erledigen.

```
SYNONYM period Integer=60;
SYNONYM macroSlotLength Integer=7*period;
```

Listing 6.2: Einfache Konstantendefinition in SDL

Für SDL-Systeme, die in Simulationsumgebungen auf PC-Hardware eingesetzt werden, wurde von A. Geraldy [41] eine Lösung vorgestellt. Hierbei werden Einstellungen in einer Text-Datei als Schlüssel-Wert-Paare gespeichert und Operatoren in einem SDL-Package zur Verfügung gestellt, die mit Hilfe des SEnF [35] die Daten ausliest. Dieses Vorgehen hat verschiedene Nachteile:

- kein Einbinden mehrerer Konfigurationen (Wiederverwendung) - Konflikt im Namensraum
- keine Typsicherheit in der Konfigurationsdatei
- keine Wertebereichsprüfungen (Assertions)
- schlecht portierbar auf Plattformen ohne Dateisystem
- durch statische Textdatei keine Parametrierung der Einstellungen
- Text muß zur Laufzeit geparst werden (aufwändiger „Lookup“)
- bei vielen Einträgen sehr große Dateien

6.2.1. Plattformunabhängige Konfigurationen

Um diese Nachteile zu verringern, wurde die Konfigurationsdatei in ein plattformunabhängiges binäres Dateiformat geändert. Gerade auf einer eingebetteten Plattformen spielt Zeit- und Speicherverbrauch eine große Rolle und der Aufwand beim Zugriff auf eine Konfigurationsvariable, den Text der Konfiguration zu parsen, würde unnötig Ressourcen verbrauchen. Auf eingebetteten Plattformen wie dem Imote2, die ohne Dateisystem betrieben werden, steht ein spezieller Speicherbereich zur Verfügung, in den die Konfiguration geschrieben wird. Um einen schnellen Zugriff auf die Daten in der Datei zu gewährleisten, wurde auf textuelle Bezeichner aufgrund ihrer unterschiedlichen Länge verzichtet, und stattdessen eine kompakte Indexstruktur eingefügt.

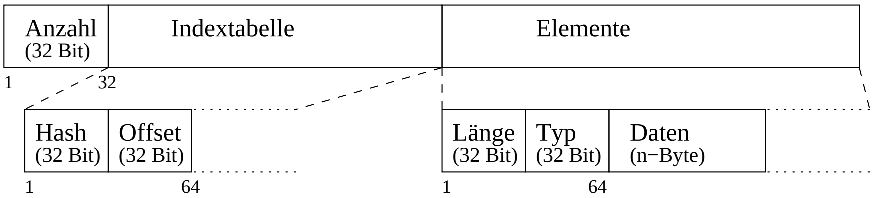


Abbildung 6.2.: Aufbau der binären Konfigurationsdatei

Im Detail sieht der Aufbau der binären Konfigurationsdatei wie in Abbildung 6.2 gezeigt aus. Das erste 32 Bit breite Feld gibt die Anzahl der Konstanten (Elemente) in der Datei an. Als nächstes folgt eine Indexstruktur, die pro Konstante einen Eintrag enthält, der aus einem eindeutigen 32 Bit Hash-Wert und der absoluten Position des Elements innerhalb der Konfigurationsdatei besteht. Dabei wird der Name einer Konstanten im Vorfeld durch einen eindeutigen Hash-Wert ersetzt. Die Indextabelle ist aufsteigend nach dem Hash-Wert sortiert, wodurch sich ein Element mittels binärer Suche sehr schnell finden läßt.[39] Anschließend an die Indextabelle folgen die Elemente, die jeweils aus einem Längen- und einem Typfeld sowie dem Konfigurationswert der angegebenen Länge besteht.

Bei der Erstellung der binären Konfigurationsdatei werden die Namen der Konstanten in einen Hash-Wert umgerechnet und für SDL eine Datei mit Konstanten (`SYN_ONYM`) für den einfachen Zugriff erstellt. Die Verwendung von Hash-Werten anstelle eines inkrementierenden Wertes hat den Vorteil, daß fehlende Werte, genauso wie eine falsche Konfigurationsdatei, erkannt werden können. Der Aufwand für das Parsen und die Erstellung der Konfiguration verlagert sich auf den Zeitpunkt der Erstellung. Um das Erstellen einer Konfigurationsdatei möglichst einfach zu gestalten, wird die Konfiguration als Java-Programm geschrieben. Dies erlaubt neben der Definition von reinen Konstanten auch Berechnungen, Ein- und Ausgaben und Fehlerprüfungen. Als Konstanten einer Konfiguration werden aus dem Programm alle nicht als `private` deklarierten Klassenvariablen benutzt. Innerhalb des Java-Programms sind alle Sprachmittel einschließlich Vererbung erlaubt, wodurch eine Konfiguration sehr flexibel wird.

Die Basisklasse für die Konfiguration des Reglers der NCS-CoM ist in Listing 6.3 gezeigt und besteht lediglich aus drei Teilen: Zum einen aus der Konfiguration der

---

[39] Die Position in der Tabelle wird nicht anhand des Hash-Wertes, wie sonst bei Hash-Tabellen üblich, ermittelt.

```
public class CONTROLLERConfig{
  MacConfig mac=new MacConfig(nodeRoles.CONTROLLER,true);
  ServiceUserConfig user=new ServiceUserConfig();
  ServiceProviderConfig provider=new ServiceProviderConfig();
}
```

Listing 6.3: Konfiguration des Reglers (Auszug)

MAC-Schicht und zwei Konfigurationen für die Middleware, einmal als Dienstnutzer und einmal als Diensterbringer. Der Konstruktor-Aufruf der MAC-Schicht besitzt zwei Parameter, von denen der erste bestimmt, um welchen Knotentyp es sich handelt und daraufhin die für ihn reservierten Mikroslots auswählt. Der zweite Parameter gibt an, ob dieser Knoten die Synchronisierung des Mediums übernimmt oder nicht.

```
1  class MacConfig{
2    final int period = 60;
3    private Integer nodeRole=null;
4    int nextFreeSlot=nodeRoles.getCount()*period;
5    Boolean master=null;
6    ...
7
8    MacConfig(Integer nodeRole, Boolean master){
9      this.master=master;
10     ...
11     assert (nodeRoles.getFirstFreeSlot()*txWindow+txWindow)<period:"Error␣in␣period";
12     ...
13   }
14 }
```

Listing 6.4: Konfiguration des MAC Layers (Auszug)

Einen Auszug der Konfiguration der MAC-Schicht zeigt Listing 6.4; die vollständige Konfiguration für die NCS-CoM ist in Anhang A.2 zu finden. Die Klasse `MacConfig`, besteht aus Variablen, die entweder konstant (wie `period`) oder berechnet sind (`nextFreeSlot`). Jede in der Klasse deklarierte geschützte (`protected`) oder öffentliche (`public`) Variable oder Konstante (`final`) wird in die binäre Konfigurationsdatei übernommen. Der Konstruktor der MAC-Schicht setzt auf der Grundlage der ihm übergebenen Parameter selbst Klassenvariablen. Da die Konfiguration als Programm formuliert ist, ist es ebenfalls möglich, die Prüfung der resultierenden Konfiguration mittels `assert` zu realisieren. Würde in die Pendelsteuerung beispielsweise ein neuer Knoten integriert, ihm ein Slot zugewiesen ohne die Zykluszeit (`period`) anzupassen, wäre das Ergebnis der Berechnung (Zeile 11) größer als die Periodendauer, die Annahme verletzt, wodurch die Erstellung der Konfiguration mit einem Fehler beendet wird.

```
$ javac ∗.java
$ java −cp .:/mnt/vstools/SEnF/tools/config/dist/config.jar config.Main −v
    CONTROLLERConfig CONTROLLER
Reading Configuration
  Reading Path CONTROLLERConfig
  Reading Path CONTROLLERConfig −> MacConfig (mac)
  Reading Path CONTROLLERConfig −> ServiceUserConfig (user)
  Reading Path CONTROLLERConfig −> ServiceProviderConfig (provider)
Writing PR File CONTROLLER.pr
Writing Config File CONTROLLER.cfg
  Sorting Configuration for Lookup Table
   Serializing items
  Writing Lookup Table
  Writing Configuration Records
BOOL mac_master[−1102791854] = true
INT mac_period[−1013250383] = 60
INT mac_nextFreeSlot[1489684365] = 360
```

Listing 6.5: Programmlauf für die Erzeugung der Regler-Konfiguration

Ist die Konfiguration als Java-Quellkode vollständig, übersetzt der Java-Compiler die Konfiguration in .class-Dateien. Der Java-Compiler übernimmt die meisten Typprüfungen und testet auf Konsistenz innerhalb der Konfiguration. Die Erzeugung der binären Konfiguration erfolgt schließlich mittels Instantiierung und Java-Reflections. Ein hierfür entwickeltes Java-Programm nimmt eine Klassendatei als Ausgangspunkt der Konfiguration entgegen und instantiiert diese sowie alle dafür benötigten Subklassen. Zu diesem Zeitpunkt werden alle nötigen Konstruktoren aufgerufen und die Wertprüfung mittels Assertions ausgeführt. Das Programm liest nun mittels Java-Reflection alle nicht privaten Variablen der Klassen unter Berücksichtigung von Vererbung, aus. Die Namen der Konfigurationsvariablen werden dabei hierarchisch mit einem Präfix, der sich aus den Objektnamen zusammensetzt, gebildet. Da in Listing 6.3 das Objekt der Klasse `MacConfig` den Namen `mac` besitzt, erhalten alle Konfigurationsvariablen der Klasse `MacConfig` den Präfix `mac_`.

```
SYNONYM cfg_mac_master Integer=−1102791854;
SYNONYM cfg_mac_period Integer=−1013250383;
SYNONYM cfg_mac_nextFreeSlot Integer=1489684365;
```

Listing 6.6: Hash-Werte der Konfiguration des Reglers in SDL (Auszug)

Listing 6.5 zeigt die Ausgaben eines Programmlaufs zur Erzeugung der binären Konfiguration. Das entwickelte Java-Werkzeug läuft per Reflection, ausgehend von der

Wurzelklasse, alle Objekte ab und sucht nach dort deklarierten Variablen. Für jeden berechneten Variablennamen wird über die Java-Hash-Funktion ein Wert gebildet, der später zum einfachen Auffinden in der binären Datei dient. Da Kollisionen zwischen zwei Hash-Werten nicht auszuschließen sind, wird in diesem Fall ein Fehler ausgegeben und ist nur durch eine Umbenennung einer der Variablen zu lösen.[40] Wird das Programm im *verbose* Modus ausgeführt, zeigt es noch die erkannten Konstanten, ihren Typ, den Hash-Wert und den dieser Konstante zugewiesenen Wert an. Die Zuordnung zwischen Name und Hash-Wert eines Konfigurationsparameters wird zur Verwendung in SDL als PR-Datei ausgegeben und sieht für dieses Beispiel wie in Listing 6.6 aus. Anschließend wird die Konfiguration in einem binären Dateiformat (siehe Abbildung 6.2) gespeichert.

6.2.2. Verwendung in SDL

In SDL bindet man die generierte PR-Datei in die SDL-Entwicklungsumgebung oder als externe Referenz in einem SDL-Package ein. Das Auslesen der Werte einer Konfiguration erfolgt über SDL-Prozeduren, die über SEnF plattformunabhängig mit den SDL-Konstanten (`SYNONYM`) der Hash-Werte als Parameter auf die Konfiguration zugreifen. Damit auch in SDL die Typinformationen erhalten bleiben, existieren vier Prozeduren, die jeweils den geforderten Typ zurückliefern. Sollte der Typ in der Konfiguration nicht dem geforderten entsprechen oder kein Eintrag zu dem Hash-Wert in der Indextabelle vorhanden sein, wird die Ausführung angehalten (über das Auslösen eines *SDLError*). Da Konfigurationen in der Regel beim Systemstart einmalig ausgelesen werden, stellt das Beenden der Ausführung kein Problem dar.

```
DCL
  master BOOLEAN,
  period, nextFreeSlot REAL;

master:=call getConfigBool('config.cfg', cfg_mac_master);
period:=float( call  getConfigInt('config.cfg', cfg_mac_period))* MiSEC;
nextFreeSlot:=float( call  getConfigInt('config.cfg', cfg_mac_nextFreeSlot))* MiSEC;
```

Listing 6.7: Verwendung der Konfiguration

In SDL erzeugt man, wie in Listing 6.7 gezeigt, eine Task und liest alle Werte mit Hilfe der Prozeduren aus. Der erste Parameter des Prozeduraufrufs enthält den Dateinamen der Konfiguration, der nur auf PC-Plattformen ausgewertet wird, der zweite

---

[40] Die Hash-Funktion von Java ist für einen String s der Länge n mit $h(s) = \left(\sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}\right)$ mod 2^{32} definiert. Kollisionen in Variablennamen sind dennoch eher selten zu erwarten.

Parameter den Hash-Wert, der über die generierten SDL-Konstanten angesprochen wird.

6.2.3. Fazit

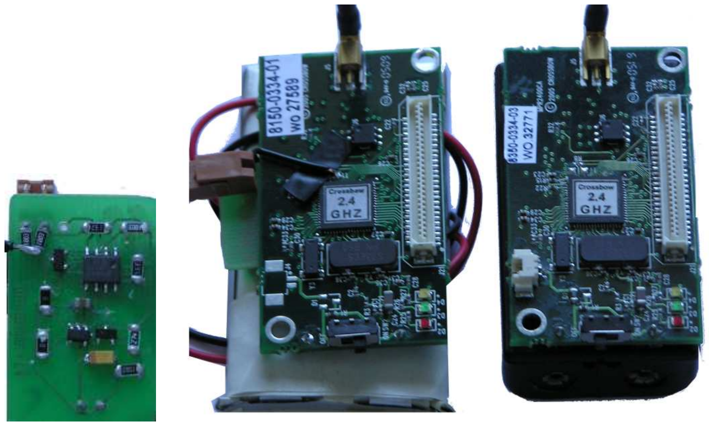
Mit Hilfe der SDL-Konfigurationsschnittstelle und des dafür entwickelten Werkzeugs in Java lassen sich auch komplizierte Konfigurationen effizient und plattformunabhängig speichern. Die Verwendung von Java und Reflections bietet klare Vorteile gegenüber normalen Textdateien, da der komplette Sprachumfang von Java zur Verfügung steht, der es erlaubt, externe Dateien und Benutzereingaben einzulesen. Bestehen Abhängigkeiten zwischen den Konstanten oder berechnen sich Konstanten aus anderen, läßt sich dies im Vorfeld berechnen und auf mögliche Fehler hin testen. Durch eine Erweiterung von T. Braun lassen sich nun auch komplexe Datenstrukturen wie Klassen und Listen (Array) innerhalb von SDL als Konstanten verwenden. Dazu wird zusätzlich zu der PR-Datei noch eine ASN.1-Datei [44, 73] erzeugt, die eine Beschreibung der Datenstruktur enthält und diese auf `struct`, `union` und den Array-Generator abbildet. Hierüber können komplexe Datenstrukturen auf einmal initialisiert werden und stehen in SDL dann direkt zur Verfügung.

6.3. Hardware-Erweiterungen für die Sensorplattformen MICAz und Imote2

Keine der verwendeten Sensorplattformen (MICAz, Imote2) wird betriebsfertig ausgeliefert. Nicht nur Software muß dafür bereitgestellt werden, es fehlt an geeigneten Batterien, Zugang zu Debug-Schnittstellen oder an Adaptern, um digitale oder analoge Sensoren/Aktuatoren an die Ein-/Ausgänge anzuschließen. Zu Beginn der Arbeit waren zudem von Herstellerseite für den Imote2 keine Erweiterungen verfügbar und aufgrund der ausschließlich digitalen Schnittstellen konnten keine analogen Sensoren angeschlossen werden. Die hierfür entwickelte Hardware wird im Folgenden vorgestellt.

6.3.1. Batteriemanagement-Platine für den MICAz

Sensorknoten sollen klein und leicht sein und eine lange Batterielaufzeit haben. Der MICAz wird vom Hersteller mit einer Halterung für handelsübliche AA-Batterien ausgeliefert. Vergleicht man Energie pro Gewichtseinheit, schneiden AA-Batterien oder *Nickel-Metallhydrid*-Akkus (NiMH) schlechter ab als moderne *Lithium Polymer*-Akkus (LiPo). In Abbildung 6.3(b) ist der Größenunterschied beider Batterie-Techniken bei gleicher Kapazität gezeigt. Der LiPo-Akku ist im Vergleich mit 5x70x37 mm, deutlich kleiner als die Variante mit 2 AA-Zellen, die mit Batterienhalterung 14x58x32 mm mißt. Die neuere LiPo-Technik ermöglicht es auch, den aktuellen Ladezustand des Akkus genauer zu ermitteln, wie detailliert in Kapitel 3.2 beschrieben. Gerade in Umgebungen, in denen Gewicht und Laufzeit eine Rolle spielen, setzt man deshalb besser auf die LiPo-Technik. Zu beachten ist jedoch, daß die Akkuspannung einer LiPo-Zelle zwischen 3,3 V und 4,2 V liegt und damit um 1,2 V höher als die Spannung zweier AA-Zellen. Eine Unterschreiten der Spannung von 3,3 V stellt eine *Tiefentladung* des Akkus dar und führt zu irreparabelen Schäden am LiPo-Akku.



(a) Batteriemanagement-Platine

(b) MICAz mit LiPo-Akku bzw. Batteriepack

Abbildung 6.3.: Batterien des MICAz

Für den Betrieb eines LiPo-Akkus an einem MICAz ist es folglich unerläßlich, die Spannung zu reduzieren und den Akku vor einer Tiefentladung zu schützen. Die für diesen Zweck entwickelte Batteriemanagement-Platine aus Abbildung 6.3(a) stellt dazu dem MICAz eine stabile und unveränderliche Eingangsspannung von 3 – 3,1 V zur Verfügung, bis die Abschaltspannung von 3,3 V erreicht ist (vergleiche Kapitel 3.2, Abbildung 3.3). Da sich die Akkuspannung nach Abschalten aller Verbraucher erholt, also um mehre Millivolt steigt, wurde die Hysterese so eingestellt, daß der Verbraucher – in diesem Fall der MICAz – nicht erneut eingeschaltet wird, bis eine Ladung des Akkus erfolgt ist. Zusätzlich wurde eine Meßbrücke eingebaut, die

die aktuelle Batteriespannung an einen Analog-Eingang des Mikrocontrollers legt und damit eine Überwachung der aktuellen Batteriespannung ermöglicht. Mit Hilfe dieser Schaltung wurde die Spannungsversorgung des MICAz-Knotens stabilisiert, wodurch die Erstellung der Energiemodelle und die überprüfenden Messungen [65], wie sie in Kapitel 3.1 und Kapitel 3.2 zu finden sind, erst möglich waren.

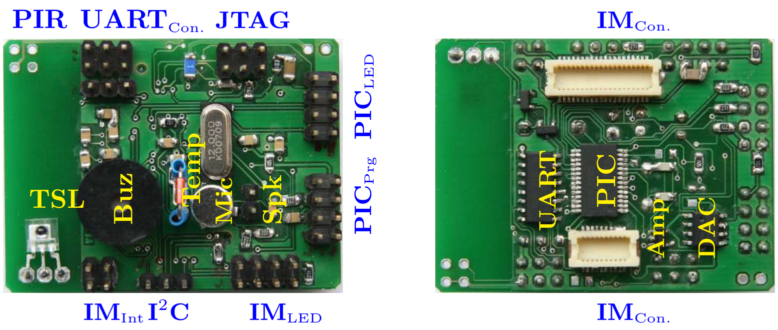


Abbildung 6.4.: Sensorboard des Imote2 (Abkürzungen siehe Tabelle 6.1)

6.3.2. Imote2-Sensorboard

Da der Imote2 ausschließlich digitale Schnittstellen besitzt, ist eine direkte Erfassung von analogen Sensorwerten nicht möglich. Auch die Schnittstellen-Konnektoren des Imote2 sind nicht dazu ausgelegt, direkt einen Sensor aufzunehmen oder ein Kabel daran anzuschließen. Aus diesen Gründen wurde ein Sensorboard mit zahlreichen Sensoren und Aktuatoren, wie es in Abbildung 6.4 zu sehen ist, entwickelt. Zusätzlich besitzt das Board einige Pfostenstecker, die sich deutlich besser für die Prototypenentwicklung eignen. Viele davon können als digitale und einige ebenfalls als analoge Ein- und Ausgänge genutzt werden. Alle analogen Sensorwerte werden dabei zunächst von einem energiesparsamen, auf diesem Board integrierten, Mikrocontroller (PIC18F4321 [82]) zu digitalen Werten vorverarbeitet und digital über die *SPI*-Schnittstelle an den Imote2 übermittelt.

Einige gebräuchliche und häufig benötigte Analogsensoren und Aktuatoren sind auf dem Sensorboard bereits integriert: Ein Temperatur-Sensor (Temp), ein Lichtsensor (TSL), ein Mikrofon inklusive Vorverstärker und Bandpaß-Filter sowie ein 8 Bit-PIC18F-Mikrocontroller zur Digitalisierung des Mikrofonsignals. Es existiert aber auch ein analoger Audio-Ausgang, der zum Anschluß eines Kopfhörers oder eines verstärkten Lautsprechers genutzt werden kann. Dazu wird das digitale Signal über den *Digital/Analog-Converter* (DAC) in ein analoges Signal umgewandelt. Neben

Amp	*Amplifier*, Verstärker für den Anschluß von Kopfhörern
Buz	*Buzzer*, Tongenerator um einen Warnton auszugeben
DAC	*Digital-Analog-Converter* (MCP4921), erzeugt aus digitalen Werten ein analoges Ausgangssignal
$\mathrm{IM}_{\mathrm{Con.}}$	Anschluß, um das Sensorboard mit dem Imote2 zu verbinden
$\mathrm{IM}_{\mathrm{Int}}$	Anschluß, um direkt auf I/O-Pins des Imote2 zuzugreifen. Diese Pins können benutzt werden, um einen Imote aus dem Tiefschlaf via Interrupt zu wecken.
$\mathrm{IM}_{\mathrm{LED}}$	Digital-I/O-Anschluß an Imote2. Alle Pins besitzen einen Vorwiderstand für eine LED.
$\mathrm{I}^2\mathrm{C}$	Anschluß an den $\mathrm{I}^2\mathrm{C}$-Bus dem Imote2
JTAG	JTAG-Anschluß zur Programmierung und zum Debuggen des Imote2
Mic	Mikrofon, an den PIC angeschlossen
PIC	Texas Instruments PIC18F4321, als Hilfscontroller zur Analog-Digital-Wandlung
$\mathrm{PIC}_{\mathrm{Prg}}$	Programmieranschluß für PIC
$\mathrm{PIC}_{\mathrm{LED}}$	Anschluß für weitere LED; Vorwiderstände bereits vorhanden.
PIR	*passive infrared*, Anschluß für einen 3 V-Bewegungsmelder. 2 digitale Eingänge, sowie 3 V Spannungsversorgung.
Spk	*Speaker*, Anschluß für einen Kopfhörer
Temp	Temperatursensor
TSL	Licht-Spannungswandler
UART	Chip für die Wandlung von 3 V-UART (TTL-Level) auf ±12 V für PC
$\mathrm{UART}_{\mathrm{Con.}}$	Anschluß von 2 UART für PC, 1 UART in TTL

Tabelle 6.1.: Abkürzungserklärung der Hardwarekomponenten des Sensorboards aus Abbildung 6.4

diesem Ausgang steht ein Summer (Buz) als digitaler Aktuator zur Verfügung. Des weiteren ist das Board für den Anschluß eines passiven Infrarot-Bewegungsmelder (PIR) mit 3V Betriebsspannung vorbereitet. Auf dem Board sind weitere LEDs vorgesehen, aber nicht fest bestückt, da es sonst nicht mehr möglich wäre, diese Pins als Ein- bzw. Ausgang zu verwenden. Alle mit „LED“ bezeichneten Anschlüsse verfügen bereits über einen für LEDs ausreichend dimensionierten Vorwiderstand zur Strombegrenzung, sodaß eine LED direkt daran angeschlossen werden kann. Für den Anschluß des UART-Ports an einer seriellen PC-Schnittstelle wurde auf dem Sensorboard ein Pegelwandler integriert, der die $0 - 3{,}3\,\mathrm{V}$ TTL-Pegel des Imote2 an die ±12 V der seriellen PC-Schnittstelle anpaßt. Die Details zur softwareseitigen Ansteuerung des Sensorboards sind in der Diplomarbeit von T. Schmelzer [95] zu finden, der dazu ein SDL-Package spezifiziert und Erweiterungen an SEnF für das Board durchgeführt hat.

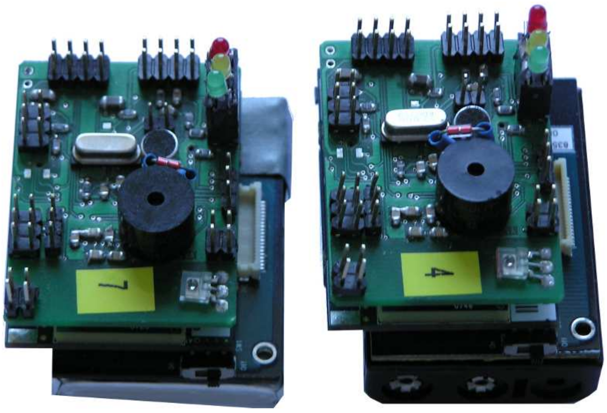


Abbildung 6.5.: Imote2 mit LiPo bzw. Batteriepack, jeweils mit Sensorboard

6.3.3. Stromversorgung des Imote2

Genau wie der MICAz wird auch der Imote2 mit einem normalen Batteriepack für drei AAA-Zellen ausgeliefert. Im Unterschied zum MICAz kann der Spannungscontroller des Imote2 direkt mit LiIon und LiPo-Akkus umgehen und benötigt außer dem Akku keine weitere Hardware. Abbildung 6.5 zeigt dazu jeweils einen Imote2 mit Sensorboard, links mit LiPo-Akku und rechts mit normalen Batteriepack.

6.4. Imote2 Bootloader

Der Imote2 Sensorknoten (vergleiche Kapitel 2.2.2) besitzt keine spezielle Programmier-Schnittstelle, kann aber über die Hardware-Debug-Schnittstelle (JTAG [53]) programmiert werden. Der Anschluß eines JTAG-Programmiergerätes ist aufgrund der Anschlüsse des Imote2 nicht problemlos möglich und setzt weitere, spezielle Hardware voraus. Durch einen Bootloader [98], der standardmäßig auf dem Imote2 installiert ist, kann die Programmierung über die Mini-USB-Schnittstelle erfolgen. Fehlt jedoch der Bootloader oder ist dieser beschädigt, muß der Flash-Speicher des Imote2 über die JTAG-Schnittstelle beschrieben werden.

Aufgrund mangelnder Zuverlässigkeit und geringer Geschwindigkeit des alten Bootloaders und vieler „defekter“ Imote2-Knoten wurde ein komplett neuer Boot- und Kodeloader [32, 63] für den Imote2 entwickelt.

6.4.1. Kode- und Bootloader

Bevor der Imote2 Knoten beschrieben werden kann, muß zunächst der C bzw. C++ Quelltext mit Hilfe des GNU C-Compilers (gcc) [38] in Maschineninstruktionen für die Imote2-Plattform gewandelt werden (siehe Abbildung 6.6). Die Maschineninstruktionen werden zusammen mit weiteren Basisfunktionen aus Bibliotheken zu einer Datei, dem *Image*, zusammengefaßt. Dieses Image übergibt man dem *Kodeloader*, einem Programm, das auf dem PC läuft und sich über die USB-Schnittstelle mit dem Imote2 verbindet. Der dort laufende *Bootloader* empfängt das Image und schreibt dieses in seinen Flash-Speicher.

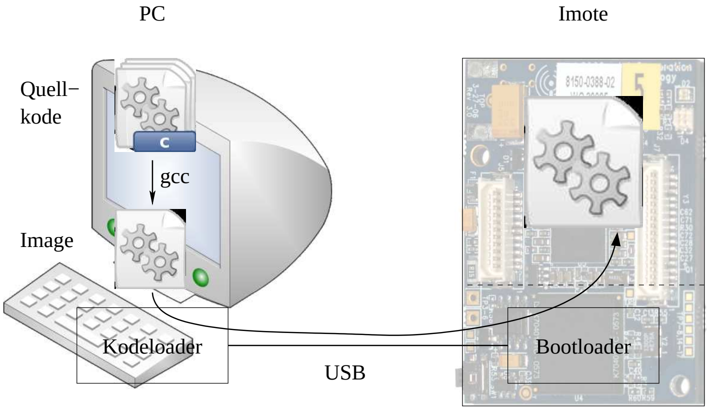


Abbildung 6.6.: Ablauf der Programmierung eines Imote2

Der Kodeloader muß jedoch zuerst die Kommunikation mit dem Bootloader initiieren und über die zu übertragenden Daten informieren. Bevor die Speicherzellen beschrieben werden können, müssen sie zuvor durch den Bootloader gelöscht werden. Im Anschluß daran überträgt der Kodeloader das Image, gefolgt von einer zusätzlichen „magischen Sequenz“, über die ein vollständiger Programmiervorgang erkannt werden kann. Der Bootloader schreibt alle so übertragenen Bytes sequentiell in die Flashzellen. Ist die Übertragung beendet, veranlaßt der Kodeloader einen Neustart des Knotens und beendet sich selbst. Beim Neustart überprüft der Bootloader, ob ein vollständiges Image im Flash vorhanden ist und ob sich ein Kodeloader mit ihm verbinden will. Ist das Image vollständig und kein Kodeloader zur Verbindung bereit, wird das Datensegment geladen und damit das Image aus dem Flash gestartet.

Bootloader Der Boot-strap-loader (Bootloader) ist auf der Imote2-Plattform für zwei wesentliche Dinge verantwortlich: Das zuverlässige Entgegennehmen und Schreiben eines Images vom PC und die Ausführung dieses Images, sofern es vollständig übertragen wurde. Beim Bootvorgang des Imote2 prüft deshalb der Bootloader zunächst, ob sich ein vollständiges Image im Flash-Speicher befindet. Anschließend wird geprüft, ob der Imote2 an einen PC angeschlossen ist. Ist dies der Fall, wird der USB-Anschluß aktiviert und der PC erhält für 2 Sekunden die Möglichkeit, die Kommunikation zu beantworten. Erhält der Bootloader innerhalb dieser Zeit keinen Verbindungsaufbau durch den Kodeloader, startet er das Image aus dem Flash-Speicher. Ist das Image jedoch unvollständig, wartet der Bootloader ständig auf die Verbindung eines Kodeloaders. Der Kommunikationsautomat des Bootloaders, der zur Kommunikation verwendet wird, ist in Abbildung 6.7 gezeigt. Aufgrund der Symmetrie des Kommunikationsautomaten für den Kodeloader entfällt dessen Darstellung.

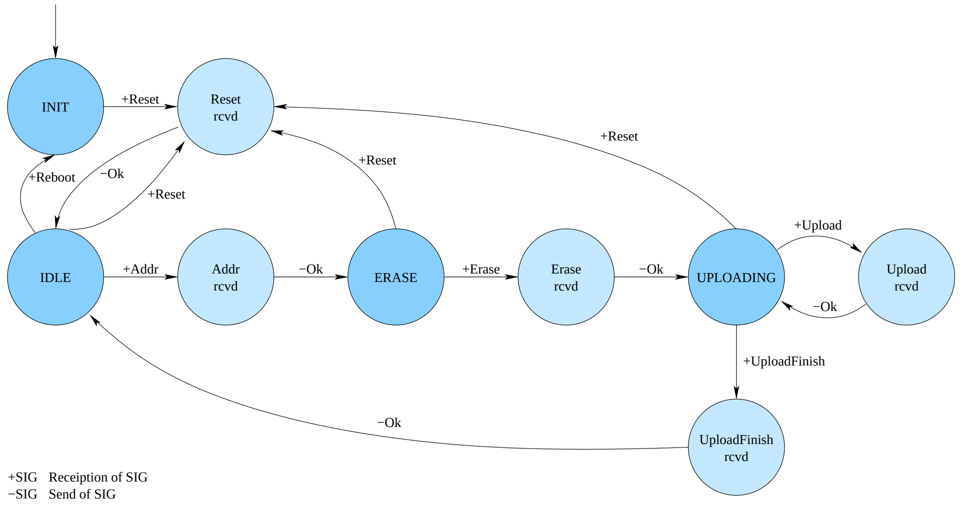


Abbildung 6.7.: Kommunikationsprotokoll auf Bootloader-Seite

Zu Beginn der Kommunikation sendet der Kodeloader zunächst ein Reset-Signal, um das Vorhandensein des Kodeloaders anzuzeigen und beide Automaten auf den definierten `IDLE`-Zustand zu bringen. Der Bootloader des Imote2 quittiert diesen

Vorgang mit einem `Ok`-Signal. Der Kodeloader kann im `IDLE`-Zustand den Knoten mittels `Reboot`-Signal den Automaten neustarten oder durch Übermittlung einer Adresse den Programmiervorgang starten. Durch die Übermittlung einer Adresse und einer Länge als Parameter, gefolgt von einem `Erase`-Signal, löscht der Bootloader ab der Adresse bis zur übermittelten Größe den gesamten Bereich. Nach dem erfolgreichen Löschen wird dies vom Bootloader mittels `Ok` quittiert und er geht in den Zustand `UPLOADING`. Alle zu programmierenden Daten werden über das Upload-Signal übertragen und jeweils mit `Ok` quittiert. Nachdem alle Daten übertragen wurden, kehrt der Bootloader nach Empfang des `UploadFinished`-Signal in den `IDLE`-Zustand zurück. Müssen weitere Bereiche des Imote2, wie z.B. eine Konfiguration geschrieben werden, kann dies direkt im Anschluß erfolgen. Ist keine weitere Übertragung nötig, startet der Kodeloader den Knoten mittels `Reboot`-Signal neu.

Kodeloader Bei der Neuentwicklung des Kodeloaders war die Plattformunabhängigkeit sehr wichtig. Deshalb wurde der Kodeloader in der Sprache Java [87] implementiert, wodurch der Kodeloader auf vielen Systemen ohne Neuübersetzen direkt lauffähig ist. Die einzige Voraussetzung ist neben einer Java-Installation, eine native Version der *libusb*-Bibliothek [86], die für den Zugriff auf die USB-Schnittstelle benötigt wird. Aktuell ist diese Bibliothek für PC/Windows, PC/Linux und MAC verfügbar und stellt somit für alle gängigen Betriebssysteme keine Einschränkung dar. Im Vergleich dazu war der ursprüngliche Kodeloader, der ausschließlich in einer *cygwin*-Umgebung [90] unter *Windows* [83] ausgeführt werden konnte, sehr stark eingeschränkt.

6.4.2. Aufteilung des Flash-Speichers

Im Gegensatz zum ursprünglichen Bootloader, der nur mit Images von maximal 1 MiB sicher umgehen konnte, kann der neue Bootloader den gesamten Speicher von 32 MiB beschreiben, lediglich 256 kiB sind exklusiv für den Bootloader reserviert. Damit die SDL-Konfigurationsschnittstelle (Kapitel 6.2) und das auf dem Knoten ausgeführte Image einige Werte des Knotens auslesen können, wurde eine Konvention für das Layout eingeführt, die in Tabelle 6.2 gezeigt ist.

Bei allen ARM-Architekturen steht in den ersten 32 Byte der Interrupt-Vektor, der Einsprungpunkte für auftretende Interrupts enthält. Dieser Vektor wird sowohl vom Bootloader als auch vom ausgeführten Image benötigt. Der nächste Speicherbereich von 256 kiB Größe wurde für den Bootloader fest reserviert. Die aktuelle Größe des Bootloaders beträgt zwar nur 60 kiB, jedoch lassen sich Flash-Zellen ausschließlich

Speicherbereich	Größe	Erklärung
`0x00000020 - 0x0003FFF7`	32 Byte	Interrupt-Vektor (fix)
`0x00000024 - 0x0003FFF7`	256 kiB	Bootloader (fix)
`0x0003FFF8 - 0x0003FFFB`	4 Byte	Knoten-ID (fix)
`0x0003FFFC - 0x0003FFFF`	4 Byte	Seriennummer (fix)
`0x00040000 - 0x003FFFFF`	1,744 MiB	Konfigurationsdaten
`0x00400000 - 0x02000000`	28 MiB	Programm

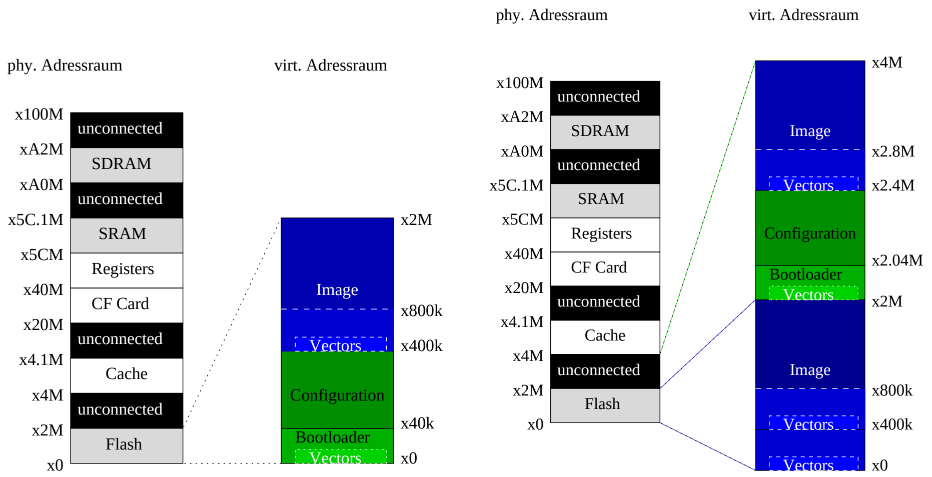
Tabelle 6.2.: Neue Speicheraufteilung des Flash-Speichers

blockweise löschen und die Blockgröße beträgt in diesem Bereich 256 kiB. Da Anwendungen, die auf einem Knoten laufen, häufig eine eindeutige Identifikation benötigen, wurden 2 Felder von jeweils 4 Byte vorgesehen. Zum einen beinhalten diese Felder die auf dem Knoten aufgedruckte, eindeutige Seriennummer, zum anderen eine (meist kürzere) frei wählbare Knoten-ID. Selbst umfangreiche Konfigurationen lassen sich in dem ~1,7 MiB großen Bereich unterbringen. Die verbleibenden 28 MiB können frei für das Image verwendet werden und bieten damit genug Raum um auch große Anwendungen auf den Sensorknoten zu laden. Durch die Trennung zwischen Konfiguration und Image ist es möglich, Parameter der Konfiguration auszutauschen, ohne dabei das Image ändern zu müssen.

6.4.3. Adressraum-Aufteilung

Bei jedem Programmiervorgang kann entschieden werden, ob das Image im Flash oder im SDRAM ausgeführt wird. Da sich die Speicheradressen jedoch erheblich unterscheiden, müssen Speicherbereiche mittels *Memory Management Unit* (MMU) und der darin enthaltenen Tabelle abgebildet werden. Tritt während der Ausführung des Images ein Interrupt auf, sollten auch die Behandlungsroutinen des Images aufgerufen werden und nicht die des Bootloaders. Eine Indirektion des Interrupts über die Routinen des Bootloaders ist zwar möglich, hätte aber Verzögerungen zur Folge, die in diesen Routinen möglichst klein gehalten werden sollten.

Ohne aktivierte MMU ist der Adressraum wie in Abbildung 6.8(a) zu sehen und entspricht dem physikalischen Zustand auf dem Knoten. Der Interrupt-Vektor an Adresse `0x0` gehört zum Bootloaders. Der Interrupt-Vektor an Adresse `0x400k` gehört zum Image und ist der Einsprungpunkt im Image beim Auftreten eines Interrupts. Ohne MMU würde somit immer der Interrupt-Vektor des Bootloaders aktiviert, nicht jedoch der des Images.



(a) Speicherlayout zum Zeitpunkt der Ausführung des Bootloaders, ohne MMU

(b) Speicherlayout bei Ausführung des Programms, mit MMU

Abbildung 6.8.: Speicherlayout zum Zeitpunkt der Ausführung des(a) Bootloaders und des (b) Programms aus dem Flash

Um eine Behandlung von Interrupts durch das Image zu ermöglichen, wird die MMU aktiviert und so konfiguriert, wie in Abbildung 6.8(b) gezeigt. Dazu wird der physische Speicher, in dem das Image liegt, inklusive Konfiguration und Bootloader in den ungenutzten Speicher ab Adresse `0x2M` abgebildet. Der Interrupt-Vektor des Images wird auf den Anfang des Speichers abgebildet und überdeckt dabei den Vektor des Bootloaders, den Bootloader selbst und auch die Konfiguration.[41] Das Image wird weiterhin von Adresse `0x400k` gestartet und der im Image enthaltene Interrupt-Vektor von Adresse `0x0` verwendet.

Um ohne Änderung am Image dieses ebenfalls aus dem SDRAM ausführen zu können, muß das Speicherlayout so angepaßt werden, sodaß es für das Image keine Änderungen an den virtuellen Adressen gibt. Abbildung 6.9 zeigt den Adressraum wie er bei der Ausführung im SDRAM (*in-RAM-Execution*) verwendet wird. Bei der in-RAM-Execution wird nur das Programm in den RAM geschrieben – Konfigurationen müssen weiterhin fest im Flash hinterlegt werden. Das Image liegt dabei physisch im SDRAM-Bereich bei Adresse `0xA0M` und wird von dort auf den virtuellen Adressbereich `0x400k` abgebildet, wo das Image im Flash-Speicher liegen würde. Der Interrupt-Vektor wird ebenfalls zusätzlich auf die Adresse `0x0` abgebildet. Es wäre

---

[41] Grund hierfür ist, das die MMU nur mit Segmenten der Granularität von 1 MiB umgehen kann

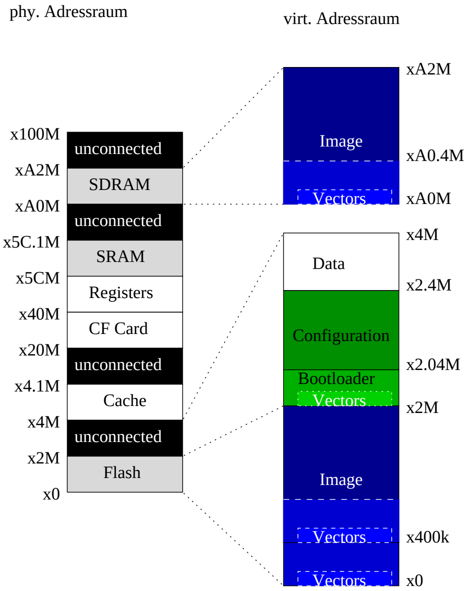


Abbildung 6.9.: Speicherlayout bei Ausführung des Programms im SDRAM

nun nicht mehr möglich auf alle Konfigurationen zuzugreifen, weshalb der Inhalt des Flash-Speichers ebenfalls auf die virtuelle Adresse `0x2M` abgebildet wird. Der Adressbereich `0x2.4M - 0x4M` enthält die Daten, die physisch dem Inhalt des Flash-Speichers entsprechen. Somit wäre es möglich, fast den kompletten Flash-Speicher für Daten und Konfigurationen zu verwenden und das Programm ausschließlich im SDRAM vorliegen zu haben.

6.4.4. Geschwindigkeit

Eine der großen Vorteile des neuen Boot- und Kodeloaders ist der schnellere und zuverlässigere Transfer der Daten über die USB-Schnittstelle. Um dies zu verdeutlichen, zeigt Tabelle 6.3 die unterschiedlichen Transferzeiten eines 1,6 MiB großen Images.[42] Die in der Tabelle angegebene Dauer ist die Zeit, die für einen kompletten Programmiervorgang inklusive Löschen benötigt wird. Für den neuen Kodeloader wird zusätzlich die dort ermittelte reine Transferrate angegeben, während die rechne-

---

[42] Images, die eine Größe von 1,6 MiB überschreiten, zerstören den alten Bootloader. Ohne JTAG-Programmieradapter führt dies zum Ausfall des Knotens führt.

rische Transferrate, die Imagegröße durch die Gesamtzeit (inklusive Flash-Vorgang) teilt.

Programmierung	Dauer	Transferrate	rechnerische Transferrate
JTAG-Schnittstelle	116 s	–	14 kiB/s
vorinstallierter Bootloader	90 s	–	18 kiB/s
neuer Bootloader (Flash)	29 s	123 kiB/s	56 kiB/s
neuer Bootloader (SDRAM)	2,8 s	773 kiB/s	580 kiB/s

Tabelle 6.3.: Vergleich der verschiedenen Upload-Möglichkeiten (Programmgröße 1,6 MiB)

Der Verlierer des Vergleichs ist die JTAG-Schnittstelle, die für das Programmieren des Images knapp 2 Minuten benötigt. Der vorinstallierte Bootloader benötigt 1 Minute und 30 Sekunden und ist somit nur geringfügig schneller. Klarer Gewinner im Vergleich ist der neue Bootloader, der mit 29 Sekunden mehr als dreifach so schnell ist wie der ursprüngliche Bootloader. Ist es nicht nötig, das Image persistent auf den Flash-Speicher zu schreiben und reicht die einmalige Ausführung, kann der neue Bootloader das Image auch direkt in den SDRAM schreiben und ohne Neustart ausführen. Diese Option ist gerade für den schnellen Prototypen-Test von Vorteil, zum einen entfallen bei großen Images längere Wartezeiten, zum anderen schont die *in-RAM-Execution* die Flashzellen, die bei jedem Programmieren geschädigt werden.[43] In diesem Test war der Upload für die in-RAM-Execution nach nur 2,8 Sekunden beendet, also im Vergleich zum ursprünglichen Bootloader um den Faktor 30 schneller!

6.4.5. Fazit

Der entwickelte Bootlader und Kodeloader stellt ein wichtiges Werkzeug zur Programmierung des Imote2 dar, ohne das die Entwicklung von großen (SDL-)Systemen, wie der NCS-CoM für den Imote2 nicht möglich gewesen wäre. Neben der höheren Zuverlässigkeit ist die Identifizierung von Knoten anhand der darauf gespeicherten ID ein weiterer Vorteil des Kodeloaders. Es ist somit möglich, auch bei vielen Knoten, nur die für den Knoten vorgesehenen Daten zu programmieren. Zur Vereinfachung können dabei die Parameter aller Knoten in einer Konfigurationsdatei hinterlegt werden. Die Zuordnung zwischen den Parametern und dem Knoten erfolgt dann anhand der dort hinterlegten Knoten-ID. Gerade bei Tests mit vielen Knoten können so

---

[43] Jeder Programmiervorgang schädigt die Flashzelle, weshalb nach einer bestimmten Anzahl an Programmiervorgängen die Flashzelle nicht mehr beschreibbar ist.

automatisch alle angeschlossenen Knoten mit den richtigen Daten versorgt werden. Alle hier entwickelten Komponenten sind dual unter der *Intel Open Source License* und der *GPLv2* lizenziert und frei verfügbar [64].

7

Kapitel 7.

Fazit

Die Entwicklung von funkbasierten Regelungssystemen ist nicht mehr nur eine Disziplin von Regelungstechnikern, die einen Regelungsalgorithmus auf einem Mikrocontroller implementieren. Für die Implementierung einer zuverlässigen funkbasierten Regelung müssen zusätzlich die Kommunikationsarchitektur und die Regelung aneinander angepaßt werden. Am Beispiel des inversen Pendels zeigt diese Arbeit, wie sich eine solche Kommunikationsarchitektur für funkbasierte Regelungen mit dem modellgetriebenen Entwicklungsansatz auf Sensorknoten umsetzen läßt. In der praktischen Evaluation (an einem real existierenden inversen Pendel) konnte die Funktionsfähigkeit des Ansatzes gezeigt werden. Trotz der hohen Anforderungen an die Performanz des Systems konnte dabei eine automatische Energieeinsparung von mehr als 60% realisiert werden.

Um dieses Ziel zu erreichen, ist eine genaue Untersuchung der verwendeten Hardwareplattform bereits vor Beginn der Implementierung nötig. Dazu gehörte die Identifizierung der verwendeten Komponenten, ihre Geschwindigkeit, der zur Verfügung stehende Speicher, die Energiezustände und der Energieverbrauch. Hieraus werden Modelle der Hardwareplattform gebildet, die als Grundlage für die Energieoptimierung dienen. Aus ihnen lassen sich die möglichen Energiezustände der Plattform und deren Umschaltzeiten ermitteln. Durch die frühzeitige Erstellung der Modelle kann der Energieverbrauch direkt bei der Modellierung mit berücksichtigt werden und muß nicht erst nachträglich in eine bestehende Anwendung integriert werden. Wissen, das zum Zeitpunkt der Spezifikation über die Abläufe vorhanden ist, geht nicht verloren, sondern kann direkt in der Spezifikation eingebracht werden und muß somit später nicht erneut erarbeitet werden. Mit Hilfe der Energiesignalisierung können somit Ressourcen-Anforderungen direkt aus der Anwendung heraus gesteuert werden. Die Verwendung des modellgetriebenen Entwicklungsansatzes in SDL bietet die Möglichkeit, Kode direkt aus dem Modell zu generieren und diesen auf der Hardwareplattform auszuführen. Neben einer besseren Wartbarkeit und einer kürzeren Entwicklungszeit kann aus der SDL-Spezifikation zusätzliches Wissen abgeleitet und für die implizite Energiesignalisierung verwendet werden, womit automatisch Energie eingespart werden kann.

Ein weiteres Verfahren zur Einsparung von Energie stellt die explizite Spezifikation von Jitter dar. Bei diesem Verfahren annotiert man die Timer mit der maximal zulässigen Auslöseabweichung. Der Scheduler kann die Timer jeweils im Rahmen dieser Abweichung auslösen. Die mögliche Einsparung ergibt sich bei diesem Verfahren aus der Zusammenlegung der auszuführenden Tasks. Die Verlängerung der inaktiven Phase ermöglicht den Wechsel in einen niedrigeren Energiezustand, der sonst aufgrund der längeren Aufwachzeit nicht ausgewählt würde.

Der Einsatz von Echtzeitsignalen in SDL ermöglicht es, eine präzise Synchronisation zwischen den Sensorknoten zu realisieren. Die Synchronisation wird zum einen für die synchrone Erfassung der Meßwerte, zum anderen für die Einteilung des Kommunikationsmediums in Slots für ein TDMA-Verfahren benötigt. Doch die Echtzeitsignalisierung ist in ihrer Anwendung nicht auf die Funkkommunikation beschränkt. Sie senkt ebenfalls die Verzögerung lokal zugestellter Signale in der SDL-Laufzeitumgebung. Da die Verwendung von Echtzeitsignalen häufig einen zusätzlichen Timer und damit auch die Transition zur Verarbeitung des Timers einspart, erhöht sich nicht nur die Wartbarkeit, sondern es verringert sich durch den geringeren Aufwand in der Laufzeitumgebung ebenfalls der Energieverbrauch. Die für die automatische Energieeinsparung verantwortliche implizite Energiekontrolle kann die Informationen der Echtzeitsignale direkt nutzen und dadurch den Energieverbrauch ebenfalls positiv beeinflussen. Doch auch die explizite Energiekontrolle profitiert von den Echtzeitsignalen, indem präzise zu einem bestimmten Zeitpunkt Hardwarekomponenten ein- bzw. ausgeschaltet werden.

Die Entwicklung eines drahtlosen Kommunikationssystems für das inverse Pendel ist in vier Schritte aufgeteilt, von denen drei Schritte im Rahmen dieser Arbeit in SDL realisiert wurden. Die Regelung soll dabei durch die Verwendung einer dienstorientierten Middleware komplett vom Netzwerk abstrahieren, wodurch sich der Zugriff auf Sensoren und Aktuatoren transparent gestaltet. Für den Dienstnutzer ist es dabei ganz gleich, ob der Dienstanbieter auf dem gleichen oder einem entfernten Sensorknoten seinen Dienst erbringt. Für echtzeitkritische Regelungen (wie dem inversen Pendel) ist es essentiell, deterministischen und exklusiven Zugriff auf das Netzwerk zu erhalten und damit sicherzustellen, daß die Daten rechtzeitig und zuverlässig beim Empfänger ankommen. Die hierfür entwickelte MAC-Schicht ermöglicht die deterministische Übertragung der Regelungsdaten. Gleichzeitig synchronisiert sie alle Sensorknoten, um damit eine synchrone Meßwerterfassung sicherzustellen.

Obwohl die Entwicklung am Beispiel des inversen Pendels erfolgte, ist die Adaption der vorgestellten Lösung auf ein anderes Regelungsproblem ohne Kodeänderungen am Kommunikationssystem möglich. Es erfordert lediglich den Austausch der Regelungsanwendung durch einen an das neue Problem angepaßten Algorithmus. Die Anpassung der Knotenkonfiguration und die Slot-Reservierung der MAC-Schicht erfolgen mit Hilfe der SDL-Konfigurationsschnittstelle. Die Middleware und die MAC-Schicht müssen somit weder verändert noch neu übersetzt werden.

Anhang A.

Werzeuge

A.1. SdlRE-Testfalllauf

Nachfolgend ist der Lauf der Testfälle für das SdlRE gezeigt.

```
./component
  _________ ________ ___ _              ________
 /  _____/ |  ______| | \ | |            |  _____|
 | |       |  |___   |   \| |            |  |___
 \_____ \  |   ___|  |  |\  |            |   ___|
  _____| | |  |_____ | | \  |            |  |
 /_______/DL |_______| |_| \_|VIRONMENT |_|RAMEWORK

 Version 0.5


Run SDL....

1 − TimerTest started.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
1 − TimerTest − Testcase 1.1: NonParamTimer: successful.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
1 − TimerTest − Testcase 1.2: SimpleParamTimer started with two distinct parameters:
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
1 − TimerTest − Testcase 1.2: SimpleParamTimer: successful.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
1 − TimerTest − Testcase 1.3: ComplexParamTimer: successful.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
1 − TimerTest − Testcase 1.4: MultiParamTimer: successful.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
1 − TimerTest − Testcase 1.5: NoWaitTimer: successful.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
1 − TimerTest − Testcase 1.6: NegativeWaitTimer: successful.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
1 − TimerTest − Testcase 1.7: PostExecutionResetTimer: successful.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
SDL.Stopped(agent:TimerTester_0)
```

```
2 - ProcessLifecycleTest started.
------------------------------
2 - ProcessLifecycleTest - Testcase 2.1: Instantiating parameterized processes and addressing
      signals to them.
2 - ProcessLifecycleTest - Testcase 2.1: successful. All signals were delivered to their
      addressees.
------------------------------
2 - ProcessLifecycleTest - Testcase 2.2: Terminating one process instance and addressing
      signals to the remaining.
SDL.Stopped(agent:DummyProcess_1)
2 - ProcessLifecycleTest - Testcase 2.2: successful. All signals were delivered to their
      addressees.
------------------------------
SDL.Stopped(agent:ProcessLifecycleTester_0)
SDL.Stopped(agent:DummyProcess_0)
SDL.Stopped(agent:DummyProcess_2)
3 - SignalOrderTest started.
------------------------------
3 - SignalOrderTest - Testcase 3.1: Fill a signal queue with identical signals using the
      keywords TO and VIA.
SDL.Stopped(agent:ReceiverNo1_0)
SDL.Stopped(agent:ReceiverNo2_0)
3 - SignalOrderTest - Testcase 3.1: successful. All signals were received in the correct order.
------------------------------
SDL.Stopped(agent:SignalTester_0)
4 - SignalDeliveryTest started.
------------------------------
4 - SignalDeliveryTest - Testcase 4.1: Send signals to four different processes and check
      whether they are delivered correctly.
SDL.Stopped(agent:TargetNo1_0)
SDL.Stopped(agent:TargetNo2_0)
SDL.Stopped(agent:TargetNo3_0)
SDL.Stopped(agent:TargetNo4_0)
4 - SignalDeliveryTest - Testcase 4.1: successful. All signals were delivered to their
      addressees.
------------------------------
SDL.Stopped(agent:SignalDeliveryTester_0)
5 - TransitionTest started.
------------------------------
5 - TransitionTest - Testcase 5.1: Traverse transitions including enabling conditions and
      continuous signals.
5 - TransitionTest - Testcase 5.1: successful.
------------------------------
SDL.Stopped(agent:TransitionTester_0)
6 - AgentQueuingTest started.
------------------------------
6 - AgentQueuingTest - Testcase 6.1: Adding a signal to an agent's queue while it is processed.
selfSigConsumed: 22s:004962us
timerConsumed: 24s:004952us
diff: 1s:999989us
SDL.Stopped(agent:Receiver_0)
6 - AgentQueuingTest - Testcase 6.1: successful.
```

```
-------------------------------
SDL.Stopped(agent:QueuingTester_0)
7 - ProcedureTest started.
-------------------------------
7 - ProcedureTest - Testcase 7.1: Calling a procedure providing values for all its parameters.
7 - ProcedureTest - Testcase 7.1: successful.
-------------------------------
7 - ProcedureTest - Testcase 7.2: Calling a procedure omitting the value for one of its
      parameters.
7 - ProcedureTest - Testcase 7.2: cancelled. Feature currently not supported.
-------------------------------
7 - ProcedureTest - Testcase 7.3: Calling a procedure with internal states.
7 - ProcedureTest - Testcase 7.3: successful.
-------------------------------
7 - ProcedureTest - Testcase 7.4: Calling a remote procedure.
7 - ProcedureTest - Testcase 7.4: cancelled. Feature currently not supported.
-------------------------------
7 - ProcedureTest - Testcase 7.5: Calling a procedure with in/out parameters.
7 - ProcedureTest - Testcase 7.5: successful.
-------------------------------
SDL.Stopped(agent:ProcedureTester_0)
8 - InheritanceTest started.
-------------------------------
8 - InheritanceTest - Testcase 8.1: Triggering responses from processes which use inheritance.
SDL.Stopped(agent:ParentProcesstype_0)
SDL.Stopped(agent:ChildProcesstype_0)
InheritanceTest: Childblocktype: parent process reported to be a parent.
InheritanceTest: Childblocktype: child process reported to be a child.
SDL.Stopped(agent:SecondController_0)
SDL.Stopped(agent:ParentProcesstype_0)
SDL.Stopped(agent:ChildProcesstype_0)
InheritanceTest: Parentblocktype: parent process reported to be a parent.
InheritanceTest: Parentblocktype: child process reported to be a child.
SDL.Stopped(agent:Controller_0)
SDL.Stopped(agent:ParentProcesstype_0)
SDL.Stopped(agent:ChildProcesstype_0)
InheritanceTest: Parentblocktype: parent process reported to be a parent.
InheritanceTest: Parentblocktype: child process reported to be a child.
SDL.Stopped(agent:Controller_0)
8 - InheritanceTest - Testcase 8.1: successful.
-------------------------------
SDL.Stopped(agent:InheritanceTester_0)
9 - RedefinitionTest started.
-------------------------------
9 - RedefinitionTest - Testcase 9.1: Triggering a response from a substructure composed of
      instances of virtual and redefined block types.
9 - RedefinitionTest - Testcase 9.1: cancelled. Feature currently not supported.
-------------------------------
9 - RedefinitionTest - Testcase 9.2: Triggering a response from a substructure composed of
      instances of virtual and redefined process types.
9 - RedefinitionTest - Testcase 9.2: cancelled. Feature currently not supported.
-------------------------------
```

```
SDL.Stopped(agent:RedefinitionTester_0)
10 − AgentSetDeletionTest started.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
10 − AgentSetDeletionTest − Testcase 10.1: Terminating the only agent of an agentset while a
     signal it sent is still to be delivered by the agentset.
SDL.Stopped(agent:Responder_0)
10 − AgentSetDeletionTest − Testcase 10.1: successful.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
SDL.Stopped(agent:AgentSetDeletionTester_0)
All tests finished.
0 testcase(s) failed.
```

A.2. SDL-Konfigurationsschnittstelle

Die komplette Konfiguration des MAC-Layers, sowie weitere Einstellungen der NCS-CoM inkl. Kommentaren. Eine Beschreibung der NCS-CoM ist in Kapitel 5.2.2, eine Beschreibung der Konfiguration ist in Kapitel 6.2 zu finden. Gezeigt wird dabei exemplarisch die Konfiguration des Positionssensors, des Controllers und eines Aktuators. Der Controller wird dabei gleichzeitig als zentraler Taktgeber zur Formatierung des Mediums für TDMA konfiguriert.

```
public class CONTROLLERConfig{
  MacConfig mac=new MacConfig(nodeRoles.SENSOR1,false);
  ServiceUserConfig user=new ServiceUserConfig();
  ServiceProviderConfig provider=new ServiceProviderConfig();
}
```

Listing A.1: Konfiguration des Positionssensors

```
public class CONTROLLERConfig{
  MacConfig mac=new MacConfig(nodeRoles.CONTROLLER,true);
  ServiceUserConfig user=new ServiceUserConfig();
  ServiceProviderConfig provider=new ServiceProviderConfig();
}
```

Listing A.2: Konfiguration des Controllers

```
public class CONTROLLERConfig{
  MacConfig mac=new MacConfig(nodeRoles.ACTUATOR,false);
  ServiceUserConfig user=new ServiceUserConfig();
  ServiceProviderConfig provider=new ServiceProviderConfig();
}
```

Listing A.3: Konfiguration des Aktuators

```
public class ServiceProviderConfig{
  //Time to determine if this service is already registered [ms]
  int confirmationTimeout=400;

  //Time to say other nodes, this service is still alive [s]
  static int alive=2;

  /*If no subscriptionIndication (periodic) received after this period,
   the subscriber is considered dead and subscription is canceled [s]
  */
  int subscriptionTimeout = ServiceUserConfig.subscribeAck * 10;
}
```

Listing A.4: Service Provider Konfiguration

```
public class ServiceUserConfig{
  //if no response was received after this time, the subscription is considered unsuccessfull [
      ms]
  int confirmationTimeout = 8000;

  //indicating a subscription is still valid - forward to provider [s]
  static int subscribeAck = 2;

  //time until a service in considered dead, and subscription is canceled [s]
  int aliveTimeout = ServiceProviderConfig.alive*10;
}
```

Listing A.5: Service User Konfiguration

```
class nodeRoles{
  static final int SYNC=0;
  static final int SENSOR1=SYNC+1;
  static final int SENSOR2=SENSOR1+1;
  static final int SENSOR3=SENSOR2+1;
  static final int CONTROLLER=SENSOR3+1;
  static final int ACTUATOR=CONTROLLER+1;
  static private final int last=ACTUATOR;

  static int getCount(){
    return last+1;
  }

  static int getSlot(Integer role){
    Integer retval=1;
    switch(role){
    case ACTUATOR:
      ++retval;
    case CONTROLLER:
      retval+=1+(MacConfig.computationTime/MacConfig.txWindow+((MacConfig.
          computationTime%MacConfig.txWindow)==0?0:1));
    case SENSOR3:
      ++retval;
    case SENSOR2:
      ++retval;
    case SENSOR1:
    }
    return retval;
  }

  static int getFirstFreeSlot(){
    return getSlot(ACTUATOR)+1;
  }
}
```

Listing A.6: NodeRoles aus MAC-Konfiguration

```
class MacConfig{
  /* the sampling interval in ms */
  int period = 60;
  int countOfSlotperMacroSlot = 7 * nodeRoles.getCount();

  /*maximum time used for sending an frame via wireless in ms*/
  static int txWindow = 4;

  /*in ms; will be adapted to txWindows*/
  static int computationTime = 16;

  /* must be a multiple of turn, which itself is a multiple of period,
  i.e. turn == ((roleCount+1) * period) */
  int macroSlotLength = countOfSlotperMacroSlot*period;

  //Channel used for transfer 1-26 (default: 11)
  int channel = 11;

  //TX power 0..31 (default: 31)
  int txPower = 31;
/********* don't touch after this point ***********/

  /* definition as in nodeRoles*/
  private Integer nodeRole=null;
  /*defines if a Node is configured as master => sends synchronisation beacon */
  Boolean master=null;
  Integer nodeSlot=null;
  //the slot, which is reserved for shared data, but specially for this node
  Integer firstFreeSlot =null;
  //how to increment firstFreeSlot to get to the next one
  int nextFreeSlot=nodeRoles.getCount()*period;

  int syncSlot = (macroSlotLength-(nodeRoles.getCount())*period) + nodeRoles.
       getFirstFreeSlot()*txWindow + nodeRoles.SYNC*period;

  MacConfig(Integer nodeRole, Boolean master){
    this.nodeRole=nodeRole;
    this.master=master;
    nodeSlot = nodeRoles.getSlot(nodeRole)*txWindow;
     firstFreeSlot = nodeRoles.getFirstFreeSlot()*txWindow+nodeRole*period;

    assert (nodeRoles.getFirstFreeSlot()*txWindow+txWindow)<period:"Error␣in␣period";
    assert syncSlot<macroSlotLength:"syncSlot␣not␣within␣macroslot:␣"+syncSlot+">"+
         macroSlotLength+"!";
    assert ((nodeRoles.getCount())<=countOfSlotperMacroSlot):"bad␣count␣of␣Slot/Makroslot
         ␣(1)";
    assert (countOfSlotperMacroSlot%nodeRoles.getCount()==0):"bad␣count␣of␣Slot/
         Makroslot␣(2)";
}}
```

Listing A.7: Konfiguration MAC Layer

```
SYNONYM cfg_mac_period Integer=−1013250383;
SYNONYM cfg_mac_countOfSlotperMacroSlot Integer=795551937;
SYNONYM cfg_mac_txWindow Integer=−1485047676;
SYNONYM cfg_mac_computationTime Integer=33329732;
SYNONYM cfg_mac_macroSlotLength Integer=1327612032;
SYNONYM cfg_mac_channel Integer=71698963;
SYNONYM cfg_mac_txPower Integer=−1578202639;
SYNONYM cfg_mac_master Integer=−1102791854;
SYNONYM cfg_mac_nodeSlot Integer=1913081872;
SYNONYM cfg_mac_firstFreeSlot Integer=1943168618;
SYNONYM cfg_mac_nextFreeSlot Integer=1489684365;
SYNONYM cfg_mac_syncSlot Integer=−1688221655;
SYNONYM cfg_user_confirmationTimeout Integer=85903608;
SYNONYM cfg_user_subscribeAck Integer=−794368301;
SYNONYM cfg_user_aliveTimeout Integer=307329992;
SYNONYM cfg_provider_confirmationTimeout Integer=2033144862;
SYNONYM cfg_provider_alive Integer=1303382271;
SYNONYM cfg_provider_subscriptionTimeout Integer=971682358;
```

Listing A.8: Konfiguration des Controller in SDL

Publikationsliste

[1] Becker, P. ; Krämer, M. : SDL Modules – Concepts and Tool Support. In: Kraemer, F. A. (Hrsg.) ; Herrmann, P. (Hrsg.): *System Analysis and Modeling: About Models – SAM 2010, 6th International Workshop on System Analysis and Modeling, Oslo, Norway, October 4–5, 2010, Co-located with MODELS 2010* Bd. 6598, Springer, 2010 (LNCS), S. 1–17

[2] Chamaken, A. ; Krämer, M. ; Litz, L. ; Gotzhein, R. : Model-based C^3-Cross-Design for Wireless Networked Control Systems. In: *NES/TCOC Symposium on Recent Trends in Networked Systems and Cooperative Control (NESCOC) and Workshop on Network Induced Constraints in Control (NETCOC), Stuttgart, Germany*, 2009

[3] Chamaken, A. ; Litz, L. ; Krämer, M. ; Gotzhein, R. : A New Approach to the Joint Design of Control and Communication in Wireless Networked Control Systems. In: *Automation 2009, VDI-Berichte/VDI-Tagungsbände*, 2009 (VDI-Berichte 2067), S. 251–255

[4] Chamaken, A. ; Litz, L. ; Krämer, M. ; Gotzhein, R. : Cross-Layer Design of Wireless Networked Control Systems with Energy Limitations. In: *European Control Conference 2009 (ECC'09).* Budapest, Hungary, 08 2009

[5] Christmann, D. ; Gotzhein, R. ; Krämer, M. ; Winkler, M. : Flexible and Energy-efficient Duty Cycling in Wireless Networks with MacZ. In: Drira, K. (Hrsg.) ; Kacem, A. H. (Hrsg.) ; Jmaiel, M. (Hrsg.): *NOTERE*, IEEE, 2010. – ISBN 978–1–4244–7066–2, S. 121–128

[6] Christmann, D. ; Gotzhein, R. ; Krämer, M. ; Winkler, M. : Flexible and Energy-efficient Duty Cycling in Wireless Networks with MacZ. In: *Concurrency and Computation: Practice and Experience* (2012). `http://dx.doi.org/10.1002/cpe.2819`. – DOI 10.1002/cpe.2819. – ISSN 1532–0634

[7] Gotzhein, R. ; Krämer, M. ; Litz, L. ; Chamaken, A. : Energy-aware System Design with SDL. In: Reed, R. (Hrsg.) ; Bilgic, A. (Hrsg.) ; Gotzhein, R.

(Hrsg.): *14th International SDL Forum* Bd. 5719, Springer, 2009 (LNCS). – ISBN 978–3–642–04553–0, S. 19–33

[8] Krämer, M. ; Braun, T. ; Christmann, D. ; Gotzhein, R. : Real-Time Signaling in SDL. In: Ober, I. (Hrsg.) ; Ober, I. (Hrsg.): *15th International SDL Forum* Bd. 7083, Springer, 2011 (LNCS), S. 184–199

[9] Krämer, M. ; Engel, M. : New Bootloader for the Imote2 platform / TU Kaiserslautern. 2009 (373/09). – Forschungsbericht

[10] Krämer, M. ; Geraldy, A. ; Marrón, P. J. (Hrsg.): Energy Measurements for MicaZ Node / Universität Stuttgart. 2006. – Forschungsbericht. – In 5. GI/ITG KuVS. Fachgespräch “Drahtlose Sensornetze”, pages 61–68

[11] Krämer, M. ; Geraldy, A. : Energieoptimiertes Scheduling für Mikrocontroller mit SDL. In: *Fachgespräch Systemsoftware und Energiebewusste Systeme, GI/ITG KuVS*, 2007. – ISSN 1432–7864

[12] Krämer, M. ; Kuhn, T. : Development of a Runtime Environment for SDL Systems on Resource-Limited Platforms / Department of Computer Science, University of Kaiserslautern, Germany. 2005 (345/05). – Forschungsbericht

[13] Krämer, M. : *Entwicklung eines leichtgewichtigen Kommunikationsframeworks auf einem Mikrocontroller mit SDL*, TU Kaiserslautern, Diplomarbeit, 2005. `http://vs.informatik.uni-kl.de/publications/2005/Kr05/DA.pdf`

[14] Webel, C. ; Fliege, I. ; Geraldy, A. ; Gotzhein, R. ; Krämer, M. ; Kuhn, T. : Cross-Layer Integration in Ad-Hoc Networks with Enhanced Best-Effort Quality-of-Service Guarantees. In: *Proceedings of World Telecommunications Congress (WTC 2006).* Budapest, Hungary, 2006

[15] Winkler, M. ; Christmann, D. ; Krämer, M. : Customized Duty Cycling with MacZ. In: Beigl, M. (Hrsg.) ; Cazorla-Almeida, F. (Hrsg.): *23rd International Conference on Architecture of Computing Systems (ARCS 2010) – Workshop Proceedings, Hannover, Germany, 2010*, 2010

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] Álvarez, J. M. ; Díaz, M. ; Llopis, L. ; Pimentel, E. ; Troya, J. M.: Integrating Schedulability Analysis and Design Techniques in SDL. In: *Real-Time Systems* 24 (2003), Nr. 3, S. 267–302

[2] Apache Software Foundation: *Maven*. `http://maven.apache.org/`. `http://maven.apache.org/`. Version: 2013

[3] Apache Software Foundation: *Subversion*. `http://subversion.apache.org/`. `http://subversion.apache.org/`. Version: 2013

[4] Aracil, J. ; Gordillo, F. : The inverted pendulum: a benchmark in nonlinear control. In: *Automation Congress, 2004. Proceedings. World* Bd. 16, 2004, S. 468 –482

[5] ARM Ltd.: *ARM Architecture*. `http://www.arm.com/`. `http://www.arm.com/`. Version: 2013

[6] Atmel: *Datasheet AT45DB041B*. `http://www.atmel.com/dyn/resources/prod_documents/doc3443.pdf`, 2007. – Revision 3443C-DFLSH-5/05

[7] Atmel: *Datasheet AT86RF230*. `http://www.atmel.com/dyn/resources/prod_documents/doc5131.pdf`, 2009

[8] Atmel: *Datasheet ATmega128*. `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, 2013. – Revision 2467M-AVR-11/04

[9] Becker, P. ; Birtel, M. ; Christmann, D. ; Gotzhein, R. : Black-Burst-Based Quality-of-Service Routing (BBQR) for Wireless Ad-Hoc Networks. In: *11th International Conference on New Technologies in Distributed Systems (NOTERE'2011), Paris, France*, IEEE, 2011. – ISBN 978–1–4577–0729–2, S. 1–8

[10] Becker, P. ; Christmann, D. ; Gotzhein, R. : Model-Driven Development of Time-critical Protocols with SDL-MDD. In: [91], S. 34–52

[11] Becker, P. ; Gotzhein, R. ; Kuhn, T. : MacZ – A Quality-of-Service MAC Layer for Ad-hoc Networks. In: *HIS '07: Proceedings of the 7th International Conference on Hybrid Intelligent Systems*, IEEE Computer Society, Sept. 2007. – ISBN 978–0–7695–2946–2, S. 277–282

[12] Becker, P. ; Krämer, M. : SDL Modules – Concepts and Tool Support. In: [61], S. 1–17

[13] Beydeda, S. ; Book, M. ; Gruhn, V. : *Model-driven software development*. Springer `http://books.google.com/books?id=tJHLt\_ivQBAC`. – ISBN 9783540256137

[14] Bluetooth SIG, Inc.: *Bluetooth Specification Documents*. `http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/`, 2013

[15] Börger, E. ; Stärk, R. : *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer, 2003

[16] Bozga, M. ; Graf, S. ; Kerbrat, A. ; Mounier, L. ; Ober, I. ; Vincent, D. : SDL for Real-Time: What is Missing? In: Sherratt, E. (Hrsg.): *SAM*, VERIMAG, IRISA, SDL Forum, 2000, S. 108–121

[17] Bozga, M. ; Graf, S. ; Mounier, L. ; Ober, I. ; Roux, J.-L. ; Vincent, D. : Timed Extensions for SDL. In: Reed, R. (Hrsg.) ; Reed, J. (Hrsg.): *SDL Forum* Bd. 2078, Springer, 2001 (Lecture Notes in Computer Science). – ISBN 3–540–42281–1, S. 223–240

[18] Bræk, R. ; Haugen, Ø. ; Welland, R. (Hrsg.): *Engineering Real Time Systems*. Prentice Hall, 1993

[19] Braun, T. ; Gotzhein, R. ; Wiebel, M. : Integration of FlexRay into the SDL-Model-Driven Development Approach. In: [61], S. 56–71

[20] Chamaken, A. ; Litz, L. : Joint Design of Control and Communication in Wireless Networked Control Systems: A Case Study. In: *American Control Conference (ACC), 2010*, 2010. – ISSN 0743–1619, S. 1835 –1840

[21] Chamaken, A. ; Litz, L. ; Krämer, M. ; Gotzhein, R. : A New Approach to the Joint Design of Control and Communication in Wireless Networked Control

Systems. In: *Automation 2009, VDI-Berichte/VDI-Tagungsbände*, 2009 (VDI-Berichte 2067), S. 251–255

[22] Chamaken, A. ; Litz, L. ; Krämer, M. ; Gotzhein, R. : Cross-Layer Design of Wireless Networked Control Systems with Energy Limitations. In: *European Control Conference 2009 (ECC'09).* Budapest, Hungary, 08 2009

[23] Christmann, D. ; Gotzhein, R. ; Krämer, M. ; Winkler, M. : Flexible and Energy-efficient Duty Cycling in Wireless Networks with MacZ. In: *Concurrency and Computation: Practice and Experience* (2012). `http://dx.doi.org/10.1002/cpe.2819`. – DOI 10.1002/cpe.2819. – ISSN 1532–0634

[24] CxxTest-Devleoper Team: *CxxTest-Homepage.* `http://cxxtest.com`, 2013

[25] Dam, T. van ; Langendoen, K. : An adaptive energy-efficient MAC protocol for wireless sensor networks. In: *SenSys*, 2003, S. 171–180

[26] Dialog Semiconductor: *Datasheet Cellular Power Management and Audio DA9030.* `http://www.dialog-semiconductor.com/product_briefs.php?pr=DA9030`, 2013

[27] Drew, M. ; Liu, X. ; Goldsmith, A. ; Hedrick, K. : Networked Control System Design over a Wireless LAN. In: *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC '05. 44th IEEE Conference on*, 2005, S. 6704 – 6709

[28] Du, S. ; Saha, A. K. ; Johnson, D. B.: RMAC: A Routing-Enhanced Duty-Cycle MAC Protocol for Wireless Sensor Networks. In: *INFOCOM*, 2007, S. 1478–1486

[29] Ebert, J.-P. : *Energy-efficient Communication in Ad Hoc Wireless Local Area Networks*, Fakultät IV – Elektrotechnik und Informatik der Technischen Universität Berlin, Diss., April 2004

[30] El-Hoiydi, A. ; Decotignie, J.-D. : WiseMAC: an ultra low power MAC protocol for the downlink of infrastructure wireless sensor networks. In: *ISCC*, IEEE Computer Society, 2004, S. 244–251

[31] Ellsberger, J. ; Hogrefe, D. ; Sarma, A. : *SDL – Formal Object-oriented Language for Communication Systems.* Prentice Hall, 1997

[32] Engel, M. : *Entwicklung eines flexiblen und robusten Bootloaders für die Imote2 Plattform*, Fachbereich Informatik, Technische Universität Kaiserslautern, Bachelorarbeit, 2009

[33] Fliege, I. : *Component-based Development of Communication Systems*, University of Kaiserslautern, Diss., 2009

[34] Fliege, I. ; Geraldy, A. ; Gotzhein, R. ; Schaible, P. : A Flexible Micro Protocol Framework. In: Amyot, D. (Hrsg.) ; Williams, A. W. (Hrsg.): *Proceedings of 4th SAM (SDL and MSC) Workshop, Ottawa, Canada* Bd. 3319, Springer, 2004 (Lecture Notes in Computer Science), S. 224–236

[35] Fliege, I. ; Geraldy, A. ; Jung, S. ; Kuhn, T. ; Webel, C. ; Weber, C. : Konzept und Struktur des SDL Environment Framework (SEnF) / TU Kaiserslautern. 2005 (341/05). – Forschungsbericht

[36] Fliege, I. ; Grammes, R. ; Weber, C. : ConTraST – A Configurable SDL Transpiler and Runtime Environment. In: Gotzhein, R. (Hrsg.) ; Reed, R. (Hrsg.): *System Analysis and Modeling: Language Profiles, 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31 – June 2, 2006, Revised Selected Papers* Bd. 4320, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–68371–2, S. 216–228

[37] Fliege, I. ; Koch, J. : AmICoM – Formally specified service platform for ambient intelligence networks / Fraunhofer IESE. 2007 (D2.6.1). – Forschungsbericht

[38] Free Software Foundation, Inc.: *GCC, the GNU Compiler Collection.* `http://gcc.gnu.org`, 2013

[39] Free Software Foundation, Inc.: *GNU Make.* `http://www.gnu.org/software/make/`, 2013

[40] Föllinger, O. ; (Hrsg.): *Regelungstechnik.* Hüthig, 1992

[41] Geraldy, A. : *Adaptive Routingprotokolle für drahtlose, mobile Ad-hoc-Netzwerke*, Technischen Universität Kaiserslautern, Diss., 2011

[42] Glaesser, U. ; Karges, R. : Abstract State Machine Semantics of SDL. In: *Journal of Universal Computer Science* 3 (1997), dec, Nr. 12, S. 1382–1414. – `http://www.jucs.org/jucs_3_12/abstract_state_machine_semantics`

[43] Glässer, U. ; Gotzhein, R. ; Prinz, A. : The Formal Semantics of SDL-2000 - Status and Perspectives. In: *Computer Networks (Elsevier), Vol. 42, No. 3*, 2003, S. 343–358

[44] Gora, W. : *ASN.1 – Abstract Syntax Notation One*. FOSSIL-Verlag GmbH, 1998

[45] Gotzhein, R. : Model-driven by SDL – Improving the Quality of Networked Systems Development (Invited Paper). In: *Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE 2007), Marrakesh, Morocco*, 2007, S. 31–46

[46] Gotzhein, R. ; Grammes, R. ; Kuhn, T. : Specifying Input Port Bounds in SDL. In: *Proceedings of the 13th international SDL Forum conference on Design for dependable systems*. Springer-Verlag (SDL'07). – ISBN 3–540–74983–7, 978–3–540–74983–7, 101–116

[47] Gotzhein, R. ; Krämer, M. ; Litz, L. ; Chamaken, A. : Energy-aware System Design with SDL. In: [91], S. 19–33

[48] Gotzhein, R. ; Kuhn, T. : Decentralized Tick Synchronization for Multi-Hop Medium Slotting in Wireless Ad Hoc Networks Using Black Bursts. In: *5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks. SECON'08, San Francisco*, IEEE, June 2008, S. 422–431

[49] Gotzhein, R. ; Kuhn, T. : Black Burst Synchronization (BBS) - A Protocol for Deterministic Tick and Time Synchronization in Wireless Networks. In: *Computer Networks* 55 (2011), Nr. 13, S. 3015–3031

[50] Hespanha, J. ; Naghshtabrizi, P. ; Xu, Y. : A Survey of Recent Results in Networked Control Systems. In: *Proceedings of the IEEE* 95 (2007), jan., Nr. 1, S. 138 –162. `http://dx.doi.org/10.1109/JPROC.2006.887288`. – DOI 10.1109/JPROC.2006.887288. – ISSN 0018–9219

[51] Höhn, R. (Hrsg.) ; Höppner, S. (Hrsg.): *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. Berlin : Springer, 2008. `http://dx.doi.org/10.1007/978-3-540-30250-6`. `http://dx.doi.org/10.1007/978-3-540-30250-6`. – ISBN 978–3–540–30249–0

[52] IBM Corp.: *Rational SDL Suite*. `http://www-01.ibm.com/software/awdtools/sdlsuite/`, 2013

[53] IEEE: *Standard Test Access Port and Boundary-Scan Architecture (1149.1-2001).* 3 Park Avenue, New York, NY 10016-5997, USA : The Institute of Electrical and Electronics Engineers, Inc., 2001. `http://dx.doi.org/10.1109/IEEESTD.2001.92950`. `http://dx.doi.org/10.1109/IEEESTD.2001.92950`. – ISBN 0–7381–2944–5

[54] IEEE: *Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs).* IEEE Computer Society `http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf`

[55] International Telecommunication Union (ITU): *ITU-T Recommendation Z.100 (11/99): Specification and Description Language (SDL).* `http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf`, 1999

[56] International Telecommunication Union (ITU): *ITU-T Recommendation Z.100 Annex F: Formal Semantics Definition.* 2000

[57] International Telecommunication Union (ITU): *ITU-T Recommendation Z.100 (11/2007): Specification and Description Language (SDL).* `http://www.itu.int/rec/T-REC-Z.100-200711-I`, 2007

[58] Kinney, P. : *ZigBee Technology: Wireless Control that Simply Works.* `http://www.zigbee.org/resources`, Okt. 2003. – Whitepaper

[59] Koch, J. ; Fliege, I. : AmICom – Middleware Support for Ambient Communication. In: *2nd Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI) at the 9th International Conference on Ubiquitous Computing (UbiComp).* Innsbruck, Austria, Sept. 16–19 2007, S. 81–86

[60] Kopetz, H. ; Stankovic, J. A. (Hrsg.): *Real-Time Systems – Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1997

[61] Kraemer, F. A. (Hrsg.) ; Herrmann, P. (Hrsg.): *SAM 2010, 6th Workshop on System Analysis and Modelling.* Bd. *6598*. Springer, 2010 (LNCS)

[62] Krämer, M. ; Braun, T. ; Christmann, D. ; Gotzhein, R. : Real-Time Signaling in SDL. In: Ober, I. (Hrsg.) ; Ober, I. (Hrsg.): *15th International SDL Forum* Bd. 7083, Springer, 2011 (LNCS), S. 184–199

[63] Krämer, M. ; Engel, M. : New Bootloader for the Imote2 platform / TU Kaiserslautern. 2009 (373/09). – Forschungsbericht

[64] Krämer, M. ; Engel, M. : *Bootloader for Imote2*. `http://vs.cs.uni-kl.de/activities/boot/`, 2013

[65] Krämer, M. ; Geraldy, A. ; Marrón, P. J. (Hrsg.): Energy Measurements for MicaZ Node / Universität Stuttgart. 2006. – Forschungsbericht. – In 5. GI/ITG KuVS. Fachgespräch "Drahtlose Sensornetze", pages 61–68

[66] Krämer, M. ; Kuhn, T. : Development of a Runtime Environment for SDL Systems on Resource-Limited Platforms / Department of Computer Science, University of Kaiserslautern, Germany. 2005 (345/05). – Forschungsbericht

[67] Krämer, M. : *Entwicklung eines leichtgewichtigen Kommunikationsframeworks auf einem Mikrocontroller mit SDL*, TU Kaiserslautern, Diplomarbeit, 2005. `http://vs.informatik.uni-kl.de/publications/2005/Kr05/DA.pdf`

[68] Kuhn, T. ; Geraldy, A. ; Gotzhein, R. ; Rothländer, F. : *ns*+SDL – The Network Simulator for SDL Systems. In: Prinz, A. (Hrsg.) ; Reed, R. (Hrsg.) ; Reed, J. (Hrsg.): *SDL 2005*, Springer, 2005 (Lecture Notes in Computer Science (LNCS) 3530)

[69] Kuhn, T. : MacZ – a QoS MAC Layer for Ambient Intelligence Systems. In: Pfeifer, T. (Hrsg.) ; Schmidt, A. (Hrsg.) ; Woo, W. (Hrsg.) ; Doherty, G. (Hrsg.) ; Vernier, F. (Hrsg.) ; Delaney, K. (Hrsg.) ; Yerazunis, B. (Hrsg.) ; Chalmers, M. (Hrsg.) ; Kiniry, J. (Hrsg.): *Advances in Pervasive Computing 2006. Adjunct Proceedings of the 4th International Conference on Pervasive Computing*, Austrian Computer Society (OCG): Vienna, 2006, S. 244ff

[70] Kuhn, T. : *Model Driven Development of MacZ – A QoS Medium Access Control Layer for Ambient Intelligence Systems*, University of Kaiserslautern, Diss., 2009

[71] Kuhn, T. ; Gotzhein, R. ; Webel, C. : Model-Driven Development with SDL – Process, Tools, and Experiences. In: Nierstrasz, O. (Hrsg.) ; Whittle, J. (Hrsg.) ; Harel, D. (Hrsg.) ; Reggio, G. (Hrsg.): *MoDELS* Bd. 4199, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–45772–0, S. 83–97

[72] Kuhn, T. ; Irigon, J. I.: An Experimental Evaluation of Black Burst Transmissions. In: Zomaya, A. Y. (Hrsg.) ; Zeadally, S. (Hrsg.): *MOBIWAC*, Proceedings of the Fifth ACM International Workshop on Mobility Management & Wireless Access, MOBIWAC 2007, Chania, Crete Island, Greece, October 2007. – ISBN 978–1–59593–809–1, S. 163–167

[73] Larmouth, J. : *ASN.1 Complete*. Bd. Academic Press. Morgan Kaufmann, 2000

[74] Lian, F.-L. ; Moyne, J. ; Tilbury, D. : Network design consideration for distributed control systems. In: *Control Systems Technology, IEEE Transactions on* 10 (2002), mar, Nr. 2, S. 297 –307. `http://dx.doi.org/10.1109/87.987076`. – DOI 10.1109/87.987076. – ISSN 1063–6536

[75] Liu, X. ; Goldsmith, A. : Wireless network design for distributed control. In: *Decision and Control, 2004. CDC. 43rd IEEE Conference on* Bd. 3, 2004. – ISSN 0191–2216, S. 2823 – 2829 Vol.3

[76] Lunze, J. : *Regelungstechnik 1*. Springer London, Limited (Springer-Lehrbuch Bd. 1). `http://books.google.de/books?id=fTylHoV0DNIC`. – ISBN 9783540689096

[77] Lunze, J. : Der Regelkreis. In: *Regelungstechnik 1*. Springer Berlin Heidelberg, 2010 (Springer-Lehrbuch). – ISBN 978–3–642–13808–9

[78] Marvell Technology Group Ltd.: *Marvell PXA Family*. `http://www.marvell.com/products/processors/applications/pxa_family/`, 2013

[79] Mayer, O. : *Reihe Informatik*. Bd. 27: *Syntaxanalyse, 3. Auflage*. Bibliographisches Institut, 1986. – ISBN 3–411–03139–5

[80] MEMSIC Inc.: *Imote 2 Datasheet*. `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=134%3Aimote2`, 2013

[81] MEMSIC Inc.: *MICAz datasheet*. `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=148%3Amicaz`, 2013. – Revision B

[82] Microchip Technology Inc.: *PIC18F4321 Family Data Sheet*. `http://ww1.microchip.com/downloads/en/DeviceDoc/39689b.pdf`, 2006

[83] Microsoft: *Windows*. http://www.microsoft.com/Windows/, 2013

[84] Mitschele-Thiel, A. : *Engineering with SDL – Developing Performance-Critical Communication Systems*. John Wiley & Sons, 2000

[85] Networked Systems Group: *Homepage SPaSs tool*. `http://vs.cs.uni-kl.de/activities/spass/`, 2013

[86] N.N.: *libusb*. http://sourceforge.net/projects/libusb/develop, 2013

[87] Oracle Corporation: *Java*. `http://java.sun.com`. `http://java.sun.com`. Version: 2013

[88] Polastre, J. ; Hill, J. L. ; Culler, D. E.: Versatile Low Power Media Access for Wireless Sensor Networks. In: Stankovic, J. A. (Hrsg.) ; Arora, A. (Hrsg.) ; Govindan, R. (Hrsg.): *SenSys*, ACM, 2004. – ISBN 1–58113–879–2, S. 95–107

[89] PragmaDev SARL: *Real Time Developer Studio*. `http://www.pragmadev.com`, 2013

[90] Red Hat, Inc.: *cygwin Homepage*. `http://cygwin.com/`, 2013

[91] Reed, R. (Hrsg.) ; Bilgic, A. (Hrsg.) ; Gotzhein, R. (Hrsg.): *SDL 2009: Design for Motes and Mobiles, 14th International SDL Forum, Bochum, Germany, September 22–24, 2009, Proceedings*. Bd. *5719*. Springer, 2009 (LNCS). – ISBN 978–3–642–04553–0

[92] Royce, W. W.: Managing the development of large software systems: concepts and techniques. In: *Proceedings of the 9th international conference on Software Engineering*. IEEE Computer Society Press (ICSE '87). – ISBN 0–89791–216–0, 328–338

[93] Sanders, R. : Implementing from SDL. In: *Telektronikk 4.2000, Languages for Telecommunication Applications*. Telenor, 2000

[94] Schmelzer, T. : *Integration des Imote2 in das SDL Environment Framework (SEnF)*, Technische Universität Kaiserslautern, Fachbereich Informatik, Projektarbeit, 2007

[95] Schmelzer, T. : *Entwicklung und Integration von Services und Gatewayfunktionalität auf Basis von AmICoM*, Technische Universität Kaiserslautern, Fachbereich Informatik, Diplomarbeit, 2008

[96] Schmidt, D. ; Wehn, N. : A Review of Common Belief on Power Management and Power Consumption / Technische Universität Kaiserslautern. 2009. – White Paper

[97] Schurgers, C. ; Tsiatsis, V. ; Ganeriwal, S. ; Srivastava, M. B.: Topology management for sensor networks: exploiting latency and density. In: *MobiHoc*, ACM, 2002. – ISBN 1–58113–501–7, S. 135–145

[98] Shahabdeen, J. A.: Boot Loader Architecture / University of California. 2005. – Forschungsbericht

[99] Sun, Y. ; Du, S. ; Gurewitz, O. ; Johnson, D. B.: DW-MAC: a low latency, energy efficient demand-wakeup MAC protocol for wireless sensor networks. In: *MobiHoc*, 2008, S. 53–62

[100] Tanenbaum, A. S.: *Computer Networks*. 4. Prentice Hall International, Inc., 2003

[101] Tanenbaum, A. S.: *Moderne Betriebssysteme*. 3. aktualisierte Auflage. Pearson Studium, 2009. – ISBN 3827373425

[102] Technische Universität Kaiserslautern: *Ambient Systems – Technologien und Anwendungen*. `http://www.amsys-uni-kl.de`, 2011. – Projekt-Homepage

[103] Texas Instruments: *CC1000 datasheet*. `http://focus.ti.com/lit/ds/symlink/cc1000.pdf`, 2007. – Revision SWRS048A

[104] Texas Instruments: *CC2420 datasheet*. `http://focus.ti.com/lit/ds/symlink/cc2420.pdf`, 2007. – Revision SWRS041b

[105] The Linux Kernel Organization, Inc.: *The Linux Kernel*. `http://www.kernel.org`, 2013

[106] The ZigBee Alliance: *Homepage*. `http://www.zigbee.org/`, 2013

[107] UC Berkeley: *TinyOS Homepage*. `http://www.tinyos.net`, 2013

[108] USC Information Sciences Institute: *The Network Simulator – ns-2*. `http://www.isi.edu/nsnam/ns/`, 2013

[109] Varga, D. Szente ; Horvath, D. ; Rencz, M. : Ni-MH battery modelling for ambient intelligence applications. In: *Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS DTIP 2007*. TIMA Editions, 332-337. – Submitted on behalf of EDA Publishing Association (http://irevues.inist.fr/EDA-Publishing)

[110] Weber, C. : *Entwurf und Implementierung eines konfigurierbaren SDL Transpilers für eine C++ Laufzeitumgebung*, Technische Universität Kaiserslautern, Diplomarbeit, 2005

[111] Winkler, M. : *Spezifikation, Integration und Simulation eines Duty-Cycling-Protokolls für MacZ*, Technische Universität Kaiserslautern, Fachbereich Informatik, Diplomarbeit, 2009

[112] Yale School of Engineering & Applied Science: Power Modes and Energy Consumption for the iMote2 Sensor Node / Enalab. 2006. – Forschungsbericht. – `http://enaweb.eng.yale.edu/drupal/system/files/imote2_power.pdf`

[113] Ye, W. ; Heidemann, J. S. ; Estrin, D. : An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In: *INFOCOM*, 2002

Index

Lebenslauf

Ausbildung / Universitäre Laufbahn

09/1985–06/1998	Grundschule und Gutenberg-Gymnasium in Mainz Leistungskurse Mathematik, Physik und Erdkunde Abschluß Abitur
10/1999–04/2002	Informatikvordiplom an der TU Kaiserslautern
10/2000–04/2001	Vorlesungsbetreuung Rechnersysteme, TU Kaiserslautern
07/2001–11/2003	Wissenschaftliche Hilfskraft in der AG VLSI Entwurf und Architektur, TU Kaiserslautern
04/2002–10/2002	Vorlesungsbetreuung Rechnerarchitekturen, TU Kaiserslautern
04/2002–09/2005	Hauptdiplom an der TU Kaiserslautern mit der Vertiefung Eingebettete Systeme
08/2004	Projektarbeit „Eine Fallstudie zur Modellierung einer Gebäudesimulation“
09/2005	Diplomarbeit „Entwicklung eines leichtgewichtigen Kommunikationsframeworks auf einem Mikrocontroller mit SDL“
04/2013	Abgabe der Dissertation „Modellgetriebene Entwicklung von Kommunikationsprotokollen für drahtlos vernetzte Regelungssysteme“
07/2013	Disputation der Dissertation „Modellgetriebene Entwicklung von Kommunikationsprotokollen für drahtlos vernetzte Regelungssysteme“