AG VERNETZTE SYSTEME
FACHBEREICH INFORMATIK
AN DER TECHNISCHEN UNIVERSITÄT
KAISERSLAUTERN

BACHELOR-ARBEIT

ENTWICKLUNG EINER SCHNITTSTELLE FÜR DIE IMOTE2-PLATTFORM MIT PRAGMADEV RTDS UND BIPS

Alexander Mater

---

9. April 2014

Entwicklung einer Schnittstelle für die Imote2-Plattform mit PragmaDev RTDS und BiPS

Bachelor-Arbeit

Arbeitsgruppe Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Alexander Mater

Tag der Ausgabe : 15. November 2013
Tag der Abgabe : 9. April 2014

Betreuer : Prof. Dr. Reinhard Gotzhein und Tobias Braun

Ich erkläre hiermit, die vorliegende Bachelorarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den 9. April 2014

(Alexander Mater)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die *Specification and Description Language* (SDL) [Int12a] erlaubt die Spezifikation des Verhaltens verteilter Systeme unter Verwendung von erweiterten endlichen Automaten. Dabei bietet SDL eine intuitive grafische Repräsentation für modellgetriebene Entwicklung. Durch die formale Definition von Syntax und Semantik ist es möglich, automatisiert SDL-Spezifikationen in Programmiersprachen zu transformieren. Dies wird durch Model-Driven Development-Werkzeuge wie *IBM Rational SDL Suite* [IBM] (vorher *Telelogic Tau SDL Suite*) und *PragmaDev Real Time Developer Studio* (RTDS) [Praa] unterstützt. Hierdurch ist es weiterhin möglich, aus einer SDL-Spezifikation ausführbare Programme zu erzeugen.

Mit SDL-Paketen können häufig benötigte Funktionalitäten gekapselt werden. Damit können beispielsweise komplexe Kommunikationsprotokolle in SDL-Paketen spezifiziert werden und in verschiedenen Projekten verwendet werden. Hiermit ermöglichen SDL-Pakete Wiederverwendung und modulares Systemdesign, wodurch die Entwicklung und Wartung von SDL-Systemen erleichtert wird.

Bisher wurde von der AG Vernetzte Systeme (AG-VS) [AG] die *IBM Rational SDL Suite* als SDL-Editor für SDL-Model-driven Development (SDL-MDD) [Got07] verwendet. Zusätzlich wurde mit *ConTraST* [FGW06] ein Codegenerator und mit *SdlRE* eine Laufzeitumgebung entwickelt, mit der automatisiert ausführbare Applikation erstellt werden können. Für die automatische Anbindung an unterschiedliche Betriebssysteme und Hardwareplattformen wurde mit dem *SDL Environment Framework* (SEnF) [FGJ+05] ein Sortiment an generischen Umgebungsschnittstellen erstellt. *ConTraST* und *SdlRE* sind nicht speziell auf eingebettete Entwicklung ausgelegt und daher nicht geeignet, da sie zu ineffizient und schwergewichtig sind. Um speziell die Anforderungen eingebetteter Systeme zu berücksichtigen, wurde mit dem *Black Burst-integrated Protocol Stack* (BiPS) [Eng13] ein Protokoll-Stack speziell für eingebettete Systeme entwickelt, der bisher keine Schnittstelle zu SDL-Systemen aufweist. Dies soll im Zuge dieser Arbeit erfolgen. Da außerdem die Produktpflege der *IBM Rational SDL Suite* eingestellt wurde, hat sich die AG Vernetzte Systeme entschieden, eine Alternative zu verwenden. Als Basis für die neue Entwicklung wurde RTDS ausgewählt. RTDS ersetzt *ConTraST* als Codegenerator und *SdlRE* als Laufzeitumgebung. Die bereitgestellten Betriebssystemvorlagen von RTDS bilden die Grundlage der neuen Laufzeitumgebung. Die notwendigen Erweiterungen für die Integration von RTDS in BiPS werden im Zuge dieser Arbeit entwickelt.

Die Arbeit ist wie folgt aufgebaut: Zuerst werden in Kapitel 2 kurz die Grundlagen für diese Arbeit vorgestellt. Diese umfassen SDL, RTDS, BiPS und den in der AG verwendeten Sensorknoten Imote2. In Kapitel 3 wird die Codegenerierung durch RTDS untersucht und in Kapitel 4 werden die Eigenschaften der generierten Systeme analysiert sowie erste notwendige Anpassungen durchgeführt, um mit den Einschränkungen umzugehen. Kapitel 5 beschreibt die Realisierung der RTDS-BiPS-Schnittstelle. Danach wird in Kapitel 6 die Erzeugung einer aus SDL erzeugten Applikation für BiPS beschrieben. Anschließend werden in Kapitel 7 mit einer ausführlichen Demonstration die Möglichkeiten der Integration evaluiert. Zuletzt fasst Kapitel 8 die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf zukünftige Entwicklungen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die notwendigen Grundlagen beschrieben, die für den Rest der Arbeit benötigt werden. Dazu werden die *Specification and Description Language*, das *Real Time Developer Studio*, der Sensorknoten *Imote2* und der *Black Burst-integrated Protocol Stack* vorgestellt.

2.1 Specification and Description Language - SDL

Die *Specification and Description Language* (SDL) ist eine in Z.100 [Int12a] und Erweiterungen definierte Modellierungssprache. SDL ist ein Standard, der von der *International Telecommunication Union* (ITU) gepflegt wird. Ziel bei der Entwicklung von SDL ist es, eine Modellierungssprache mit formaler Syntax und Semantik für den Entwurf von Telekommunikationssystemen zu bieten. Die SDL-Spezifikation von Verhalten wird mit erweiterten endlichen Automaten mit asynchroner Kommunikation beschrieben. Durch die Einführung von Timern wird auch ein Konzept der Zeit realisiert. Die erste Edition der Z.100 wurde bereits im Jahr 1984 veröffentlicht. Die Entwicklung von SDL wird noch immer weitergeführt und die letzte große Erweiterung stellt SDL-2010 aus dem Jahr 2011 dar.

SDL-Systeme sind hierarchisch aufgebaut und bestehen aus Blöcken und Prozessen. Der Inhalt eines Blocks oder Prozesses wird in einem Diagramm dargestellt und die ausführliche Beschreibung nur mit einer Referenz angegeben. Unter der Voraussetzung einer guten Strukturierung ist es damit möglich, auch komplexe SDL-Systeme überschaubar zu halten. Ein weiterer Vorteil dieser hierarchischen Systembeschreibung ist, dass einzelne Blöcke und Prozesse angepasst werden können ohne das gesamte System neu spezifizieren zu müssen. Dies ermöglicht ebenfalls die Wiederverwendung von Blöcken und Prozessen in anderen Systemen. Dazu können SDL-Pakete verwendet werden. Diese erlauben die Modularisierung von Systemteilen, wodurch eine Wiederverwendung in weiteren Projekten erleichtert wird. SDL kann für modellgetriebene Entwicklung [Got07] eingesetzt werden, in welcher SDL-Modelle die zentralen Bausteine bilden. Hier wird mit SDL eine plattformunabhängige Spezifikation erstellt wird. Diese kann mit passenden Werkzeugen (z.B. RTDS) in ein plattformspezifisches Modell weiterentwickelt und anschließend automatisiert in eine Applikation transformiert werden.

SDL bietet zwei Repräsentationen mit gleicher Mächtigkeit, SDL-PR und SDL-GR. Die *textuelle Repräsentation* (SDL-PR) eignet sich für die maschinengestütze Verarbeitung mit geeigneten Tools. Die *graphische Repräsentation* (SDL-GR) ist besser geeignet, um damit manuell zu arbeiten und Systeme zu spezifizieren. Um dies zu demonstrieren, wurde ein einfaches Beispiel für ein SDL-System vorbereitet. Dieses SDL-System besteht aus dem Block *PingPong* (Abbildung 2.1) der zwei Prozesse, *Ping* und *Pong*, enthält. Die zwei SDL-Prozesse kommunizieren ununterbrochen miteinander durch den Austausch der gleichnamigen Signale. Der Prozess *Ping* zählt, wie häufig das Signal *Ping* gesendet wurde.

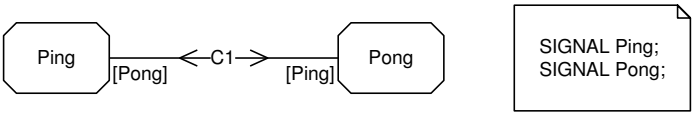

Abbildung 2.1: Block PingPong mit Signaldefinition

Das Verhalten von SDL-Prozessen wird durch erweiterte endliche Automaten beschrieben. Zusätzlich zu Zuständen und Zustandsübergängen sind Datenstrukturen und Variablen erlaubt. Diese werden während Zustandsübergängen manipuliert. Abbildung 2.2 und 2.3 zeigen exemplarisch die Spezifikation des Verhaltens von SDL-Prozessen.

Der SDL-Prozess *Ping* (Abbildung 2.2) sendet über den Kanal *C1* das Signal *Ping* und wechselt in den *idle*-Zustand. Der Prozess *Pong* (Abbildung 2.3) empfängt dieses Signal, antwortet darauf mit dem Signal *Pong* und verbleibt in seinem *idle*-Zustand. Das Signal wird über den Kanal *C1* übertragen und vom Prozess *Ping* empfangen. Dieser inkrementiert die Integer-Variable *counter* und sendet erneut ein Signal *Ping*. Dieser Ablauf wiederholt sich solange das SDL-System ausgeführt wird.

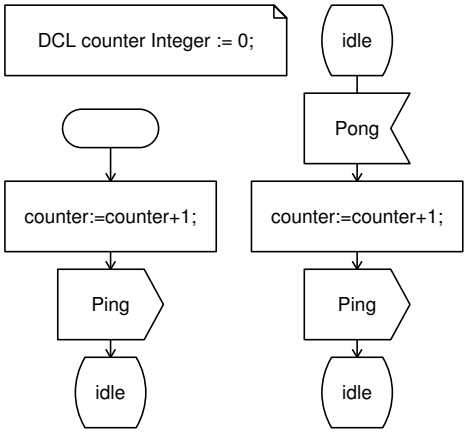

Abbildung 2.2: Prozess Ping

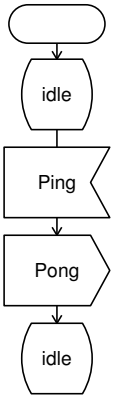

Abbildung 2.3: Prozess Pong

Das gleiche System sieht in SDL-PR wie folgt aus:

```
1 SYSTEM PingPongDemo;
2
3 BLOCK PingPong;
4 SIGNAL Ping;
5 SIGNAL Pong;
6
```

```
7  SIGNALROUTE C1
8  FROM Pong TO Ping WITH Pong;
9  FROM Ping TO Pong WITH Ping;
10
11 PROCESS Ping;
12 DCL counter Integer := 0;
13 START ;
14 TASK counter:=counter+1;
15 OUTPUT Ping;
16 NEXTSTATE idle;
17
18 STATE idle;
19 INPUT Pong;
20 TASK counter:=counter+1;
21 OUTPUT Ping;
22 NEXTSTATE idle;
23 ENDSTATE;
24 ENDPROCESS;
25
26 PROCESS Pong;
27 START ;
28 NEXTSTATE idle;
29
30 STATE idle;
31 INPUT Ping;
32 OUTPUT Pong;
33 NEXTSTATE idle;
34 ENDSTATE;
35 ENDPROCESS;
36
37 ENDBLOCK;
38
39 ENDSYSTEM;
```

Listing 2.1: SDL-PR-Code des SDL-Systems

Da das gezeigte SDL-System nicht mit der Außenwelt kommuniziert, wird es als geschlossenes SDL-System bezeichnet. SDL bietet das Konzept des SDL-Environments. Dadurch wird in einer SDL-Spezifikation die Außenwelt repräsentiert. Signale können in SDL-Systemen mit dem SDL-Environment ausgetauscht werden. Diese SDL-Systeme werden als offenes SDL-System bezeichnet. Die Behandlung der Signale im SDL-Environment wird nicht in SDL spezifiziert, sondern muss durch eine plattformspezifische Implementierung bereitgestellt werden, in welcher z.B. Treiber oder das zugrunde liegende Betriebssystem angesprochen wird.

2.2 Real Time Developer Studio - RTDS

Das *Real Time Developer Studio* (RTDS) [Praa] wurde als Entwicklungsumgebung für die von PragmaDev entwickelte *SDL Real Time* (SDL-RT) [SDL13] Erweiterung für SDL erstellt. Ziel von SDL-RT ist es, die Nachteile von SDL zu beseitigen. Die Entwicklung von SDL-RT folgte zunächst zwei Grund-Prinzipien:

- Ersetzen der SDL-Datentypen durch C-Datentypen
- Unterstützung von Semaphoren in SDL-Diagrammen

Das Ergebnis beschreiben die SDL-RT-Entwickler als eine objektorientierte, grafische Sprache auf Basis von Standard-Sprachen mit Unterstützung von klassischen Echtzeit-Konzepten. SDL-RT und RTDS befinden sich beide in aktiver Entwicklung.

Wir verwenden RTDS nicht wegen der SDL-RT-Erweiterung, sondern benutzen in dieser Arbeit ausschließlich die SDL-Unterstützung von RTDS. Die für uns wichtigen Bestandteile von RTDS sind der SDL-Editor und der Codegenerator, um C/C++ Code zu generieren. Der SDL-Editor unterstützt den Entwurf von SDL-Systemen gemäß Z.100 in der grafischen Repräsentation. Die Abbildungen 2.1, 2.2 und 2.3 sind Beispiele für Diagramme, die mit RTDS erzeugt wurden.

Der RTDS-Codegenerator übersetzt die grafische Darstellung in C/C++ Code. Dabei werden unterschiedliche Varianten der Codegenerierung unterstützt. Die generierten Systeme können mit speziellen Templates an verschiedene Echtzeitbetriebssysteme (RTOS) angepasst werden. RTDS unterstützt *FreeRTOS*, *OSE Epsilon*, *Windows* und *POSIX*-konforme Betriebssysteme. Es besteht auch die Möglichkeit, Templates ohne RTOS zu verwenden. Diese benötigen zusätzlich noch einen separaten Scheduler für die generierten Systemkomponenten. Dieser Scheduler kann manuell erstellt werden. Alternativ wird von RTDS ein simpler Scheduler angeboten. Für unsere Schnittstelle passen wir das Template ohne RTOS an (*rtosless*) und erweitern den in RTDS vorhanden Scheduler (*cppscheduler*).

2.3 Imote2

Der Imote2 [Cro07] soll als Hardware-Plattform für die SDL-Systeme dienen. Die Kommunikation erfolgt über den speziell entwickelten Protokollstack BiPS (siehe Kapitel 2.4). Der Sensorknoten Imote2 (siehe Abbildung 2.4) setzt sich u.A. aus folgenden Bestandteilen zusammen. Die auf dem Imote2 verbaute CPU ist ein PXA271 XScale Prozessor. Dieser bietet energiesparende Betriebsmodi (Sleep und Deep Sleep) und Dynamic Voltage Scaling zum Regulieren der CPU-Frequenz. Damit kann die Frequenz in Stufen zwischen 13 MHz und 416 MHz eingestellt werden. Der PXA271 ist ausgestattet mit 256 KB SRAM, 32 MB SDRAM und 32 MB Flashspeicher. Als Anschlüsse stehen unter anderem drei High-Speed UART, I2C, GPIO und ein Mini-USB zur Verfügung. Der CC2420 [Tex13] ist die drahtlose Schnittstelle des Imote2. Diese bietet eine maximale Übertragungsrate von 250 kb/s mit 16 Kanäle auf dem 2.4 GHz-Band. Der CC2420 ist IEEE 802.15.4-konform [IEE03].

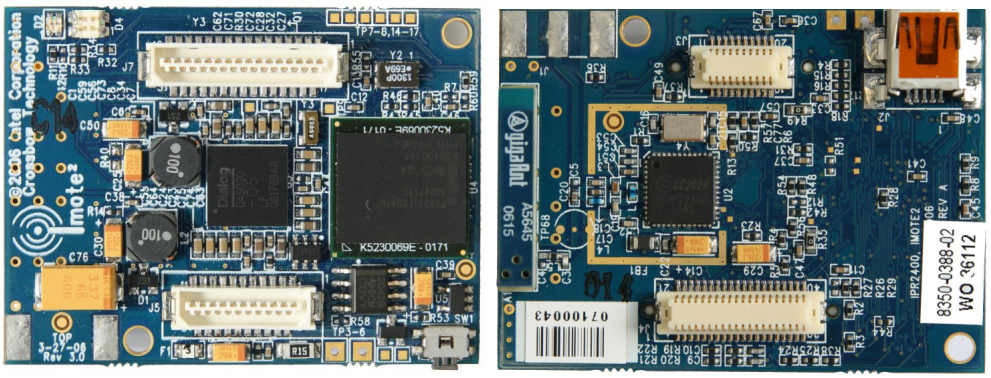

Abbildung 2.4: Vorder- und Rückseite des Imote2 [Eng13, Abb. 3.1]

2.4 Black Burst-integrated Protocol Stack - BiPS

Die Grundlagen des *Black Burst-integrated Protocol Stacks* (BiPS) wurden im Zuge der Masterarbeit „Optimierung und Evaluation Black Burst-basierter Protokolle unter Verwendung der Imote2-Plattform“ [Eng13] entwickelt. BiPS ist speziell für die Anforderungen eingebetteter Systeme ausgelegt und auf die Imote2-Plattform zugeschnitten. Es wird zurzeit aktiv von der AG Vernetzte Systeme weiterentwickelt.

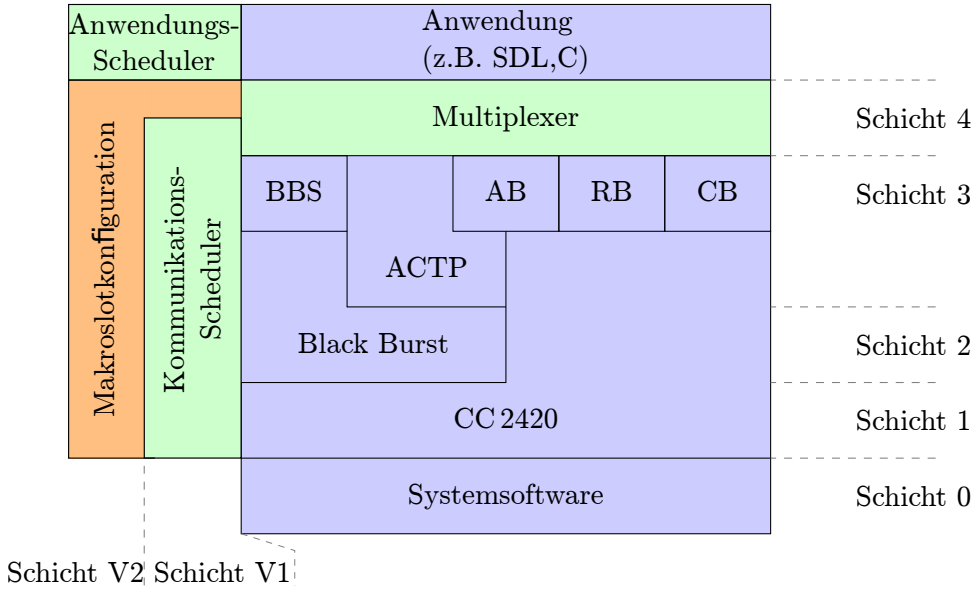

Abbildung 2.5: Übersicht der BiPS-Struktur

Abbildung 2.5 zeigt einen Überblick über den Aufbau des Protocol Stacks. Der Protocol Stack ist in einer Schichtenarchitektur aufgebaut und bietet low-level Systemsoftware und Bereitstellung von Schnittstellen zur Ansteuerung der Peripherie für Protokolle und Anwendungen. Auf dieser Basis stellt BiPS deterministische Protokolle auf Grundlage des namensgebenden *Black Bursts* bereit. Dieser wird von M. Engel als „eine Kommunikationsprimitive, bei der Übertragungen kollisionsresistent sind“ [Eng13, Kapitel 2.2 Seite 4] beschrieben. Wie in Abbildung 2.5 zu sehen, werden unter anderem das *Arbitrating/Cooperative Transfer Protocol* (ACTP) [CGR12] und das *Black Burst Synchronization* (BBS) [GK08, GK11] Protokoll unterstützt. Diese können vom BiPS-Kommunikationsscheduler verwendet werden, um das Medium zu verwalten. Um eine hohe Genauigkeit für die Einhaltung von Zeitslots zu erreichen, läuft der BiPS-Kommunikationsscheduler im Interruptkontext.

Anwendungen ohne strikte Echtzeitanforderungen werden im Nicht-Interruptkontext ausgeführt. Dafür wird der Anwendungs-Scheduler bereitgestellt, der ein Timer- und Eventsystem für Anwendungen bietet. Damit ist es möglich, Hardwaretimer zu verwenden und Anwendungen schlafen zu legen. Durch Events kann die Ausführung weniger zeitkritischer Anwendungen später im Nicht-Interruptkontext fortgesetzt werden.

Die SDL-Integration ist eine Erweiterung, um den Entwurf von nicht zeitkritischen Anwendungen für den Protocol Stack zu erleichtern, indem die Systeme in SDL spe-

zifiziert werden. Dabei soll die SDL-Integration die Generierung von Anwendungen für den Protocol Stack ermöglichen. Mögliche Anwendungsbereiche umfassen die Implementierung von Routing- oder Clusteringprotokollen, da diese im Allgemeinen keine harten Echtzeitanforderungen haben.

Kapitel 3

Umsetzung des SDL-Systems als Quellcode

Dieses Kapitel beschäftigt sich mit der Transformation einer SDL-Spezifikation in C/C++-Code durch den RTDS Codegenerator. Der Schwerpunkt der Betrachtung liegt auf dem *rtosless*-Template, der Codegenerierung ohne RTOS, welche den bereitgestellten RTDS-Scheduler nutzt. Das *rtosless*-Template wurde ausgewählt, weil zu Beginn der Arbeit noch kein *FreeRTOS* auf dem Imote2 ausgeführt wurde. Zusätzlich ermöglichen die geringen Vorraussetzungen für das *rtosless*-Template, dass damit für Testzwecke neben Imote2-BiPS-Anwendungen auch Linux-Anwendungen erzeugt werden können. Zuerst wird die Struktur des Quellcodes beschrieben, der vom Codegenerator erzeugt wird. Danach werden die generierten Dateien und ihre Aufgaben geschildert. Zuletzt erfolgt die Betrachtung der Signal- und Timerverarbeitung.

3.1 Überblick über das RTDS-System

Der Quellcode, der von dem RTDS-Codegenerator erzeugt wird, ist der Startpunkt für die BiPS-Integration. Da der Quelltext des Codegenerators nicht zugänglich ist, haben wir keinen Einfluss auf die Codegenerierung. Der RTDS-Codegenerator verwendet speziell für unterschiedliche Betriebssysteme erstellte Templates. Durch diese angepassten Templates wird z.B. die Aufgabe des Schedulings von Prozessinstanzen (siehe Abbildung 3.1) an das Betriebssystem abgegeben. Sollte kein Betriebssystem vorhanden sein, werden in dem *rtosless*-Template die notwendigen Funktionalitäten des Betriebssystems rudimentär bereitgestellt.

Der Codegenerator bietet die Möglichkeit C oder C++ zu erzeugen. Diese RTDS-Templates werden mit dynamisch erzeugten Quellcode ergänzt, der die Spezifikation des SDL-Systems enthält. Dadurch erfüllt das RTDS-Template die Rolle der Laufzeitumgebung für die spezifizierten SDL-Prozesse. Die SDL-Prozesse rufen während ihrer Ausführung die durch das RTDS-Template bereitgestellten Makros für Verarbeitungsschritte auf.

Abbildung 3.1 zeigt beispielhaft den Aufbau eines RTDS-Systems. Den Kern bildet das SDL-System, dass durch die Spezifikation beschrieben wird und vom Co-

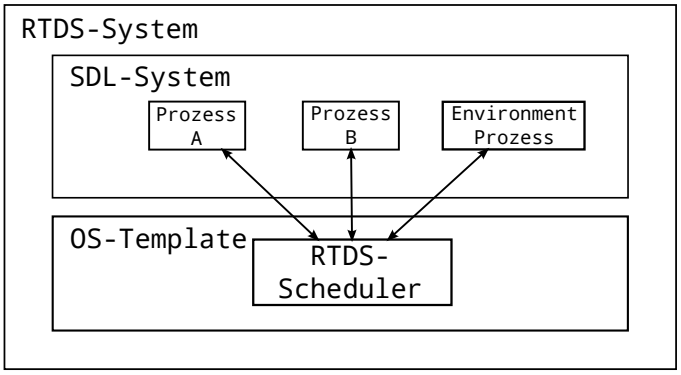

Abbildung 3.1: Struktur des RTDS-Systems

degenerator zu Quellcode transformiert wurde. Alle SDL-Prozesse und das SDL-Environment erhalten eine Prozessinstanz, die vom RTDS-Scheduler verwaltet wird. Der RTDS-Scheduler verwaltet die Signale (siehe Kapitel 3.3) und das *rtosless*-Template eine globale Timerliste (siehe Kapitel 3.4). Die Instanzen der Prozesse kommunizieren nur indirekt über den RTDS-Scheduler miteinander.

Zusätzlich werden vom Codegenerator noch weitere Dateien (siehe Kapitel 3.2) erzeugt, die für die Compilierung und Initialisierung der Prozessinstanzen zuständig sind. Dadurch kann ein RTDS-System generiert und zu einem lauffähigen Programm kompiliert werden, wenn nicht die vereinzelten Fehler (siehe Kapitel 4.4) bzw. Eigenheiten von RTDS (siehe UML Deployment Diagramm in Abbildung 6.1) auftreten.

3.2 Analyse der generierten Dateien

Der Codegenerator erzeugt eine Menge unterschiedlicher Dateien. Dabei lassen sich nicht alle auf ein Element der SDL-Spezifikation zurückführen. Dieses Kapitel basiert auf dem Reference Manual [Prab, Kapitel 9.2.2.] Die Abbildung 3.2 zeigt, welche Dateien für die einzelnen SDL-Diagramme generiert werden.

Für jedes Diagramm (System, Block und Prozess) wird eine C-Header-Datei generiert. Die Header-Datei des übergeordneten Diagramms wird eingebunden. Diagramme die Verhalten spezifizieren erhalten zusätzlich eine C-Quelltext-Datei, die dieses Verhalten umsetzt. Zusätzlich werden Dateien erzeugt, die keinem Diagramm entsprechen. Diese sind in jedem RTDS-System enthalten und beinhalten Funktionalitäten der Laufzeitumgebung:

- RTDS_Start.c: Beginn der Ausführung des RTDS-Systems. Enthält die main-Methode in der die notwendigen Initialisierungen durchgeführt werden.
- RTDS_messages.h: Enthält Signal-Makros zum versenden und empfangen der Signale.
- RTDS_gen.h: Enthält generierte Konstanten für den generierten Quelltext. Dazu gehören die Identifikationsnummern (IDs) für alle Signale, Timer, Zustände und Prozesse im spezifizierten SDL-System.

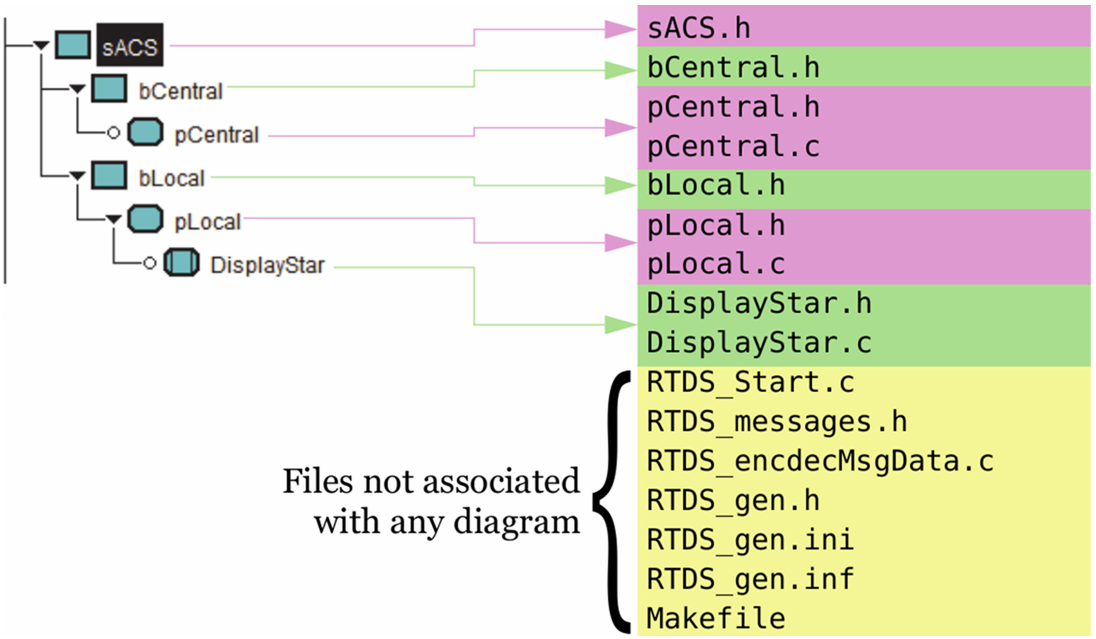

Abbildung 3.2: Generierte Dateien für SDL-Diagramme [Prab, Kapitel 9.2.2]

- Makefile: Das Makefile für das gesamte RTDS-System, dass aus der SDL-Spezifikation generiert wurde.

Die restlichen generierten Dateien, RTDS_encdecMsgData.c, RTDS_gen.ini und RTDS_gen.inf, sind für diese Arbeit nicht relevant und werden nur zur Vollständigkeit angegeben.

3.3 Verarbeitung von Signalen

Bei der RTDS-Codegenerierung werden alle SDL-Kanäle aufgelöst. Die Adressierung eines RTDS-Signals, der Repräsentation des in SDL spezifizierten Signals im generierten RTDS-System, erfolgt durch die Verwendung von RTDS-Prozess-IDs und RTDS-Prozess-Namen (siehe Prozess-Name „Pong“ und Prozess-ID RTDS_process_Pong in Abbildung 3.3). Während der weiteren Verarbeitung von `RTDS_MSG_QUEUE_SEND_TO_NAME` mit dem *rtosless*-Template wird der Prozess-Name verworfen und nur die Prozess-ID (Integer-Wert) verwendet. Das RTDS-Signal wird in der Implementierung auch als RTDS-Message bezeichnet (siehe Signal Ping und RTDS_message_Ping in Abbildung 3.3).

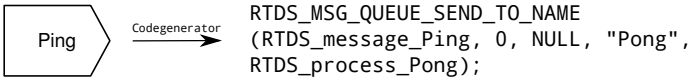

Abbildung 3.3: Signalsymbol und generierter C++-Code

Das RTDS-Signal wird erzeugt und in eine globale Signalqueue des RTDS-Schedulers eingefügt. Ein RTDS-Signal besteht aus der in RTDS_gen.h definierten RTDS-Signal-ID und notwendigen Datenstrukturen, die in RTDS_message.h definiert sind.

Die RTDS-Signalqueue arbeitet nach dem FIFO-Prinzip und wird kontinuierlich vom RTDS-Scheduler abgearbeitet. Wird ein RTDS-Signal an einen Prozess zugestellt, wird die Transition ausgeführt, die für das SDL-Signal in diesem Zustand spezifiziert wurde. Für SDL-Signale, die mit dem SAVE-Symbol im SDL-Prozess zurückgestellt werden, hat jeder RTDS-Prozess eine eigene Savequeue. Diese Savequeue wird an den Anfang der Signalqueue kopiert, wenn der RTDS-Prozess seinen Zustand wechselt. Signale die nicht im aktuellen Zustand spezifiziert sind, werden implizit konsumiert.

Das SDL-Environment wird als eigenständiger RTDS-Prozess in der Datei RTDS_Env.c implementiert. Alle SDL-Signale an das SDL-Environment werden diesem RTDS-Prozess zugestellt und in der Standardimplementierung durch den RTDS-Prozess verworfen.

3.4 Verarbeitung von Timern

RTDS akzeptiert zwei Varianten von SET-Befehlen für Timer, die jeweils nur mit relativen Delays arbeiten. Variante 1 gibt den Delay im SET-Befehl explizit an und erwartet zwei Parameter. Als erster Parameter wird NOW+Delay akzeptiert. Der zweite Parameter ist der Timername. Der Timer *Timername* muss in RTDS, im Gegensatz zum SDL-Standard, vorher nicht explizit deklariert werden, sondern kann auch implizit bei der Codegenerierung deklariert werden. Auch wenn die Syntax des Timers nach einem absoluten Timer aussieht, arbeitet diese erste Variante nur mit relativen Verzögerungen, da nur das Delay ausgewertet wird. Die zweite Variante setzt eine explizite Deklaration des Timers mit einem Standard-Delay voraus. In diesem Fall wird nur der Timername als Parameter vom SET-Befehl benötigt.

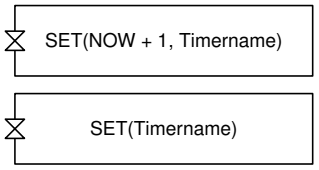

Abbildung 3.4: Beide Varianten von RTDS-SDL-SET-Befehlen

Beide Varianten werden zum RTDS_setTimer-Befehl übersetzt. Dieser Befehl sortiert den Timer anhand des Delays in die globale Timerliste des *rtosless*-Templates ein. Diese Liste wird vom RTDS-Scheduler kontrolliert und das Timersignal direkt ausgeführt. Es wird in der Standardimplementierung kein Timersignal in die Signalqueue eingefügt. Die Problematik der Timerausführung mit dem RTDS-Scheduler wird in Kapitel 4.1 und 4.3 genauer betrachtet.

Kapitel 4

Plattformunabhängige Einschränkungen des RTDS-Systems

Mit der Wahl von RTDS als SDL-Editor und Codegenerator ergeben sich direkt einige plattformunabhängige Einschränkungen, die betrachtet werden müssen. Diese Einschränkungen sind unvollständige Implementierungen des SDL-Standard. Im Verlauf der Arbeit wurden ebenfalls einige Fehler entdeckt, die in Kapitel 4.4 untersucht werden. Der Schwerpunkt der Analyse liegt, wie in Kapitel 3 auf dem *rtosless*-Template. In den Kapiteln 4.1 - 4.4 wird zunächst nur das unveränderte RTDS-System betrachtet. Im Kapitel 4.5 werden für unsere Anwendungen notwendige plattformunabhängige Änderungen beschrieben.

4.1 Scheduling mit dem RTDS-Scheduler

Um Scheduling mit dem *rtosless*-Template zu ermöglichen wird ein rudimentärer Scheduler durch das *cppscheduler*-Template angeboten. Dieser kennt keine absolute Zeit und verwendet SystemTicks als einziges Zeitmaß.

Der Ablauf der Scheduler run()-Methode, ist in der cppscheduler/RTDS_Scheduler.cpp Datei implementiert und ist in Abbildung 4.1 dargestellt. Um die Darstellung übersichtlicher zu gestalten, werden nur die relevanten Schritte dargestellt. Wie in Abbildung 4.1 zu sehen, werden SDL-Signale abgearbeitet, bis keine mehr in der Queue des Schedulers vorhanden sind. Erst danach wird ein SystemTick erzeugt. Im Anschluss an diesen SystemTick schreitet die Zeit voran und abgelaufene Timer werden gesucht. Wenn ein Timer abläuft, wird das Timersignal erzeugt und direkt durch den RTDS-Scheduler verarbeitet. Dieses Vorgehen hat einen unerwünschten Nebeneffekt, dass bei ununterbrochene Signal-Austausch keine SystemTicks erzeugt werden und die Zeit stehen bleibt. Folglich kann auch kein Timer ablaufen.

Der Prozess in Abbildung 4.2 ist eine Abwandlung des Beispiels in Abbildung 2.2. Der Prozess *Ping* enthält zusätzlich einen Timer *Alarm*. Dieser ist so definiert, dass nach 100 Zeiteinheiten der Alarm auslöst und den Austausch von *Ping*- und *Pong*-Signalen beendet, indem der *Ping*-Prozess terminiert. Dadurch, dass Zeit erst aktualisiert wird, wenn keine Signale mehr in der Queue sind, wird in diesem Beispiel der

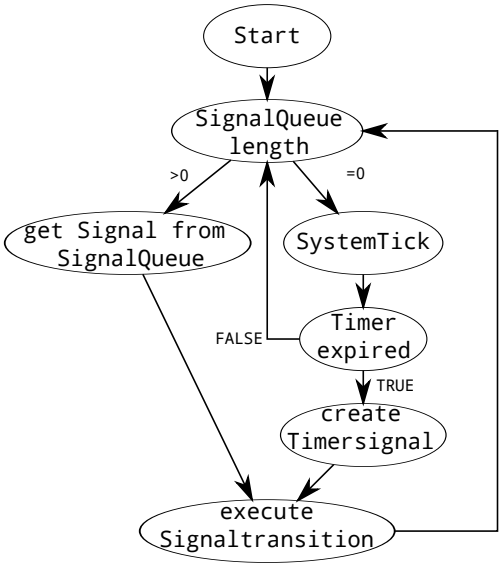

Abbildung 4.1: Ablauf der RTDS-Scheduler run()-Methode

Timer nie ausgelöst. Dies entspricht nicht dem gewünschten und im SDL-Standard definierten Verhalten. In Kapitel 4.5.2 wird daher ein neuer Ablauf für den Scheduler vorgeschlagen, der dieses Verhalten korrigiert.

4.2 Einbinden von SDL-Paketen

Der USE-Befehl wird in SDL verwendet, um in einem System- oder Blockdiagramm anzugeben, welche SDL-Pakete verwendet werden. Dies erlaubt es Systemteile, z.B. SDL-Prozess-Typ- oder SDL-Block-Typ-Spezifikationen, in zentralen Paketen zu definieren und in unterschiedlichen Projekten wieder zu verwenden. RTDS erlaubt jedoch in jedem Diagramm nur eine USE-Anweisung (RTDS User Manual [Prac, Kapitel 7.1.10]). Diese Einschränkung wirkt sich negativ auf die Struktur eines SDL-Systems aus. Erschwerend für die Entwicklung ist, dass der Syntax-Check von RTDS keine Warnung bei mehreren USE-Anweisungen ausgibt. Dadurch ist es möglich das die Codegenerierung erfolgreich verläuft und erst beim Compilieren eine fehlende Deklaration zu einer Fehlermeldung führt. Diese Einschränkung lässt sich zwar mit einer anderen Strukturierung der Pakete umgehen, ist aber nicht benutzerfreundlich. Im Allgemeinen ist es notwendig, alle verwendeten Deklaration in einem SDL-Paket zusammenzufassen (siehe Include-Paket in Abbildung 7.1).

4.3 Realisierung von Timern

SDL erlaubt die Definition von Timern mit absoluten und relativen Werten. Hierfür definiert der SDL-Standard in Z.101 [Int12b, Kapitel 11.15 Seite 65] den SET-Befehl für einen Timer als `SET(Time-Expression, Timername)`. Dabei kann die Time-Expression auf drei verschiedene Möglichkeiten zusammengesetzt werden. Die erste, rein absolute Möglichkeit, ist die Übergabe eines beliebigen Werts vom Typ

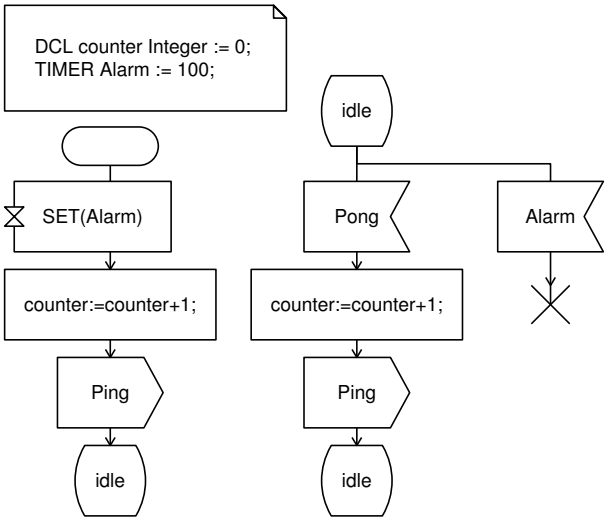

Abbildung 4.2: Modifizierter Ping-Prozess mit Timer zur Veranschaulichung von Problemen mit RTDS-Scheduler

Time. Um relative Werte zu verwenden, muss eine Referenzzeit angegeben werden. Daher besteht die zweite Möglichkeit darin, einen Time-Wert mit einem Delay anzugeben. Dies wird sehr häufig in Kombination mit NOW, der aktuellen Zeit des SDL Systems, und dem gewünschten Delay verwendet. Anstelle von NOW kann auch jeder beliebige Zeitpunkt als Referenzzeit verwendet werden. Die dritte Möglichkeit ist die Time-Expression auszulassen und bei der Definition des Timers einen Delay festzulegen (siehe Abbildung 4.2). Dies ist gleichbedeutend zu einem `SET(NOW+Delay, Timername)`, wobei Delay das bei der Deklaration verwendete Delay ist.

In RTDS sind Timer allerdings nicht dem Standard entsprechend umgesetzt. RTDS erlaubt nur die beiden letzten Varianten `SET(NOW+X,Timername)` und `SET(Timername)`. Es ist es nicht möglich einen absoluten Zeitwert zu nutzen, weil alle Timer relativ zum aktuellen Zeitpunkt gesetzt werden. Um eine höhere Genauigkeit zu erzielen und strikt periodische Timer zu realisieren, benötigen wir jedoch spezielle Referenzzeitpunkte [BCG09, Kapitel 4.6, Seite 47]. Daher brauchen wir eine Möglichkeit absolute Timer zu verwenden. Dazu wird in Kapitel 4.5.1 eine Änderung beschrieben, die absolute Timer ermöglicht.

Die beiden in RTDS erlaubten Varianten des SET-Befehls werden im Quelltext in einer unerwarteten Form abgebildet. Wie in Abbildung 4.3 zu sehen, wird bei `SET(NOW+delay, Timername)` der „NOW+“-Teil komplett ignoriert.

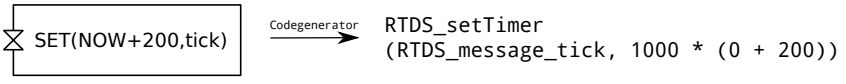

Abbildung 4.3: Timersymbol mit entsprechendem C++-Code

Dies liegt daran, dass alle SET-Befehle als ein Timer mit dem Delay relativ zur Verarbeitungszeit verarbeitet werden. Dieses Verhalten ist in dem RTDS Reference Manual [Prab, Kapitel 8.2.7.21] dokumentiert.

Zusätzlich sieht man in der Abbildung 4.3, dass das Delay mit dem Faktor 1000 multipliziert wird. Dies führt zusammen mit der Definition des Zeit-Datentyps als signed

integer, wie ihn RTDS bei der Codegenerierung erzeugt, zu einem Laufzeitproblem. Der SDL-Standard definiert keine spezifische Zeiteinheit. Daher ist die Abbildung von Zeit als int zunächst unproblematisch, weil im Zusammenhang mit SystemTicks nur relative Verzögerungs-Werte verwendet werden. Folglich kann ein Überlauf nur auftreten wenn der Wert des Delays zu hoch gewählt wird. Im Zusammenhang mit Protokollen auf dem Imote2 werden jedoch im Allgemeinen μs als Zeiteinheit verwendet. Dies führt zu einem Überlauf nach ca. 36 Minuten:

Definition Zeit: int mit 32 Bit und Vorzeichen
$2^{31} = 2147483648$

Voraussetzung: Zeitauflösung in μs
$\approx$ 2147 Sekunden
$\approx$ 36 Minuten

Wie diese Rechnung verdeutlicht, tritt bei einem 32-Bit Integer bei einer Auflösung in Mikrosekunden der Überlauf spätestens nach 36 Minuten auf. Wenn wir zusätzlich noch mit dem Faktor 1000 multiplizieren, der mit dem SET-Befehl erzeugt wird, erhalten wir eine maximale Wertebereich von zwei Sekunden. Dies ist bei weitem nicht ausreichend. Die Entwickler von RTDS haben sich nach Rücksprache bereit erklärt, in der RTDS-Implementierung eine Möglichkeit zu bieten, den verwendeten Datentyp für Zeit auf einen anderen numerischen Datentyp umzustellen. Dadurch würde sich die Laufzeit durch Verwendung eines 64-Bit Integers erhöhen lassen:

Definition Zeit: 64 Bit Integer ohne Vorzeichen
2^{64}

Voraussetzung: Zeitauflösung in μs
$\approx$ 580000 Jahre

Die benutzerdefinierte Konfiguration des Zeit-Datentyps ist für die nächste Version von PragmaDev RTDS (4.5) geplant, die im 1.Quartal 2014 erscheinen sollte, aber zum Zeitpunkt der Abgabe dieser Arbeit noch nicht veröffentlicht wurde.

4.4 Fehler in der Implementierung

Diese Liste erhebt keinen Anspruch auf Vollständigkeit. Es handelt sich hierbei um Fehler, die während der Entwicklung aufgetreten sind. Zusätzlich zur Beschreibung des Fehlers wird, wenn möglich, eine Lösung oder ein Workaround angegeben. Die folgenden Punkte wurden den Entwicklern mitgeteilt und könnten mit kommenden Versionen von RTDS behoben werden.

4.4.1 rtosless RTDS_Idle-Compilefehler

Das *rtosless*-Template enthält einen kleinen Fehler, der die Compilierung eines Programms verhindern kann. Das SDL-Environment (siehe Kapitel 3.3) wird in der Datei rtosless/RTDS_Env.c implementiert. Wenn Signale an das SDL-Environment gesendet werden, wird der RTDS-Environment-Prozess verwendet, um diese Signale zu verarbeiteten. Wenn der RTDS-Environment-Prozess eingebunden wird, kommt es jedoch zu diesem Fehler:

```
1 error: 'RTDS_Idle' was not declared in this scope
2      RTDS_setSdlState( RTDS_Idle );
```

Listing 4.1: Fehler bei Verwendung des *rtosless*-Template

Hierbei wird der falsche Name für den Idle-State verwendet. Das Ersetzen von RTDS_Idle durch RTDS_state_RTDS_Idle in RTDS_setSdlState(), behebt den Fehler.

4.4.2 String concat()-Fehler

SDL bietet die Möglichkeit, Zeichenketten zu konkatenieren. Dafür wird der Operator // angeboten. Dieser funktioniert in den meisten Fällen wie erwartet und wird durch den Codegenerator ohne Fehler übersetzt.

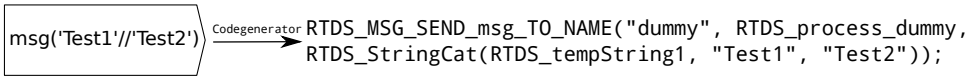

Abbildung 4.4: Sonderfall der Konkatenation führt zu Fehler

Ein Sonderfall ist, wenn in Signalparametern Zeichenketten konkateniert werden sollen (siehe Abbildung 4.4). Hierbei kommt es beim Auflösen des Makro RTDS_MSG_SEND_msg_TO_NAME zu diesem Fehler:

```
1 error: lvalue required as unary '&' operand
2    RTDS_MSG_SEND_msg_TO_NAME("dummy", RTDS_process_dummy,   RTDS_StringCat(
         RTDS_tempString1, "Test1", "Test2"));
```

Listing 4.2: Fehler bei der Konkatenation

Dieser Fehler konnte noch nicht dauerhaft behoben werden, da Änderungen am Codegenerator notwendig sind. Daher ist es notwendig, die Konkatenation der Zeichenketten vor dem Versenden der Signale in einer Variable zwischenzuspeichern (siehe Abbildung 4.5).

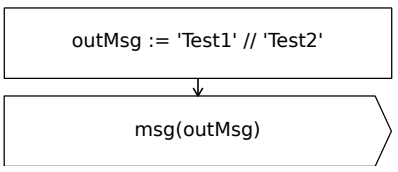

Abbildung 4.5: Workaround für Sonderfall der Abbildung 4.4

4.4.3 Fehlende time()-Methode für NOW

In SDL ist es gemäß SDL-Standard möglich, die aktuelle System-Zeit auszulesen. Dazu existiert in SDL das reservierte Schlüsselwort NOW. Diese wird vom Codegenerator zur Methode time() übersetzt (siehe Abbildung 4.6). Diese Methode ist jedoch standardmäßig nicht in der *rtosless*-Implementierung definiert. Daher kommt es zu diesem Fehler:

```
1 error: 'time' was not declared in this scope
2    tmp = time(NULL) + 1000;
```

Listing 4.3: Fehler bei der Konkatenation

Um diesen Fehler zu beheben, wurde in dem modifizierten *rtosless*-Template die Methode time() implementiert und der Wert der Variable RTDS_globalSystemTime zurückgegeben. Die Variable war bereits in rtosless/RTDS_OS.c definiert und speichert den aktuellen SystemTick.

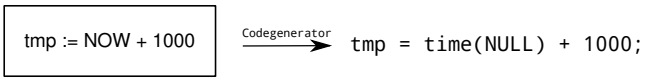

Abbildung 4.6: NOW-Anweisung mit entsprechendem C++-Code

4.5 Änderungen am RTDS-Template

Folgende Änderungen am RTDS-System sind für unsere Anwendungszwecke notwendig. Die durchgeführten Veränderungen an den Timern und dem RTDS-Scheduler sind mittels ifdef-Blöcken gekapselt und lassen sich durch Compileroptionen (siehe Anhang A.1.1 und A.2) deaktivieren, um das Originalverhalten wieder herzustellen.

4.5.1 Realisierung von absoluten Timer

Wie in Kapitel 4.3 erklärt, kann in der RTDS-Implementierung kein absoluter Timer genutzt werden. Es wurde gezeigt, dass `SET(NOW+Delay,Timername)` nicht eins-zu-eins übersetzt wird, sondern nur das Delay verwendet wird, um einen relativen Timer zu setzen. Um die Verwendung absoluter Timer zu ermöglichen, wurde die Implementierung von `RTDS_setTimer()`, der Repräsentation des SET-Befehls im C++-Code, für die Schnittstelle verändert. Nach der Änderung werden alle SET-Befehle als absolut behandelt und es ist nicht mehr möglich relative Timer zu verwenden. Die modifizierte *rtosless*-Version bietet die Compileroption '**-D RTDS_Relative_Timer**', um wieder auf den relativen Modus umzustellen.

Dadurch, dass der SET-Befehl nur den „Delay"-Teil auswertet und das Schlüsselwort `NOW` zu 0 auswertet, muss als 'Delay' die absolute Zeit übergeben werden (siehe Abbildung 4.7).

Das NOW in dem Taskblock kann durch einen beliebigen Referenzzeitpunkt ausgetauscht werden. Das NOW im SET-Befehl kann wegen der RTDS-Syntaxprüfung

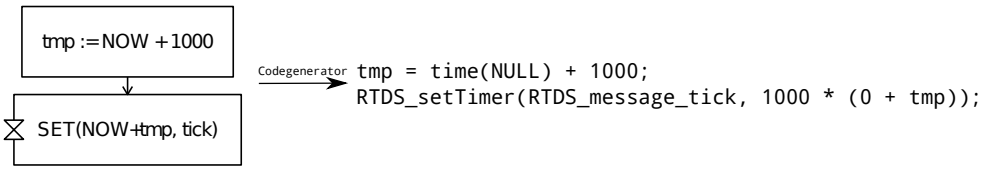

Abbildung 4.7: korrekte Verwendung von absoluter Zeit in Timern mit RTDS

nicht ausgelassen werden und wird immer als 0 interpretiert wird. Eine Implementierung der fehlenden `time()` Funktion für NOW (siehe Kapitel 4.4.3) wurde hinzugefügt. Wegen dem absoluten `RTDS_setTimer()` Befehl wurde die Timer-Verwaltung angepasst. Jetzt werden anstelle von Delays nur Zeitstempel zur Verwaltung verwendet. Wie bei dem Überlaufproblem mit Timern (siehe Kapitel 4.3) kann der Faktor 1000 Berechnungen verfälschen. Deshalb wird der Faktor vor der eigentlichen Verarbeitung aus dem absoluten Zeitwert wieder entfernt. Der Faktor schränkt den möglichen Wertebereich bis zum Überlauf im Zeitstempel des `RTDS_setTimer()` Befehls allerdings weiter ein.

4.5.2 Anpassungen am RTDS-Scheduler

Ziel des neuen Schedulers ist es, dass die Zeit selbst dann voranschreitet, wenn ununterbrochen Signale ausgetauscht werden. Ziel ist es, das Einfrieren der Zeit, wie im Beispiel in Kapitel 4.1 gezeigt, zu vermeiden. Deshalb wird im neuen Scheduler vor jeder Transitionsausführung die Zeit aktualisiert.

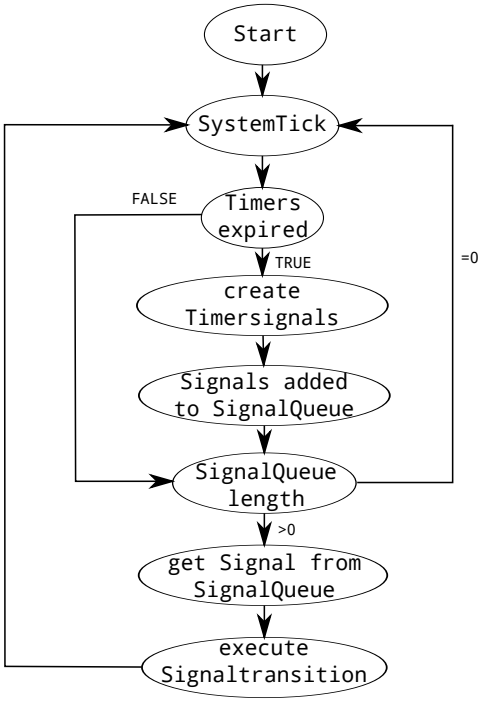

Abbildung 4.8: Ablauf der neuen Scheduler run()-Methode

Dazu wird in jedem Schleifendurchlauf der neuen run()-Methode (Abbildung 4.8) ein SystemTick erzeugt. Nachdem die Zeit aktualisiert wurde, wird geprüft, ob ein

Timer abgelaufen ist. Wenn ein Timer abgelaufen ist, wird ein Timer-Signal in die Signalqueue des Schedulers eingefügt. Damit haben wir das in Z.101 [Int12b, Kapitel 11.15 Seite 65] spezifizierte Verhalten erfüllt. Dieses besagt, dass ein Timer nicht vor seiner Ablaufzeit ausgelöst werden darf, erlaubt aber eine Verzögerung. Diese Verzögerung ist in der neuen run()-Methode abhängig von den Signalen, die sich bereits in der Queue befinden. Nachdem die Timer überprüft wurden, wird das nächste Signal aus der Queue verarbeitet. Sollten keine Signale mehr in der Queue vorhanden sein, wird die Schleife erneut gestartet. Dieser Ablauf ist hinsichtlich Overhead nicht optimal und wird daher für den Einsatz auf Imote2 in Kapitel 5.2 optimiert.

Kapitel 5

Anbindung vom RTDS-System an BiPS

In diesem Kapitel werden die durchgeführten Maßnahmen dargestellt, um die Anbindung des RTDS-SDL-Systems an BiPS zu ermöglichen. Dazu wird zuerst ein Überblick über die modifizierte Systemstruktur gegeben. Nachfolgend werden die einzelnen Veränderungen und Erweiterungen im Detail betrachtet. Zuletzt wird an zwei Beispielen die Verarbeitung von Signalen detailliert beschrieben.

5.1 Überblick über das erweiterte RTDS-System

In Kapitel 3.1 wurde ein Überblick über das RTDS-System gegeben. In diesem Kapitel wird ein kurzer Überblick über die durchgeführten Anpassungen gegeben. Neue oder modifizierte Systemteile sind in Abbildung 5.1 gestrichelt hervorgehoben. Um Funktionen (z.B. Verwendung des BiPS-UART-Treibers) in SDL anzu-

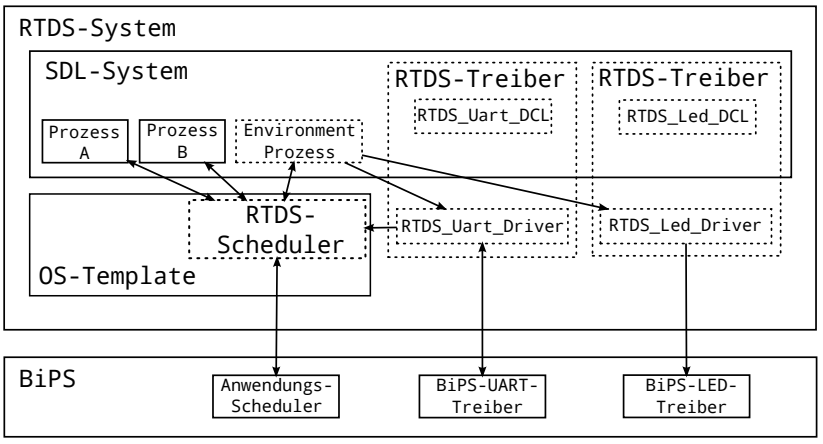

Abbildung 5.1: Erweiterte Struktur des RTDS-Systems

bieten und die Codegenerierung zu ermöglichen, werden RTDS-Treiber erstellt. Jeder RTDS-Treiber hat klar definierte Aufgaben, z.B. Steuerung eines Peripheriegeräts. Der RTDS-Treiber setzt sich dabei aus zwei Teilen zusammen, einer

SDL-Declaration-Datei (siehe z.B. RTDS_Uart_DCL in Abbildung 5.1) und einer C++-Treiber-Implementierung (siehe z.B. RTDS_Uart_Driver in Abbildung 5.1). Im SDL-System werden in SDL-Declaration-Dateien Signale und Synonyme für den RTDS-Treiber definiert, die innerhalb des Systems die Verwendung notwendiger Signale erlauben. Um ein Signal im SDL-System zu versenden, muss eindeutig ein Empfänger angegeben werden. Deshalb wird das Signal an den SDL-Environment-Prozess gesendet. Das Environment leitet das Signal an den richtigen RTDS_X_Driver weiter. Diese verarbeiten die Signale und verwenden, wenn notwendig, die BiPS-Treiber (siehe Kapitel 5.7).

Eingehende Signale müssen ebenfalls eindeutig an einen Prozess innerhalb des SDL-Systems adressiert werden. Um diese Adressierung umzusetzen gibt es zwei Lösungsansätze, Verwendung von festen Prozessnamen oder dynamische Anmeldung eines Prozesses als Empfänger. Die Verwendung von festen Prozessnamen ist unpraktisch, weil ein Prozess dadurch nicht Empfänger für Signale mehrerer Treiber sein könnte und nachträgliche Änderungen an einem System umständlicher werden. Durch die dynamische Anmeldung des Prozesses mit einem Signal ist es möglich, dass ein Prozess Empfänger für Signale mehrerer Treiber wird. Ein weiterer Vorteil ist, dass der Empfänger während der Ausführung des Systems verändert werden kann. Daher wurde die Anmeldung als Empfänger ausgewählt und mit einem RTDS_X_Init Signal umgesetzt. Dieses Initialisierungssignal muss aufgerufen werden, damit ein Prozess die eingehenden Signale eines Treibers erhält.

Zusätzlich wurde die Implementierung des RTDS-Schedulers optimiert. SystemTicks werden durch die Verwendung der Hardwaretimer ersetzt. Der RTDS-Scheduler wird mit Verwendung des BiPS-Anwendungsschedulers optimiert, in dem sich der RTDS-Scheduler schlafen legt und erst durch äußere Events (z.B. eingehende Signale, siehe Kapitel 5.8).

5.2 Optimierung des Schedulers mit BiPS

Der RTDS-Scheduler verwendet in der Standard-Implementierung SystemTicks, um das SDL Zeitmodell zu realisieren. Dies ist für kleine Tests ausreichend, eignet sich aber nicht für eine ernsthafte Verwendung, da viel Energie und Rechenzeit verschwendet wird. BiPS bieten die Möglichkeit Imote2-Hardwaretimer zu verwenden. Daher wurde der RTDS-Scheduler modifiziert, so dass statt der SystemTicks der Wert des Hardwaretimers verwendet wird. Die bereits in Kapitel 4.4.3 verwendete Variable RTDS_globalSystemTime speichert ab sofort den Wert des Hardwaretimers. Die in Kapitel 4.3 beschriebene Einschränkung des Wertebereichs, macht eine Skalierung der Schrittweite für Timer notwendig, weil ein maximaler Delay bzw. Zeitstempel von zwei Sekunden nicht ausreichend ist. Standardmäßig ist diese Schrittweite auf 1000 μs/(SDL-)Zeiteinheit eingestellt.

Timer-Befehl: `SET(NOW+2000,tick)`
Schrittweite: 1000 μs/Zeiteinheit
Berechnung: Timerwert $\times$ Schrittweite = tatsächliche Zeit
2000 Zeiteinheiten $\times$ 1000 μs/Zeiteinheit = 2 s

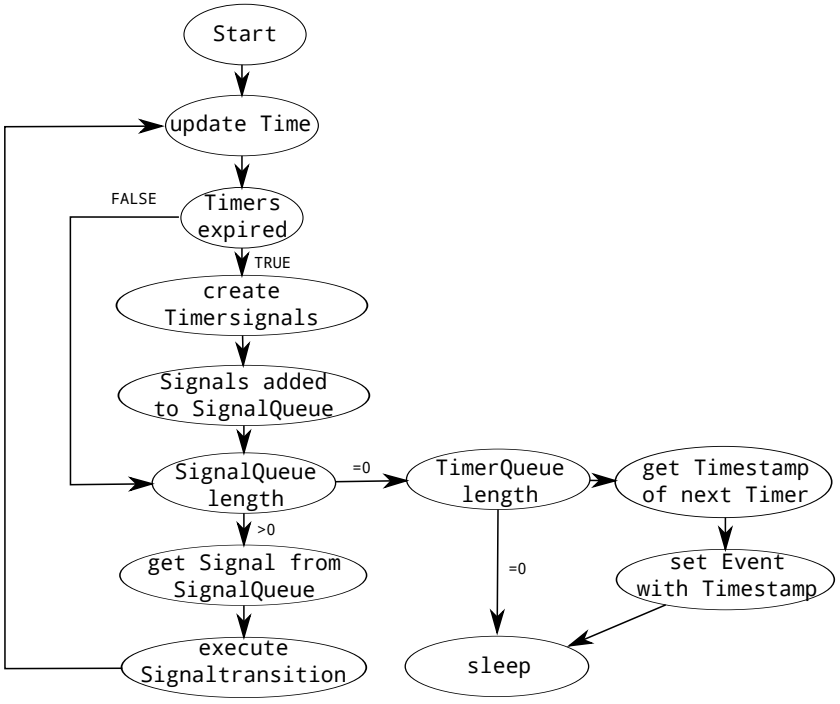

Abbildung 5.2: Ablauf der modifizierten RTDS-Scheduler run()-Methode mit Events

Mit dem Timer- und Eventsystem des BiPS ist es möglich, dass sich eine Anwendung schlafen legt (siehe Abbildung 5.2) und erst zum nächsten Event geweckt wird. Dabei müssen zwei verschiedene Arten von Events beachtet werden: Timer- und Signalevents. Die Timerevents werden vom RTDS-System selbst gesetzt, wenn zurzeit kein weiteres Signal bearbeitet werden kann. Damit kann das RTDS-System bis zum nächsten Event (z.B. dem gesetzten Timer Event) pausiert werden. Die Signalevents werden durch Interrupts vom BiPS, z.B. nach Empfang eines Rahmens, ausgelöst und treten unerwartet auf. Sie sollten möglichst schnell behandelt werden. Deshalb wird der RTDS-Scheduler auch im Falle eines solchen Ereignisses geweckt. In beiden Fällen kann sich die Abarbeitung von Events aufgrund zeitkritischer Protokoll-Funktionen auf dem Imote2 verzögern, weil diese priorisiert im Interruptkontext abgearbeitet werden.

5.3 Anpassung des SDL-Environment-Prozesses

Das hier realisierte SDL-Environment hat zwei Aufgaben innerhalb des RTDS-Systems. Zum einen dient es zur Weiterleitung von Signalen an die zuständigen RTDS-Treiber und ist zweitens für die Initialisierung der RTDS-Treiber verantwortlich. Die Anmeldung des RTDS-Treibers beim Environment (siehe Kapitel 5.3.1) ist notwendig, damit der zuständige RTDS-Treiber für ausgehende RTDS-Signale ermittelt werden kann. Die Initialisierung der RTDS-Treiber wird in Kapitel 5.3.2 vorgestellt.

5.3.1 Implementierung der Anmeldung von RTDS-Treibern

Aufgabe des SDL-Environment-Prozesses ist das Weiterleiten der Signale an die RTDS-Treiber. Der nicht modifizierte SDL-Environment-Prozess würde dieses Signal einfach verwerfen. Die Realisierung nutzt eine Abwandlung des Observer-Pattern [RJG94] (siehe Abbildung 5.3).

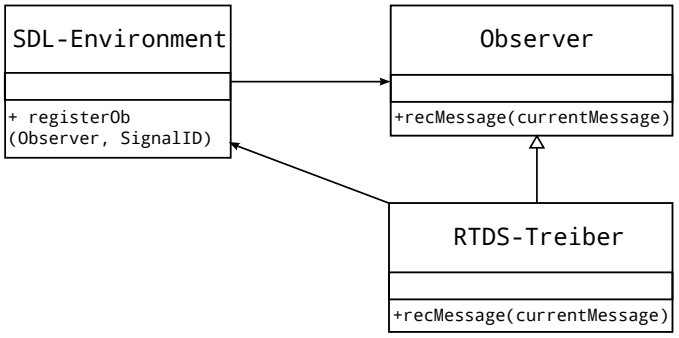

Abbildung 5.3: Überblick über die Environment-Treiber-Kommunikation

Nach dem Observer-Pattern gibt es ein betrachtetes Subjekt, dass die Daten enthält, die von den Observern beobachtet werden können. Diese Observer registrieren sich bei dem Subjekt und werden dann informiert, wenn sich die relevanten Daten ändern. In der vorgestellten Variante wird der SDL-Environment-Prozess zum beobachteten Subjekt und bietet die Möglichkeit, dass sich RTDS-Treiber als Observer registrieren. Bei der Registrierung wird die Signal-ID angegeben, die beobachtet werden soll. Wird ein Signal vom SDL-Environment empfangen, wird überprüft ob ein Observer diese Signal-ID angemeldet hat. Ist das der Fall wird das Signal an diesen weitergeleitet, ansonsten wird das Signal verworfen.

5.3.2 Initialisierung der RTDS-Treiber

Die Initialisierung der RTDS-Treiber wird als Teil der Initialisierung des SDL-Environments durchgeführt (siehe Message Sequence Chart (MSC) in Abbildung 5.4). Dafür wird im Konstruktor des RTDS-Environment-Prozesses überprüft, ob ein Aktivierungssignal für den Treiber in der RTDS_gen.h Datei definiert wurde. Für dieses Aktivierungssignal wird ein spezielles Signal (RTDS_EnvObserver_X_Enable) verwendet. Dieses Signal hat keine weiteren Effekte und wird im SDL-System nicht weiter verwendet. Theoretisch wäre auch die Verwendung eines beliebigen Signals des RTDS-Treibers möglich.

Anhand definierter Aktivierungssignale können wir eindeutig ableiten, welche Treiber für die Ausführung des RTDS-Systems notwendig sind. Diese Treiber werden initialisiert und erhalten dabei eine Referenz auf das SDL-Environment. Mit dieser Referenz kann sich der Treiber als Observer für die für ihn relevanten Signale registrieren. Die Anzahl der aktivierten RTDS-Treiber hängt vom jeweiligen RTDS-System ab. Ein verwendeter RTDS-Treiber registriert jedoch immer alle ausgehenden Signale die er verarbeiten kann (siehe Abbildung 5.4).

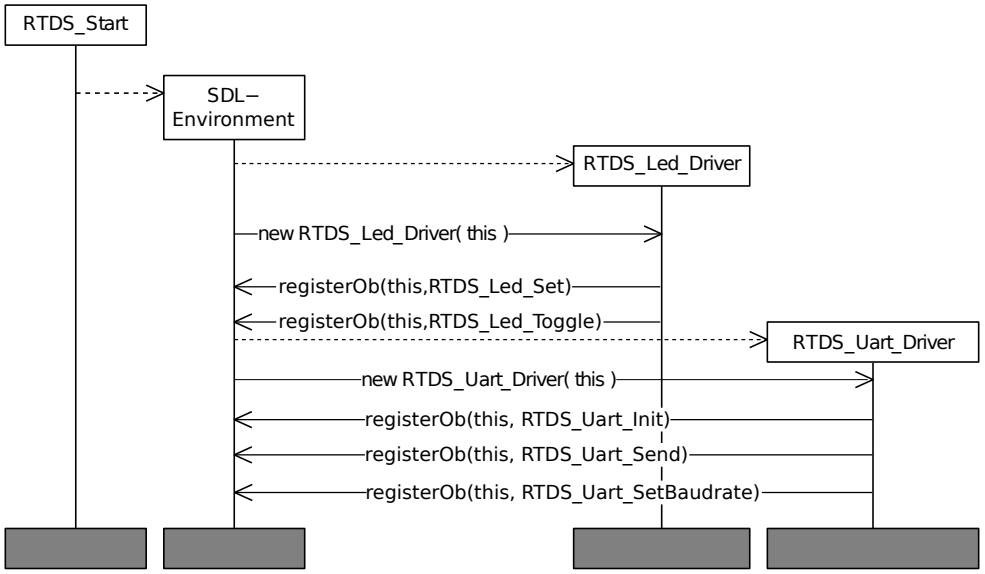

Abbildung 5.4: Initialisierung und Registrierung der RTDS-Treiber am Beispiel von RTDS-LED- und RTDS-UART-Treiber

5.4 Deklaration der Treiber-Signale in SDL

Für jeden RTDS-Treiber sind Signale und Datentypdefinitionen in SDL notwendig, um die Funktionalitäten zu nutzen. Diese SDL-Declaration-Dateien sind alle nach der selben Konvention aufgebaut und heißen RTDS_*Name*_DCL, wobei *Name* die Funktionalität (LogIF für Logging Interface) oder Peripherie (z.B. UART) angibt. Sie sind wie folgt aufgebaut (Notation: **fest**, *treiberspezifisch*, frei wählbar):

- Datentypen: X
- Operatoren: X
- Synonyme: **RTDS_***Name*_X
- Treiber-Aktivierungssignal: **RTDS_EnvObserver_***Name*_**Enable**
- Initialisierung eingehender Signale: **RTDS_***Name*_**Init**
- weitere Signale: **RTDS_***Name*_X
- Signallist aller ausgehender Signale: **SDL2***Name*
- Signallist aller eingehender Signale: *Name***2SDL**

Operatoren und Datentypen unterliegen keinem festen Namensschema. Im Allgemeinen beginnen Synonyme und Signale mit dem RTDS_Name_-Präfix. Die einzige Ausnahme stellt das Treiber-Aktivierungssignal (siehe Kapitel 5.3.2) dar, dies ist mit dem speziellen Präfix RTDS_EnvObserver_Name_ markiert. Dies soll die spezielle Aufgabe dieses Signals darstellen. Außer dem Aktivierungssignal sind alle Teile der Definition optional und können in der SDL-Declaration-Datei deshalb ausgelassen werden. Die vollständigen SDL-Declaration-Dateien für LED, UART und LogIF sind im Anhang A.3 zu finden.

5.5 Aufgabe der Treiber-Implementierung

Die Aufgabe der Treiber-Implementierung, nachfolgend als RTDS_X_Driver bezeichnet, ist das Empfangen und Verarbeiten aller SDL-Signale des RTDS-Treibers. Dazu werden bei der Initialisierung des Treibers alle Signal-IDs des RTDS-Treibers beim SDL-Environment-Prozess angemeldet (siehe Kapitel 5.3.2). Die bei der Codegenerierung dynamisch vergebenen Signal-IDs, werden beim Compilieren aus der RTDS_gen.h Datei ausgelesen. Betrachten wir dazu einen Ausschnitt aus der RTDS_gen.h Datei eines größeren Systems:

```
1 /* DEFINES FOR SIGNALS AND TIMERS */
2 #define RTDS_message_RTDS_EnvObserver_Led 1
3 #define RTDS_message_RTDS_Led_Set 2
4 #define RTDS_message_RTDS_LogIF_SetLogLvl 3
5 #define RTDS_message_RTDS_Uart_Init 4
```

Listing 5.1: Ausschnitt der RTDS_gen.h

In diesem Ausschnitt werden vier Signal-IDs definiert. RTDS_message_RTDS_EnvObserver_Led wurde definiert, weil in der RTDS_Led_DCL-Datei das Aktivierungssignal nach Kapitel 5.4 enthalten ist. Betrachtet man z.B. den RTDS_Led_Driver, muss aus diesem Ausschnitt die ID 2 beim SDL-Environment angemeldet werden, weil es sich dabei um das RTDS_Led_Set Signal handelt. Die IDs 3 und 4 sind vom LogIF- und UART-Treiber und daher für den RTDS_Led_Driver bedeutungslos.

Wenn der SDL-Environment-Prozess ein Signal an den RTDS_X_Driver übergibt, wird in Abhängigkeit von der Signal-ID das Signal verarbeitet. In der Regel nutzt der RTDS_X_Driver hierfür die BiPS-Treiber, um z.B. konkrete Hardware anzusprechen. Sollte besondere Funktionalität notwendig sein, kann diese direkt im RTDS_X_Driver implementiert werden, oder in eine separate Datei ausgelagert werden.

Der RTDS_X_Driver ist auch für die eingehenden Signale verantwortlich die an einen Prozess im System gesendet werden sollen. Der Umweg über das SDL-Environment ist nicht notwendig, da durch die Verwendung des Initialisierungssignals (siehe Kapitel 5.1) direkt der richtige Prozess adressiert werden kann.

5.6 Interrupt Handling

Mit Ausnahme der eingehenden Signale, die in Interrupts auf dem Imote2 erstellt und zwischengespeichert werden, arbeitet ein RTDS-System im Nicht-Interruptkontext. Dadurch, dass BiPS auch zeitkritische Protokolle beinhaltet, ist es notwendig, dass die Systemintegrität geschützt wird und dabei die Funktionen des gesamten Stacks möglichst wenig gestört werden. Die Systemintegrität ist gefährdet, wenn Fehler in der Synchronisation vom gemeinsamen Speicher auftreten. Die kritische Stelle in der SDL-Integration stellt dabei die Signalqueue dar. Diese wird dauerhaft vom RTDS-Scheduler im Nicht-Interruptkontext verarbeitet und ist gleichzeitig die Stelle, an der die RTDS-Treiber Signale im Interruptkontext ins System übergeben. Um Race-Conditions auf der Signalqueue durch Interrupts zu vermeiden, werden deshalb

an den Zugriffen auf die Signalqueue Interrupts zeitweise gesperrt. Zurzeit werden dabei noch alle Interrupts deaktiviert. Hierbei besteht noch die Möglichkeit einzelne Interrupts zu sperren und damit die auftretenden Verzögerungen unabhängiger Interruptverarbeitungen zu reduzieren.

Folgende kritische Stellen im RTDS_Scheduler erfordern es die Interrupts zu sperren:

- Einfügen eines Signals in die Signalqueue
- Entnehmen eines Signals aus der Signalqueue
- Kopieren der Savequeue in die Signalqueue

An diesen Stellen wird jeweils überprüft, was sich in der Signalqueue befindet, und diese verändert. Durch Sperren der Interrupts an diesen Stellen ist die Integrität der Signalqueue gesichert.

5.7 Beispiel eines ausgehenden Signals

Wie in Kapitel 5.4 geschildert wurde, werden ein- und ausgehende Signale eines RTDS-Treibers in einer SDL-Declaration-Datei definiert. Im Beispiel in Abbildung 5.5 wird innerhalb eines SDL-Systems das Signal RTDS_Led_Toggle gesendet. Dieses Signal muss an den SDL-Environment Prozess gesendet werden. Der SDL-Environment Prozess sucht in seiner Liste von Observern den zuständigen RTDS-Treiber und gibt das Signal unverändert weiter. Erst im RTDS_Led_Driver, der sich für dieses Signal registriert hat, wird das Signal zerlegt und ausgewertet. In diesem Beispiel ist keine besondere Verarbeitung notwendig, es wird nur der LED_toggle Befehl in BiPS aufgerufen.

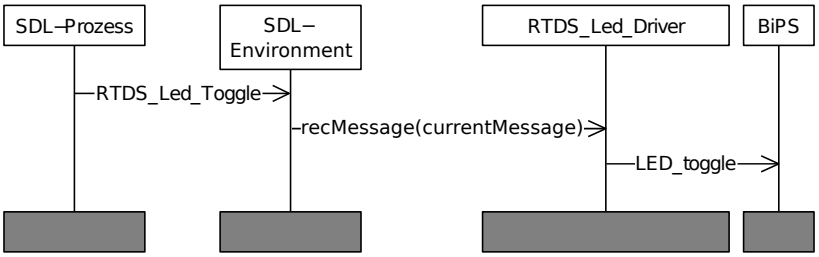

Abbildung 5.5: MSC eines ausgehenden RTDS_Led_Toggle-Signals

5.8 Beispiel eines eingehenden Signals

In diesem Beispiel wird angenommen, dass es zurzeit keine anderen Signale in der Signalqueue gibt, und der RTDS-Scheduler deshalb pausiert. Eingehende Signale von der Umgebung beginnen mit einem Interrupt auf dem Imote2, der von

der Hardware ausgelöst wird. BiPS ruft im Beispiel in Abbildung 5.6 den registrierten Callback des RTDS_Uart_Drivers auf. Diese Registrierung wurde durch RTDS_Uart_Init-Signal vorher durchgeführt und ist nicht in Abbildung 5.6 gezeigt. Der RTDS_Uart_Driver konstruiert hierauf ein Signal aus den übergebenen Daten. Dieses wird mit sendMessage in die Signalqueue des RTDS-Schedulers eingefügt. Dann ruft der Treiber die eventWakeup-Funktion des RTDS-Schedulers auf, damit dieser EVENT_emit in BiPS aufruft. EVENT_emit signalisiert, dass der BiPS-Anwendungsscheduler den RTDS-Scheduler wecken soll, um den RTDS-Scheduler im User-Kontext auszuführen. Hiermit endet die Interruptverarbeitung in der Schnittstelle. Der BiPS-Scheduler hat den EVENT_emit empfangen und setzt zum nächsten möglichen Zeitpunkt die Ausführung des RTDS-Schedulers im Nicht-Interruptkontext fort. Nach einer kurzen Verarbeitungszeit wird dann das RTDS_Uart_Receive an den korrekten Prozess zugestellt.

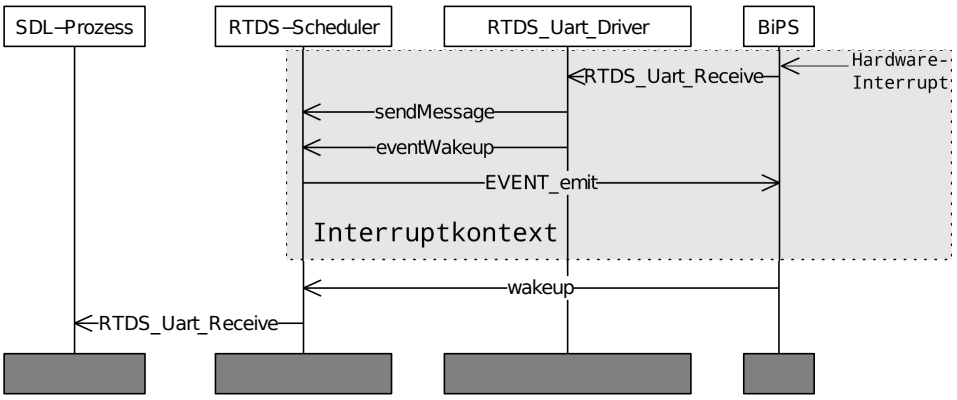

Abbildung 5.6: MSC eines eingehenden UART-Signals

Sollte der RTDS-Scheduler nicht pausiert sein, erfolgt der gleiche Ablauf. In dem Fall wird EVENT_emit verworfen und die reguläre Ausführung des RTDS-Schedulers nach dem Interrupt und damit die Abarbeitung der nun veränderten Signalqueue fortgesetzt. Durch die in Kapitel 5.6 vorgestellte Sperrung der Interrupts treten dabei keine Konflikte auf.

Kapitel 6

Ausführbares RTDS-System erzeugen

Dieses Kapitel erklärt, wie aus einem SDL-System ein ausführbares RTDS-System generiert werden kann und wie eine auf BiPS aufbauende Anwendung erzeugt wird. Dazu betrachten wir die Code Generation Options von RTDS und die für die RTDS-BiPS-Schnittstelle relevanten Makefiles.

6.1 Codegenerierung mit RTDS

Bevor es möglich ist ein *rtosless*-Anwendung mit RTDS zu erzeugen, ist es notwendig mit RTDS ein UML Deployment Diagramm für das System zu erstellen. Dazu ist es für unsere Zwecke ausreichend im Deployment Diagramm einen Systemblock zu erstellen und diesen mit dem Tag *scheduled* zu versehen (siehe Abbildung 6.1). Das Deployment Diagramm gibt an, dass der RTDS-Codegenerator den RTDS-Scheduler für das Scheduling verwenden soll. Für eine ausführliche Erklärung sei auf das RTDS User Manual [Prac, Kapitel 6.3.5.1 Seite 184] verwiesen.

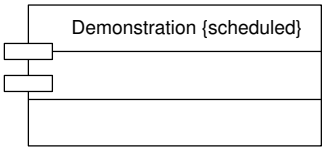

Abbildung 6.1: UML Deployment Diagramm

Danach wird unter „Generate“ > „Options...“ das Muster (siehe Abbildung 6.2 und 6.3) für die Code Generation Options verwendet, um den Code zu generieren. Im Beispiel wird die Ausführung für das Betriebssystem Linux gezeigt. Dafür ist das Einbinden der externen Makefile.linux (siehe Kapitel 6.2) notwendig. Die „Include external makefile“- und die „Do build“-Option können für eine BiPS-Applikation ausgelassen werden. Im Muster nicht verwendete Code Generation Options werden nicht benötigt.

Dieses Muster ist auf die Ordnerstruktur der Bachelorarbeit angepasst, eine Korrektur der Pfade für „Code templates dir:“ und „external makefile“ sind bei Veränderung der Struktur notwendig. Die grün markierten Compileroptionen (siehe Abbildung

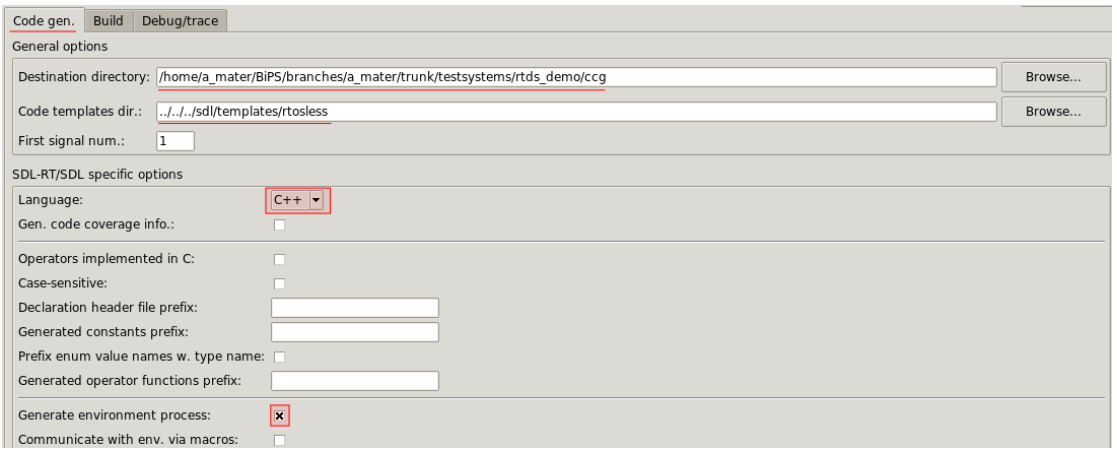

Abbildung 6.2: „Code gen.“-Reiter der Optionen

6.3) sind abhängig von dem gewünschten Verhalten zu wählen und betreffen z.B. wie in der Abbildung gezeigt die Konfiguration des Log-Interfaces. Für Dokumentation der verschiedenen Compileroptionen siehe Anhang A.

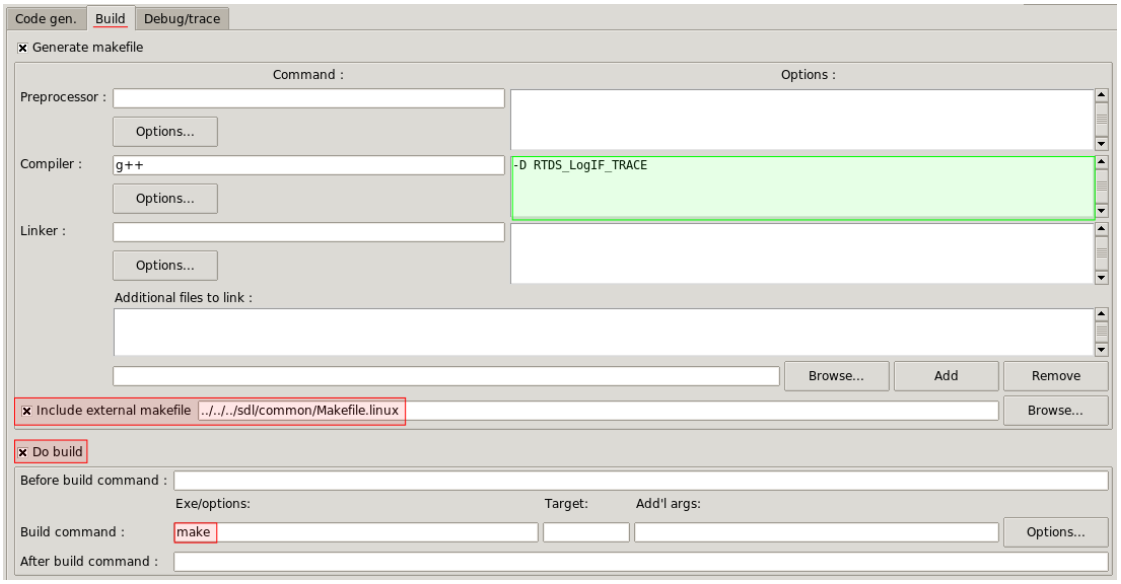

Abbildung 6.3: „Build“-Reiter der Optionen

Folgende Optionen des Musters sind dabei besonders wichtig:

- **Generate environment process: checked** Wie in Kapitel 5.4 erklärt, leitet der SDL-Environment-Prozess die Signale an die RTDS-Treiber weiter. Daher ist dieser Prozess für unsere Systeme notwendig und darf nicht weggelassen werden.
- **include external makefile: ../../../sdl/common/Makefile.linux** Der Codegenerator führt im Laufe der Ausführung die make-Anweisung aus, um Code zu generieren. Dabei wird die eingebundene externe Makefile bereits ausgeführt und kann Fehler erzeugen. Die Makefile.linux fängt diesen Fall ab, in dem der Inhalt der Makefile nur verarbeitet wird, wenn

der RTDS_TARGET_BASE_NAME nicht RTDS_includes4MsgStruct entspricht.

- **Build command: make** Dieser Befehl wird während der Codegenerierung das erste Mal ausgeführt. Veränderungen am Build command können deshalb die Codegenerierung verhindern.

6.2 Makefiles für das RTDS-System

Insgesamt werden für die RTDS-Systeme bis zu 4 Makefiles verwendet. Abbildung 6.4 zeigt den Aufbau von **Makefile** (Farben-Legende: blau = auszuführende Makefile für Architektur, grün = eingebundene Makefile). Diese wird verwendet, um eine Linux-Anwendung zu erstellen. **Makefile.linux** ist in der **Makefile** eingebunden. Die Pfeile in der Abbildung zeigen dabei, in welchem Bereich die einzelnen Bestandteile der Anwendung compiliert werden werden.

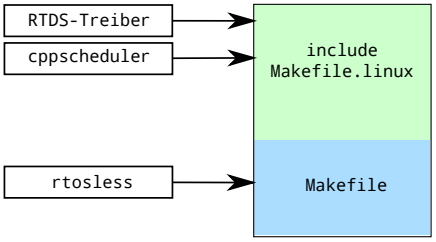

Abbildung 6.4: Aufbau der **Makefile** für Linux-Anwendungen

Abbildung 6.5 zeigt den Aufbau der **Makefile.bips**, diese wird verwendet um Imote2-Anwendungen zu erstellen. Die **Makefile.bips** bindet zusätzlich noch die **Makefile.imote** ein, um die Bestandteile von BiPS zu verwenden.

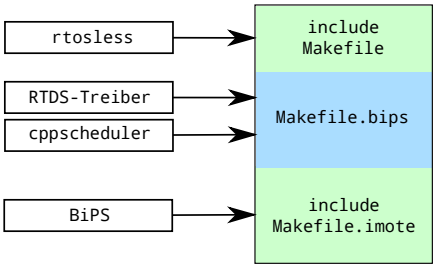

Abbildung 6.5: Aufbau der **Makefile.bips** für Imote2-Anwendungen

Der Inhalt und die Aufgabe der einzelnen Makefiles sind wie folgt:

- **Makefile** wird automatisch durch den RTDS-Codegenerator erzeugt. Das Makefile wird verwendet, um die Code Generation Options, Pfadangaben und Compileroptionen auszulesen. Wenn der Codegenerator mit den Optionen nach Kapitel 6.1 erzeugt wurde, kann das Makefile ausgeführt werden, um eine Unix-Executable zu erzeugen.

- **Makefile.bips** ist notwendiger Ausgangspunkt, um ein Imote2-Executable zu erzeugen. Das Makefile.bips bereitet durch den Codegenerator erzeugte Dateien für Makefile.imote vor, in dem Pfade und Compileroptionen auf die von Makefile.imote erwartete Form gebracht werden. Um das Imote2-Executable zu erstellen, wird die Makefile.bips nach der Codegenerierung ausgeführt.

- **Makefile.imote** ist ein Bestandteil von BiPS. Das Makefile bekommt Source- und Header-Dateien übergeben und ist für die Compilierung des Imote2-Executables zuständig. Die korrekte Verwendung von Makefile.imote wird mit Makefile.bips realisiert.

- **Makefile.linux** ist notwendig, um eine fehlerfreie Compilierung des RTDS-Systems für Unix zu erreichen. Mit dem Codegenerierungs-Optionen (siehe Kapitel 6.1) wird das Makefile.linux in das Makefile des RTDS-Codegenerator eingebunden. Das Makefile.linux ergänzt die Dateien der RTDS-Treiber und verwendet die durch Kapitel 5 modifizierten Dateien (z.B. RTDS_Scheduler.cpp).

Kapitel 7

Test und Evaluation der RTDS-Anbindung an BiPS

In diesem Kapitel wird anhand eines Demo-System gezeigt, welche Möglichkeiten die RTDS-Anbindung an BiPS zu diesem Zeitpunkt bietet. Dazu wird ein System betrachtet, dass die drei implementierten RTDS-Treiber (UART, LED und LogIF) verwendet und den für die Imote2-Plattform entworfenen Scheduler testet. Nachfolgend wird zuerst das Demo-System detailliert beschrieben und die Anweisungen ausführlich erklärt. Dabei wird die Ausführung auf dem Imote2 betrachtet und die Funktionalität des UART und der LEDs getestet. Anschließend wird die RTDS-Anbindung durch Log-Ausgaben von zwei Testläufen evaluiert.

7.1 Aufbau des Demo-Systems

In Abbildung 7.1 ist die hierarchische Baumstruktur der Systemkomponenten dargestellt. Das Projekt mit dem Namen Demonstration besteht aus dem Systemblock *Demonstration*. Dieser besteht aus drei Blöcken *Clock*, *DriverDemo* und *Noise*. Die-

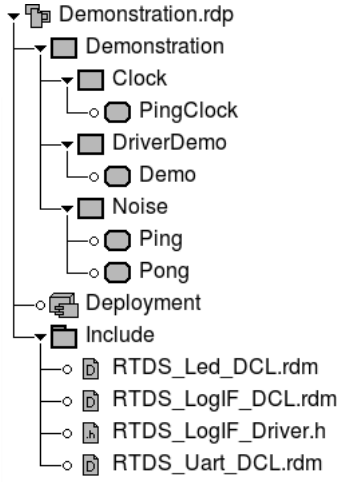

Abbildung 7.1: Projektstruktur der Demonstration

se drei Blöcke beinhalteten jeweils verschiedene Prozesse. Zusätzlich enthält der Projektbaum noch das UML Deployment Diagramm (vgl. Abbildung 6.1). Dieses ist notwendig, um ein RTDS-System auf Basis des *rtosless*-Template zu generieren. Da der Diagrammeditor von RTDS die Information über die Definitionen der RTDS-Treiber (z.B. Signale und Datentypen) benötigt, werden diese im Package *Include* eingebunden. Dieses Package enthält die Declaration-Dateien gemäß Kapitel 5.4 für alle drei RTDS-Treiber-Implementierungen (Endung: .rdm). Zusätzlich muss der RTDS_LogIF_Driver.h Header eingebunden werden, um alle überladenen Varianten des Operators `toStr` (siehe Anhang A.3.3) zu unterstützen. Das Problem besteht wenn der `toStr`-Operator in Verbindung mit NOW verwendet wird, weil der Rückgabe-Wert vom Typ `uint64_t`. Dieser wird wegen des größeren Wertebereichs verwendet, wird aber noch nicht automatisch als Zeittyp von SDL eingesetzt.

7.2 Systemblock Demonstration

Der Systemblock (Abbildung 7.2) beinhaltet drei Blöcke und stellt auf Systemebene alle benötigten Signale und Deklarationen bereit. Dazu wird das Package *Include*

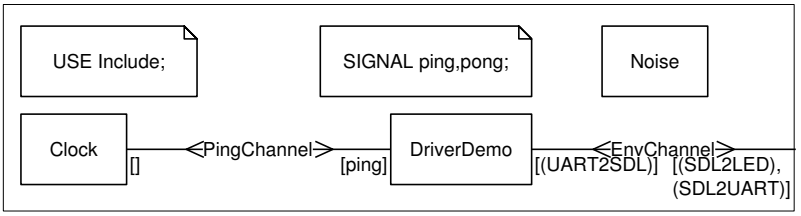

Abbildung 7.2: Systemblock Demonstration

mit der USE-Anweisung eingebunden. Dadurch stehen in allen Diagrammen des Systems die Signale der RTDS-Treiber zur Verfügung. Zusätzlich wird noch das Signal *ping* definiert. Alle ausgehenden SDL-Signale an die Treiber werden über den *EnvChannel* an das Environment und dadurch an die RTDS-Treiber geschickt, um die Verarbeitung nach Kapitel 5.7 durchführen zu können. Dabei werden aus Gründen der Übersichtlichkeit die Signal-Listen der Treiber verwendet. Die Rückrichtung der UART-Signale vom Environment zum Prozess *DriverDemo* ist nur im SDL-Diagrammeditor notwendig, weil die Kanäle während der Codegenerierung aufgelöst (siehe Kapitel 3.3 und Kapitel 5.5) und eingehende Signale direkt an den Prozess gesendet werden (vgl. Abbildung 5.6). Die Rückrichtung kann im Diagramm nicht weggelassen werden, weil sonst die Syntaxprüfung von RTDS den Empfang der Signale im Prozess nicht erlaubt. Über den zweiten Kanal sendet der Block *Clock* Signale vom Typ *ping* an den Block *DriverDemo*. Der Block *Noise* besitzt keine Känäle und kann deshalb keine Signale an einen anderen Block senden.

7.3 Block Clock

Der Block *Clock* (Abbildung 7.3) besitzt einen Prozess *PingClock*, der über den Kanal *ticker* nach außen verbunden ist. Diese Verbindung wird im Systemblock von

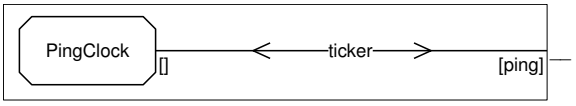

Abbildung 7.3: Block Clock

PingChannel zum Block *DriverDemo* fortgeführt. Der Prozess *PingClock* (Abbildung 7.4) setzt bei seiner Initialisierung einen Timer *tick*, der zum Zeitpunkt 1000 ausgelöst wird. Die tatsächliche Zeit hängt vom eingestellten Wert für die Schrittweite ab (siehe Einführung von absoluten Timern in Kapitel 4.5.1 und Kapitel 5.2).

Schrittweite: 1000 μs/Zeiteinheit
Berechnung: 1000 Zeiteinheiten $\times$ 1000 μs/Zeiteinheit = 1 s

Schrittweite: 1 μs/Zeiteinheit
Berechnung: 1000 Zeiteinheiten $\times$ 1 μs/Zeiteinheit = 1 ms

Dabei wird der Timer nur implizit durch den SET-Befehl definiert. Dies ist in RTDS möglich, entspricht aber nicht dem SDL-Standard. Nach dem Erstellen des Timers

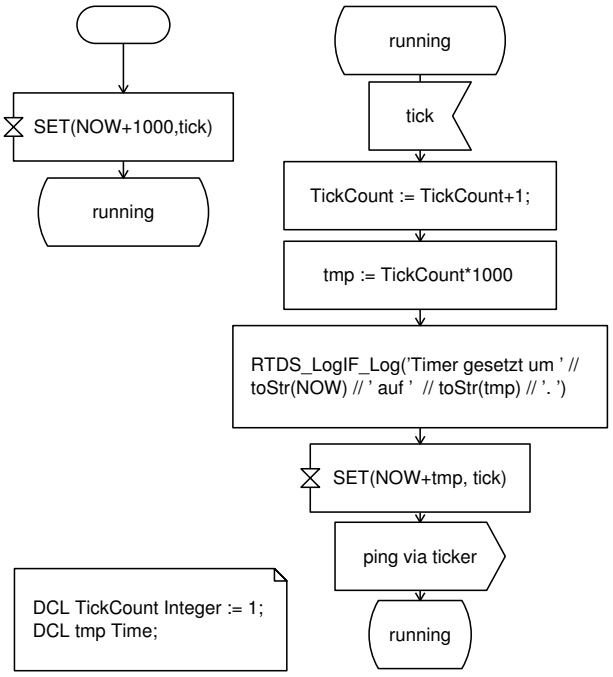

Abbildung 7.4: Prozess PingClock

wird in den Zustand *running* gewechselt. Der Zustand *running* wartet auf das Signal vom Timer *tick*. Dann wird die absolute Zeit mit einem *TickCount*-Zähler berechnet und diese in der Variable *tmp* gespeichert. Der berechnete Zeitstempel und der aktuelle Zeitstempel werden mit dem Befehl `RTDS_LogIF_Log` ausgegeben und der Timer *tick* wird erneut gesetzt. Die Verwendung von `NOW+tmp` in `SET(NOW+tmp,tick)` ist gleichbedeutend, mit dem Zeitpunkt *tmp*. Hier ist die Unterscheidung der zwei

Auswertungen von `NOW` extrem wichtig. Das `NOW` im SET-Befehl wird vom Codegenerator stets als 0 interpretiert, während das `NOW` außerhalb zu einem Aufruf der `time`-Methode wird, die die aktuelle Systemzeit zurückgibt.

Der absolute Timer-Modus ist Standard in dem modifizierten *rtosless*-Template und kann mit einer Compileroption (siehe Kapitel 4.5.1) auf den relativen Modus umgestellt werden. Nachdem der Timer wieder gesetzt wurde, wird zuletzt das Signal *ping* verschickt. Danach verbleibt der Prozess im Zustand *running*. Dadurch besteht der gesamte Ablauf des Prozesses in der periodischen Generierung des Signals *ping*.

7.4 Block DriverDemo

Der Block *DriverDemo* (Abbildung 7.5) enthält nur eine Fortsetzung der Kanäle im Systemblock zu einem Prozess.

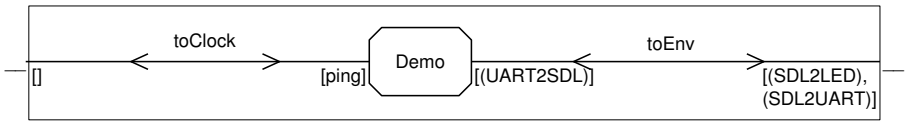

Abbildung 7.5: Block DriverDemo

Eine Aufteilung der Funktionalität des *Demo* Prozesses war nicht notwendig. Der Prozess wechselt nach der Initialisierung (Abbildung 7.6) in seinen einzigen Zustand *idle*. Dieser erwartet die Signale *RTDS_ Uart_ Receive* (Abbildung 7.7) und *ping* (Abbildung 7.8). Die Initialisierung besteht darin, den Imote2 auf den gewünsch-

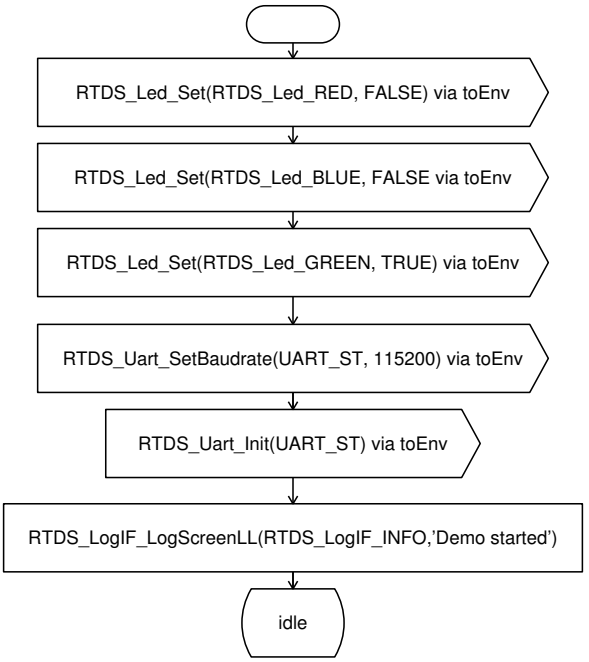

Abbildung 7.6: Initialisierung im Prozess Demo

ten Ausgangszustand zu bringen. Dazu werden zuerst alle LEDs außer der grünen ausgeschaltet. Danach werden die UART-Einstellungen auf dem Standard-Port

UART_ST vorgenommen. Die Baudrate wird auf den gewünschten Wert 115200 bit/s gesetzt. Danach wird mit dem *RTDS_ Uart_ Init*-Signal der Prozess *Demo* als Empfänger beim RTDS-UART-Treiber registriert. Erst diese Registrierung ermöglicht die Zustellung der eingehenden UART-Signale wie in Abbildung 5.6 dargestellt. Zuletzt wird mit dem LogIF-Treiber eine Notiz auf Debuglevel Info mit dem Operator `RTDS_LogIF_LogScreenLL` ausgegeben. Auf dem Imote2 wird die Ausgabe des Operators über die UART-Schnittstelle verschickt, da es hier keinen Screen im klassischen Sinn gibt. Bei der Ausführung auf dem PC würde eine Ausgabe in der Konsole erfolgen. Nach dieser Ausgabe geht der Prozess in den Zustand *idle* über.

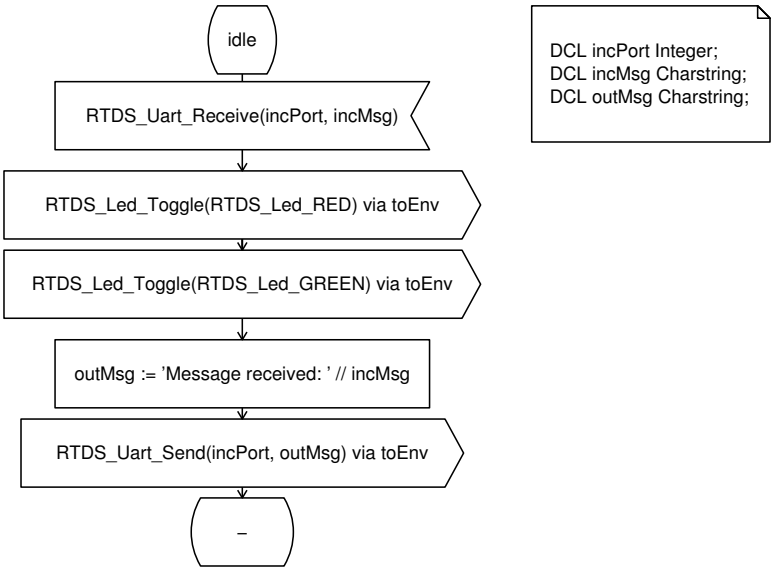

Abbildung 7.7: UART-Verarbeitung im Prozess Demo

Der Zustand *idle* hat zwei Transitionen: Die Verarbeitung eines UART-Signals nach Abbildung 7.7 und die Verarbeitung des Signals *ping* nach Abbildung 7.8. Die UART-Transition beginnt mit dem Empfang des UART-Signals mit zwei Parametern: Port und Nachricht. Die Nachricht wird in der Variable *incMsg* gespeichert. Zusätzlich wird der Port, über den das Signal empfangen wurde, in der Variable *incPort* gespeichert. Um den Empfang auf dem Imote2 zu signalisieren, werden die rote und grüne LED umgeschaltet. Diese werden mit den RTDS_Led_Farbe-Synonymen angesprochen. Da bei der Initialisierung die grüne LED aktiv war, wechseln sich somit die LEDs bei jedem empfangen Signal ab. Als nächstes wird in *outMsg* die empfangene Nachricht mit einer Meldung ergänzt. Diese Konkatenation muss wegen eines Fehlers in RTDS (siehe Kapitel 4.4.2) zwischengespeichert werden. Zuletzt wird die Nachricht in *outMsg* über den selben Port versendet, auf dem das ursprüngliche Signal empfangen wurde. Die Funktionalität lässt sich damit zusammenfassen als Antworten auf eine Nachricht mit einer leicht erweiterten Originalnachricht und einer Signalisierung mit den LEDs.

Die andere Transition beschreibt die Verarbeitung des Signals *ping*. Dieser Verarbeitungsschritt wird mit der blauen LED auf dem Imote2 signalisiert. Zusätzlich wird durch den Operator `RTDS_LogIF_Log` eine Ausgabe erzeugt. Dieser Operator erzeugt auf dem PC zusätzlich zu den Ausgaben von `RTDS_LogIF_LogScreen` eine Eintrag in eine Logdatei (für Details siehe Anhang A.3.3).

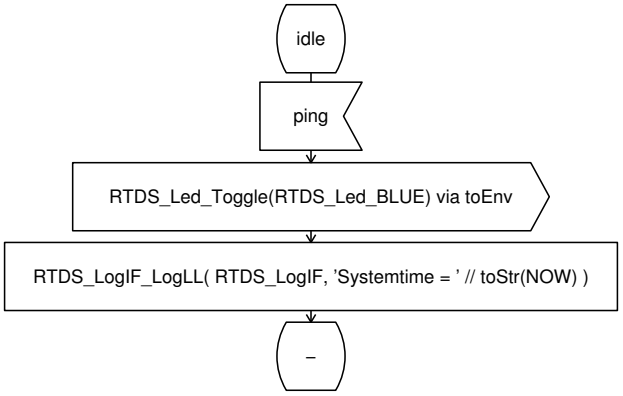

Abbildung 7.8: Verarbeitung des Ping Signals im Prozess Demo

7.5 Block Noise

Der Block *Noise* (siehe Abbildung 7.9) besteht aus zwei Prozessen: *Ping* und *Pong*. Diese bilden das Beispiel aus Kapitel 4.1 nach. Damit wird getestet, dass der neue Scheduler bei ununterbrochenem Signalaustausch seine Verarbeitung korrekt fortsetzt und die Zeit nicht einfriert. Um die Verarbeitung sichtbar zu ma-

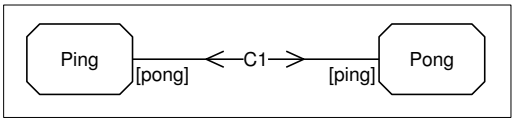

Abbildung 7.9: Block Noise

chen, wurde der Prozess *Ping* gegenüber dem Beispiel in Abbildung 4.2 erweitert. Um die Initialisierung des Prozesses *Ping* auf dem Imote2 anzuzeigen, wird mit

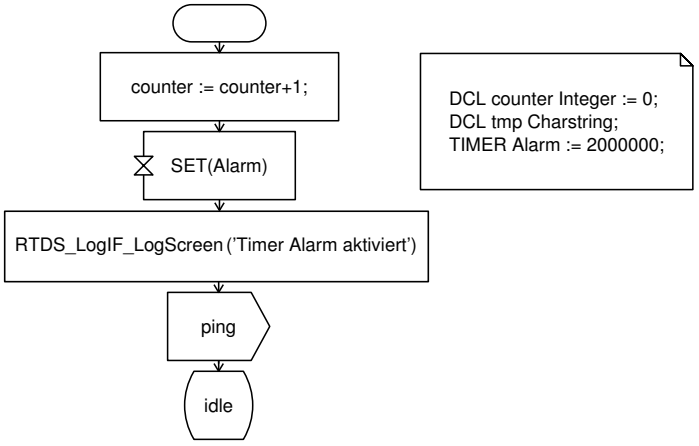

Abbildung 7.10: Initialisierung des Ping-Prozess im Block Noise

`RTDS_LogIF_LogScreen` die Log-Ausgabe „Timer Alarm aktiviert“ erzeugt (siehe Abbildung 7.10). Danach wechselt der Prozess in den Zustand *idle*.

Wenn der Timer *Alarm* ausgelöst wird (siehe Abbildung 7.11, erfolgt eine weitere Log-Ausgabe. Diese zeigt wann der Timer verarbeitet wurde und wie viele *Ping*-

Signale in der Zwischenzeit ausgetauscht wurden. Der Prozess *Pong* ist im wesentlichen identisch zu dem Beispiel in Abbildung 2.2 und wurde daher weggelassen.

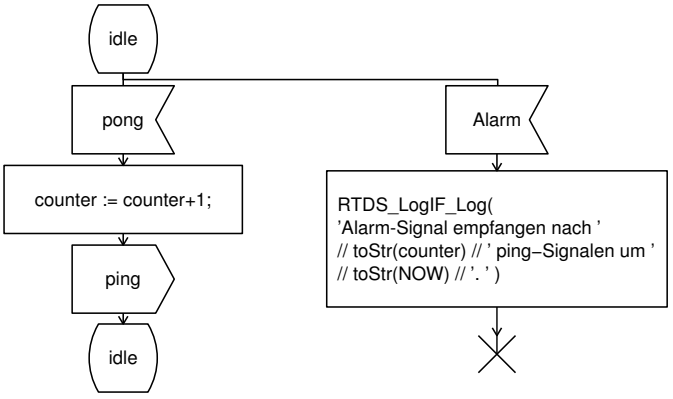

Abbildung 7.11: Signalverarbeitung des Ping-Prozess im Block Noise

7.6 Evaluation der durchgeführten Testläufe

Die folgenden Log-Ausschnitte wurden während zwei separater Testläufe des in diesem Kapitel vorgestellten Systems aufgezeichnet. Der einzige Unterschied zwischen den beiden Testläufen war eine unterschiedliche Einstellung der Schrittweite (siehe Kapitel 5.2). Testlauf 1 wurde mit einer Schrittweite von 1000 μs/Zeiteinheit durchgeführt. Testlauf 2 erfolgte mit einer Schrittweite von nur 1 μs/Zeiteinheit.

Listing 7.1 Zeile 1 und 2 sind die Ausgaben während der Initialisierung der Prozesse *Demo* (Abbildung 7.6) und *Ping* (Abbildung 7.10). Die folgenden Zeilen zeigen die erwartete periodische Verarbeitung von Timern im Prozess *PingClock* (Abbildung 7.4) mit anschließendem Empfang des Signals im Prozess Demo (Abbildung 7.8). Zeile 15 zeigt die erfolgreiche Verarbeitung einer empfangenen UART-Nachricht (Abbildung 7.7).

```
1  Info : Demo started
2  Debug: Timer Alarm aktiviert
3  Debug: Timer gesetzt um 1000 auf 2000.
4  Info : Systemtime = 1000
5  Debug: Timer gesetzt um 2000 auf 3000.
6  Info : Systemtime = 2000
7  Debug: Timer gesetzt um 3000 auf 4000.
8  Info : Systemtime = 3000
9  Debug: Timer gesetzt um 4000 auf 5000.
10 Info : Systemtime = 4000
11 Debug: Timer gesetzt um 5000 auf 6000.
12 Info : Systemtime = 5000
13 Debug: Timer gesetzt um 6000 auf 7000.
14 Info : Systemtime = 6000
15 Message received: RTDS_UART Input Test 1
16 Debug: Timer gesetzt um 7000 auf 8000.
17 Info : Systemtime = 7000
```

Listing 7.1: Testlauf 1 bei Schrittweite von 1000 μs/Zeiteinheit

In Listing 7.2 ist ein späterer Ausschnitt des Logs vom ersten Testlauf zu sehen. Zeile 63 ist die Ausgabe des Prozesses *Ping*, der nach dieser Nachricht terminiert. Dieser

ist zu entnehmen, dass der Prozess *Ping* insgesamt 1188717 *ping*-Signale innerhalb der 30 Sekunden verschickt hat.

```
61 Debug: Timer gesetzt um 29000 auf 30000.
62 Info : Systemtime = 29000
63 Debug: Alarm-Signal empfangen nach 1188717 ping-Signalen um 30000.
64 Debug: Timer gesetzt um 30000 auf 31000.
65 Info : Systemtime = 30000
```

Listing 7.2: Testlauf 1 bei Schrittweite von 1000 μs/Zeiteinheit (Fortsetzung)

Der zweite Testlauf mit der reduzierten Schrittweite von nur 1 μs/Zeiteinheit lieferte den Ausschnitt des Logs in Listing 7.3. Dieses unterscheidet sich von Listing 7.1, weil die Ausführungszeitpunkte der einzelnen Transitionen nicht perfekt auf die eingestellten Werte passen, weil im zweiten Testlauf die Verarbeitungszeit sichtbar wird. Zeile 3 zeigt, dass die Log-Ausgabe 29 μs später erzeugt wurde. Bis zur Verarbeitung des anschließend generierten *ping*-Signals vergingen weitere 203 μs. Diese Verzögerungen waren im ersten Testlauf aufgrund der größeren Timerschrittweite nicht erkennbar, sind aber bei der Genauigkeit im zweiten Testlauf zu erwarten.

```
1 Info : Demo started
2 Debug: Timer Alarm aktiviert
3 Debug: Timer gesetzt um 1029 auf 2000.
4 Info : Systemtime = 1232
5 Debug: Timer gesetzt um 2026 auf 3000.
6 Info : Systemtime = 2137
```

Listing 7.3: Testlauf 2 mit Schrittweite von 1 μs/Zeiteinheit

Beide Testläufe demonstrieren, dass die RTDS-Treiber für UART und LogIF funktionieren. Der RTDS-LED-Treiber funktionierte ebenfalls wie erwartet. Im ersten Testlauf war das periodische Blinken erkennbar und das Umschalten der LEDs bei UART-Nachrichtenempfang funktionierte in beiden Testläufen.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Integration von SDL in BiPS auf Basis von PragmaDev's RTDS durchgeführt. Dafür wurde die Codegenerierung von RTDS analysiert und Veränderungen an den vorhandenen Templates durchgeführt. Anschließend wurden RTDS-Treiber implementiert und ihre Funktionen getestet.

Hierzu wurde zunächst eine kurze Einführung in SDL, BiPS, RTDS und dem Imote2 gegeben. Danach wurde die Codegenerierung von RTDS untersucht. Dabei wurde die Struktur eines RTDS-Systems betrachtet und die Aufgaben der generierten Dateien ermittelt. Anschließend wurde das Verhalten des Codegenerators und der RTDS-Systeme mit dem *rtosless*-Template genauer durchleuchtet. Dabei wurden nicht dem SDL-Standard entsprechende Implementierungen festgestellt und Korrekturen durchgeführt sowie Workarounds beschrieben. Danach erfolgte die Implementierung der Schnittstelle zwischen RTDS und BiPS. Diese wurde mit dem SDL-Environment und RTDS-Treibern realisiert. Zentrale Aspekte der Schnittstelle wurden detailliert dargestellt. Anschließend wurde gezeigt, wie eine Applikation aus der SDL-Spezifikation erzeugt wird. Zuletzt wurde zur funktionaler Evaluation der Implementierung ein Beispiel präsentiert, das alle erstellten RTDS-Treiber umfasst. An dem Demo-System wurden Besonderheiten der RTDS-Systeme nochmal hervorgehoben.

Das Demo-System zeigt, dass die Erzeugung von auf BiPS-basierten SDL-Applikationen funktioniert. Der nächste Schritt ist die Erstellung von RTDS-Treibern für weitere BiPS-Funktionalitäten, wie z.B. zur Anbindung an den CC2420-Transceiver. Mit den zugesagten Änderungen von PragmaDev, kann das Überlaufproblem (siehe Kapitel 4.3) komplett behoben werden.

Eine Chance bietet die Verwendung von *FreeRTOS* mit BiPS. Dadurch können manche SDL-Prozesse priorisiert behandelt werden. Die Erfahrungen des *rtosless*-Templates lassen sich auf das *FreeRTOS*-Template übertragen. Die erstellten RTDS-Treiber sollten mit geringem Aufwand angepasst werden können. Für die Evaluation von großen Testszenarien mit vielen Knoten ist ebenfalls eine Anbindung von RTDS an FERAL [BGFK13, BCG+13] sinnvoll.

Anhang A

Compileroptionen und Treiberdokumentation

A.1 Konfiguration des Timer-SET-Befehls

A.1.1 Konfiguration der Timer-Verarbeitung

Mit der Erweiterung wird der delay in den SET-Befehlen standardmäßig als absoluter Wert interpretiert. Dies ist eine Folge des in Kapitel 4.3 angesprochenen Verhaltens bei der Codegenerierung, die normalerweise nur relative Timer erlauben würde. Um die Auswertung der Befehle wieder auf relative Timer umzustellen, muss in den Code Generation Options die Compiler-Option '**-D RTDS_Relative_Timer**' angegeben werden.

A.1.2 Einstellen der Timer-Schrittweite

Standardmäßig ist die Timer-Schrittweite auf 1000 μs/(SDL-)Zeiteinheit eingestellt. Daraus ergibt sich folgende Beziehung für Timer-Werte und tatsächliche Zeit:

Formel: Timerwert $\times$ Schrittweite = tatsächliche Zeit
Beispiel: 1 Zeiteinheit $\times$ 1000 μs/Zeiteinheit = 1 ms

Diese Schrittweite wird mit RTDS_TIME_STEP in rtosless/RTDS_OS.h definiert. Bei Verwendung der höchsten Genauigkeit (1μs/Zeiteinheit) ist die Überlaufproblematik zu beachten.

A.2 Deaktivierung des neuen RTDS-Schedulers

Standardmäßig wird der neue RTDS-Scheduler verwendet, der in Kapitel 4.5.2 beschrieben wurde. Um den alten RTDS-Scheduler (siehe Kapitel 4.1) zu verwenden, muss in den Code Generation Options die Compiler-Option '**-D RTDS_OLD_RUN**' angegeben werden.

A.3 Überblick über die RTDS-Treiber

Im Verlauf der Entwicklung der Schnittstelle wurden die ersten Pakete für SDL erstellt. Der Inhalt dieser Pakete und die Auswirkung der Signale im Environment wird in den folgenden Abschnitten erläutert.

A.3.1 LED

Mit dem RTDS_Led Paket lassen sich die drei LEDs des Imote2 Knotens steuern.

Synonyme:

Name	Typ	Wert
RTDS_Led_RED	Integer	1
RTDS_Led_GREEN	Integer	2
RTDS_Led_BLUE	Integer	4

ausgehende Signale:

Name	Funktionalität	Parameter → Typ
RTDS_Led_Toggle (LED-ID)	Wechselt den Zustand der LED	LED-ID → Integer
RTDS_Led_Set (LED-ID, Zustand)	Setzt den Zustand der LED auf den angegeben Wert	LED-ID → Integer Zustand → Boolean

Signallisten:

Name	Signale
SDL2LED	RTDS_Led_Toggle, RTDS_Led_Set

```
1  /* Synonym List */
2  SYNONYM RTDS_Led_RED Integer = 1;
3  SYNONYM RTDS_Led_GREEN Integer = 2;
4  SYNONYM RTDS_Led_BLUE Integer = 4;
5
6  /* Signal Definition */
7  /* Outgoing Signals */
8  SIGNAL RTDS_Led_Toggle(Integer);
9  SIGNAL RTDS_Led_Set(Integer,Boolean);
10
11 /* Pseudo-Signal for Definition in RTDS_gen.h - used to trigger Driver */
12 SIGNAL RTDS_EnvObserver_Led;
13
14 /*Signallist*/
15 SIGNALLIST SDL2LED = RTDS_Led_Toggle, RTDS_Led_Set;
```

Listing A.1: SDL-Declaration Datei von RTDS_Led

A.3.2 UART

Mit dem UART-Treiber lassen sich Rahmen über die UART-Schnittstelle des Imote2 versenden und empfangen. Zusätzlich sind im RTDS-Treiber Signale für die Konfiguration bereitgestellt. Das Signal RTDS_Uart_Init ist **notwendig**, damit UART-Signale empfangen werden können.

Ausgehende Signale:

Name	Funktionalität	Parameter → Typ
RTDS_Uart_Init (Port)	Registriert sendenden Prozess für Empfang auf Port [**notwendig**]	Port → Integer
RTDS_Uart_SetBaudrate (Port, Baudrate)	Ändert die Baudrate auf dem Port	Port → Integer Baudrate → Integer
RTDS_Uart_Send (Port, Nachricht)	Sendet einen String über den Port	Port → Integer Nachricht → Charstring

Eingehende Signale:

Name	Funktionalität	Parameter → Typ
RTDS_Uart_Receive (Port, Nachricht)	Nachricht enthält den empfangenen UART-String	Port → Integer Nachricht → Charstring

Signallisten:

Name	Signale
SDL2UART	RTDS_Uart_SetBaudrate, RTDS_Uart_Send, RTDS_Uart_Init
UART2SDL	RTDS_Uart_Receive

```
1  SYNONYM UART_ST Integer = 2;
2  SYNONYM UART_FF Integer = 1;
3  SYNONYM UART_BT Integer = 0;
4
5  /* Signal Definition */
6  /* Outgoing Signals */
7
8  /* Initialize necessary for Callbacks */
9  /* Integer Portnumber */
10 SIGNAL RTDS_Uart_Init(Integer);
11
12 /* Integer Portnumber, Integer Baudrate*/
13 SIGNAL RTDS_Uart_SetBaudrate(Integer,Integer);
14
15 /* Integer Portnumber, Integer Message */
16 SIGNAL RTDS_Uart_Send(Integer,Charstring);
17
18 /* Incoming Signals */
19 /* Integer Portnumber, Charstring Message */
20 SIGNAL RTDS_Uart_Receive(Integer,Charstring);
21
22 /* Pseudo-Signal */
23 SIGNAL RTDS_EnvObserver_Uart;
24
25 /*Signallist*/
26 SIGNALLIST SDL2UART = RTDS_Uart_SetBaudrate, RTDS_Uart_Send, RTDS_Uart_Init;
27 SIGNALLIST UART2SDL = RTDS_Uart_Receive;
```

Listing A.2: SDL-Declaration Datei von RTDS_Uart

A.3.3 LogIF

Durch den LogIF-Treiber wird es ermöglicht, Log-Ausgaben zu nutzen. Auf dem Imote2 wird dafür die UART-Schnittstelle eingesetzt, während unter Linux eine Ausgabe auf der Konsole und in eine Datei möglich ist.

Synonyme:

Name	Typ	Wert
RTDS_LogIF_TRACE	Integer	1
RTDS_LogIF_DEBUG	Integer	2
RTDS_LogIF_INFO	Integer	3
RTDS_LogIF_FATAL	Integer	4
RTDS_LogIF_ERROR	Integer	5

Operatoren:

Name	Funktionalität	Parameter → Typ
RTDS_LogIF_Log (Nachricht)	Kombination von LogFile und LogScreen	Nachricht → Charstring
RTDS_LogIF_LogLL (LogLevel, Nachricht)	RTDS_LogIF_Log mit Log-Level Angabe	LogLevel → Integer Nachricht → Charstring
RTDS_LogIF_LogFile (Nachricht)	Loggt die Nachricht in eine Datei (nur Linux)	Nachricht → Charstring
RTDS_LogIF_LogFileLL (LogLevel, Nachricht)	LogFile Befehl mit Log Level Angabe	LogLevel → Integer Nachricht → Charstring
RTDS_LogIF_LogScreen (Nachricht)	Loggt die Nachricht auf Konsole bzw. UART-Port	Nachricht → Charstring
RTDS_LogIF_LogScreenLL (LogLevel, Nachricht)	LogScreen Befehl mit Angabe des Log-Levels	LogLevel → Integer Nachricht → Charstring
RTDS_LogIF_SetLogLvl (LogLevel)	Ändert den Default Log-Level auf den angegeben Wert	LogLevel → Integer
toStr(Wert)	wandelt den Wert zu einem Charstring um [überladener Operator]	Wert → Integer o. Wert → Natural o. Wert → Real o. Wert → Boolean o. Wert → Time
add(String,Wert)	wandelt den Wert zu einem Charstring um und hängt diesen an den bestehenden String [überladener Operator]	String → Charstring Wert → Integer o. Wert → Natural o. Wert → Real o. Wert → Boolean o. Wert → Time

Die Voreinstellung einiger Werte lässt sich mit folgenden Compileroptionen, die unter Code Generation Options gesetzt werden können, verändern:

- **- D RTDS_LogIF_LOGFILE=X.log** Ändert den Namen der Log Datei zu X.log.
  Standard-Name ist RTDS_LogIF_LOG.txt (**nur Linux**)
- **- D RTDS_LogIF_PORT=X** Ändert den verwendeten UART-Port für Log-Ausgaben auf X. Erlaubte Optionen sind 0 (UART_BT), 1 (UART_FF) und 2 (UART_ST)[Standard].

Standardmäßig werden nur Log-Ausgaben verarbeitet die mindestens vom Typ Debug sind. Dies bedeutet das Trace-Ausgaben nicht angezeigt werden. Dieses Verhalten lässt sich mit den folgenden Optionen ändern (absteigende Reihenfolge):

- **-D RTDS_LogIF_ERROR**
- **-D RTDS_LogIF_FATAL**
- **-D RTDS_LogIF_INFO**
- **-D RTDS_LogIF_DEBUG**
- **-D RTDS_LogIF_TRACE**

Die Option setzt den angegeben Typ als niedrigste Stufe, die verarbeitet wird. Dies bedeutet, dass z.B. mit der Option **-D RTDS_LogIF_FATAL** nur Log-Ausgaben der Stufen FATAL und ERROR verarbeitet werden. Wenn mehrere Optionen gleichzeitig verwendet werden, überschreibt die kleinste angegebene Stufe die anderen Optionen.

```
1  /* Operator Definition */
2  NEWTYPE RTDS_LogIF_StringConvert
3    OPERATORS
4    add: Charstring, Integer -> Charstring;
5    add: Charstring, Natural -> Charstring;
6    add: Charstring, Real -> Charstring;
7    add: Charstring, Boolean -> Charstring;
8    add: Charstring, Time -> Charstring;
9    toStr: Integer -> Charstring;
10   toStr: Natural -> Charstring;
11   toStr: Real -> Charstring;
12   toStr: Boolean -> Charstring;
13   toStr: Time -> Charstring;
14 ENDNEWTYPE;
15
16 NEWTYPE RTDS_LogIF_LogOperations
17   OPERATORS
18   /* Log to File and Screen*/
19   RTDS_LogIF_Log:         Charstring -> Integer;
20   /* with specified LogLevel */
21   RTDS_LogIF_LogLL:       Integer,Charstring -> Integer;
22   /* Log to File only */
23   RTDS_LogIF_LogFile:     Charstring -> Integer;
24   /* with specified LogLevel */
25   RTDS_LogIF_LogFileLL:   Integer,Charstring -> Integer;
26   /* Log to Screen only */
27   RTDS_LogIF_LogScreen:   Charstring -> Integer;
28   /* with specified LogLevel */
29   RTDS_LogIF_LogScreenLL: Integer,Charstring -> Integer;
30   /* Change LogLevel - DEBUG is default */
31   RTDS_LogIF_SetLogLvl:   Integer -> Integer;
32 ENDNEWTYPE;
33
34 /* Synonym List */
35 SYNONYM RTDS_LogIF_TRACE  Integer = 1;
36 SYNONYM RTDS_LogIF_DEBUG  Integer = 2;
37 SYNONYM RTDS_LogIF_INFO   Integer = 3;
38 SYNONYM RTDS_LogIF_FATAL  Integer = 4;
39 SYNONYM RTDS_LogIF_ERROR  Integer = 5;
40
41 /* Signal Definition */
42
43 /* Pseudo-Signal for Definition in RTDS_gen.h - used to trigger Driver */
44 SIGNAL RTDS_EnvObserver_LogIF;
```

Listing A.3: SDL-Declaration Datei von RTDS_LogIF

Literaturverzeichnis

[AG] AG Vernetzte Systeme: *Homepage*. `http://vs.informatik.uni-kl.de`.

[BCG09] Becker, Philipp, Dennis Christmann und Reinhard Gotzhein: *Model-Driven Development of Time-critical Protocols with SDL-MDD*. In: Reed, Rick et al. [RBG09], Seiten 34–52.

[BCG+13] Braun, Tobias, Dennis Christmann, Reinhard Gotzhein, Anuschka Igel, Thomas Forster und Thomas Kuhn: *Virtual Prototyping with Feral – Adaptation and Application of a Simulator Framework*. In: *The 24th IASTED International Conference on Modelling and Simulation*, 2013.

[BGFK13] Braun, T., R. Gotzhein, T. Forster und T. Kuhn: *FERAL – Framework for Simulator Coupling on Requirements and Architecture Level*. In: *Eleventh ACM-IEEE International Conference on Formal Methods and Models for Codesign, 18-20 Oktober*, 2013.

[CGR12] Christmann, D., R. Gotzhein und S. Rohr: *The Arbitrating Value Transfer Protocol (AVTP) - Deterministic Binary Countdown in Wireless Multi-Hop Networks*. In: *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, Seiten 1 –9, aug 2012.

[Cro07] Crossbow: *Imote 2 Hardware Reference Manual*. `http://agvs.cs.uni-kl.de/downloads/Imote2_Hardware_Reference_Manual.pdf`, 2007.

[Eng13] Engel, Markus: *Optimierung und Evaluation Black Burst-basierter Protkolle unter Verwendung der Imote 2-Plattform*. Diplomarbeit, TU Kaiserslautern, 2013.

[FGJ+05] Fliege, Ingmar, Alexander Geraldy, Simon Jung, Thomas Kuhn, Christian Webel und Christian Weber: *Konzept und Struktur des SDL Environment Framework (SEnF)*. Technischer Bericht 341/05, TU Kaiserslautern, 2005.

[FGW06] Fliege, Ingmar, Rüdiger Grammes und Christian Weber: *ConTraST – A Configurable SDL Transpiler and Runtime Environment*. In: Gotzhein, Reinhard und Rick Reed [GR06], Seiten 216–228.

[GK08] Gotzhein, Reinhard und Thomas Kuhn: *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*, Seiten 422–431, June 2008.

[GK11] Gotzhein, Reinhard und Thomas Kuhn: *Black Burst Synchronization (BBS) - A Protocol for Deterministic Tick and Time Synchronization in Wireless Networks.* Computer Networks, 55(13):3015–3031, 2011.

[Got07] Gotzhein, Reinhard: *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*, Seiten 31–46, June 4-8 2007.

[GR06] Gotzhein, Reinhard und Rick Reed (Herausgeber): *System Analysis and Modeling - Language Profiles*, Band 4320 der Reihe *Lecture Notes in Computer Science*. Springer, 2006.

[IBM] IBM Rational: *IBM Rational SDL Suite.* `http://www.ibm.com/developerworks/rational/products/sdlsuite/`.

[IEE03] 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, New York, NY, USA, Oct. 2003.

[Int12a] International Telecommunication Union (ITU): *ITU-T Recommendation Z.100 (12/11) – Specification and Description Language – Overview of SDL 2010*, 2012.

[Int12b] International Telecommunication Union (ITU): *ITU-T Recommendation Z.101 (12/11) – Specification and Description Language - Basic SDL-2010*, 2012.

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

[Prab] PragmaDev SARL: *Real Time Developer Studio V4.4 Reference Manual.* `http://www.pragmadev.com`.

[Prac] PragmaDev SARL: *Real Time Developer Studio V4.4 User Manual.* `http://www.pragmadev.com`.

[RBG09] Reed, Rick, Attila Bilgic und Reinhard Gotzhein (Herausgeber): *SDL 2009: Design for Motes and Mobiles, 14th International SDL Forum, Bochum, Germany, September 22–24, 2009, Proceedings*, Band 5719 der Reihe *LNCS*. Springer, 2009.

[RJG94] Ralph Johnson, John Vlissides, Richard Helm und Erich Gamma: *Design Patterns. Elements of Reusable Object-Oriented Software.* 1994.

[SDL13] SDL-RT Consortium: *SDL-RT – Specification & Description Language – Real Time V2.3).* `http://www.sdl-rt.org/standard/V2.3/pdf/SDL-RT.pdf`, 2013.

[Tex13] Texas Instruments: *CC2420 datasheet.* `http://www.ti.com/lit/ds/symlink/cc2420.pdf`, 2013. Revision SWRS041c.