Technik
Informatik & Medien
Hochschule Ulm
University of
Applied Sciences

Fakultät Elektrotechnik und
Informationstechnik

Codegenerierung aus UML Aktivitäts-diagrammen und Implementierung einer Ethernet Schnittstelle für Embedded Systems

Master-Projektarbeit
an der Hochschule Ulm

Studiengang Systems Engineering and Management,
Vertiefungsrichtung Electrical Engineering

Bearbeiter:

Timo Steinmeyer
Studiengang SYE 2
Matrikelnr.: 3117247

Stefan Pollithy
Studiengang SYE 2
Matrikelnr.: 3117268

Betreuerin: Prof. Dr. rer. nat. M. von Schwerin

Bearbeitungszeitraum: WS 2014/2015

Abgegeben am: 27.02.2015

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CPU ..... Central Processing Unit

IDE ..... Integrated Development Environment

OMD ..... Object Model Diagramm

RTOS ..... Real-Time Operating System

RXF ..... Real-time Execution Framework

TCP/IP ..... Transmission Control Protocol/Internet Protocol

UDP ..... User Datagram Protocol

UML ..... Unified Modeling Language

1 Einleitung

Dieses Kapitel beschreibt einleitend die Motivation dieser Arbeit. Anschließend wird ein Überblick über den Stand der Dinge gegeben. Schließlich werden die zu bearbeitenden Aufgaben vorgestellt.

1.1 Motivation dieser Arbeit

In dieser Arbeit soll aus Unified Modeling Language (UML) Aktivitätsdiagrammen automatisch Code für ein Target (MCB1700) generiert werden.

Die UML ist eine Sprache, die unabhängig von Fach- und Realisierungsgebiet zur Modellierung, Dokumentation, Spezifizierung und Visualisierung komplexer Systeme eingesetzt werden kann.

Im Rahmen der Veranstaltung Embedded Systems wurde die Autocodegenerierung bereits verwendet. Bereits dort hat uns beeindruckt, wie strukturiert, übersichtlich und einfach der Code generiert werden kann. Jedoch wurden dort bisher lediglich Zustands-diagramme eingesetzt, woraus sich die Thematik für diese Studienarbeit ergab. Die Codegenerierung aus Aktivitätsdiagrammen soll untersucht werden.

Des Weiteren spielt die Autocodegenerierung aus UML eine zunehmend wichtige Rolle in der Industrie. Vermehrt wird sie in Betrieben eingesetzt, weswegen eine Wissensvertiefung auf diesem Gebiet sehr vorteilhaft für das spätere Berufsleben ist. Beispielsweise wird bei der Firma HILTI aus Zustands- und Aktivitätsdiagrammen automatisch ein Code für die Steuerung verschiedenster Bohrmaschinen generiert.

Hinzu kommt die Herausforderung, neue Abläufe und Funktionen kennenzulernen. Während unseres Erststudiums konnten wir nur begrenzte Kenntnisse auf dem Gebiet der Programmierung sowie der Embedded Betriebssysteme erlangen. Wir wollen uns in dem Fachgebiet weiterbilden und haben uns daher für dieses Thema als Studienarbeit entschieden.

1.2 Stand der Dinge

Wie bereits erwähnt, wurde während der Vorlesung Embedded Systems mit dem UML-Modellierungstool Rhapsody und dem Willert Software Tools RXF (Real-time Execution Framework) automatisch Code aus UML Diagrammen für ein Target (MCB1700) erstellt. In Laborübungen wurden bestimmte Aufgaben bearbeitet, wie beispielsweise das Blinken einer Led mit einer bestimmten Frequenz.

Die Aufgabe wurde folgendermaßen realisiert. Es wurde ein Klassendiagramm angelegt, welchem ein Zustandsdiagramm hinterlegt wurde. Das Klassendiagramm stellt die Led mit ihren Operationen (off und on) dar und das Zustandsdiagramm beschreibt das Verhalten, was beispielsweise bei Ablauf eines Timers geschehen soll (sog. toggeln).

Zustandsdiagramme werden eingesetzt, um zu beschreiben, wie auf äußere Einflüsse reagiert werden soll. Ein äußerer Einfluss kann das Drücken eines Tasters, das Verändern eines Potentiometers oder der Ablauf eines Timers sein. Solche ereignisgetriebene (zeitdiskrete) Aufgaben, wie das Blinken einer Led mittels eines Timers, lassen sich sehr elegant mit einem Zustandsdiagramm modellieren.

Sollen nun aber Signale (aus zeitlicher Sicht kontinuierlich gültige Werte z.B. in Form von Variablen) erfasst werden, so lässt sich dies wesentlich eleganter mit einem ablauforientierten Diagramm, dem Aktivitäts-diagramm, realisieren.

Verwendet man zur Erfassung von kontinuierlichen Signalen Zustands-diagramme anstelle von Aktivitätsdiagrammen, so kann sich eine Problematik, nämlich ein Fehlverhalten wie in Abschnitt 3.4 beschrieben, ergeben.

Deswegen sind die Zustandsdiagramme nicht für alle Aufgaben geeignet und für manche Anwendungen müssen Aktivitätsdiagramme eingesetzt werden.

1.3 Aufgaben

Die Arbeit gliedert sich in folgende zwei Aufgabengebiete.

- Codegenerierung aus Aktivitätsdiagrammen
- Einbindung der Ethernet Schnittstelle des MCB1700

Der erste Teil der Arbeit befasst sich mit der Autocodegenerierung aus Aktivitätsdiagrammen, um zeitkontinuierliches Verhalten modellieren zu können. Es gilt herauszufinden, inwieweit es möglich ist, funktionsfähige UML Modelle und Code aus Aktivitätsdiagrammen zu generieren. Dies soll mittels eines einfachen Beispiels getestet und die daraus resultierenden Erkenntnisse vorgestellt werden. Aufgrund einer sich ergebenden Problematik wurde das ursprüngliche Thema beendet und um folgende Aufgabe erweitert.
Diese umfasste die Einbindung der Ethernet Schnittstelle des MCB1700, sodass das Senden und Empfangen von Nachrichten über Ethernet in einem UML-Modell modelliert werden kann.

2 Plattform und Werkzeuge

Das folgende Kapitel stellt die verwendete Plattform und Werkzeuge vor. Als Plattform wird das MCB1700 Evaluationsboard von Keil verwendet. Die Werkzeugkette besteht aus den Programmen Rational Rhapsody, Willert Software Tools RXF und der Embedded Integrated Development Environment (IDE) Keil-µVision.
Im Folgenden wird auf das MCB1700 Evaluationsboard eingegangen und dessen Funktionen vorgestellt.

2.1 Das MCB1700 Evaluationsboard

Im Rahmen dieser Arbeit wird das MCB1700 Evaluationsboard von Keil verwendet.

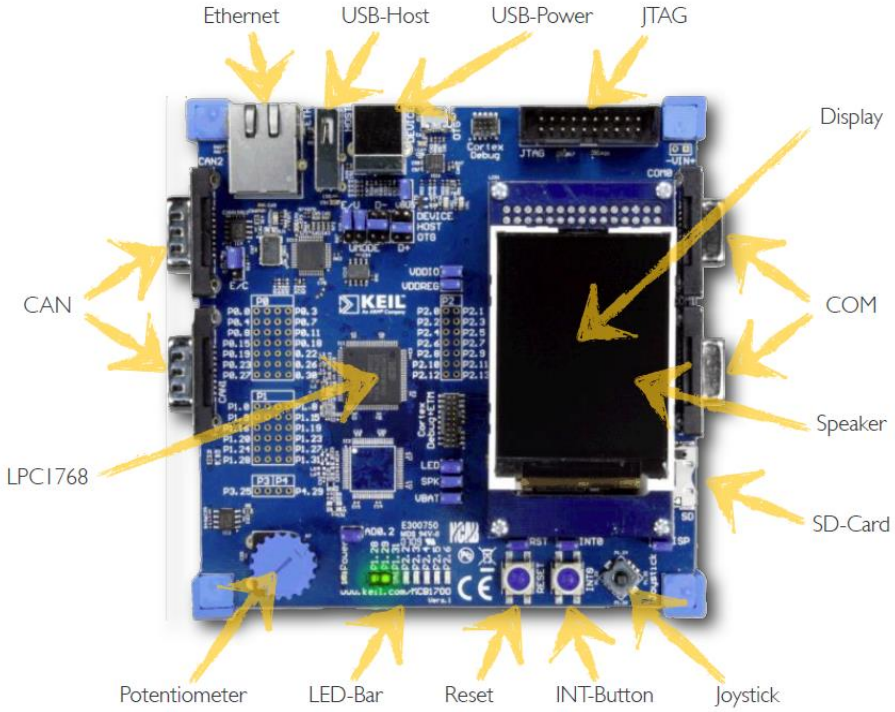


Abb. 1 MCB1700 Board [6]

Das MCB1700 Evaluationsboard bietet eine Vielzahl von Funktionen, welche im Folgenden aufgelistet sind.

- 100MHz LPC1768 ARM Cortex™-M3 processor-based MCU in 100-pin LQFP
- On-Chip Memory: 512KB Flash & 64KB RAM
- Color QVGA TFT LCD
- 10/100 Ethernet Port
- USB 2.0 Full Speed - USB, USB-OTG, & USB Host
- CAN Interfaces
- Serial Ports
- SD/MMC Card Interface
- 5-position Joystick and push-button
- Analog Voltage Control for ADC Input
- Amplifier and Speaker
- 70 GPIO pins
- Debug Interface Connectors

Auf dem MCB1700 Board ist der Microcontrolller NXP LCP1768 im Einsatz. Im Anhang 8.1 befindet sich ein Blockdiagramm des NXP LCP1768. Anhand des Blockdiagramms erhält man eine Übersicht über die onChip Peripherien des NXP.

2.2 Toolchain zur Codegenerierung aus Aktivitätsdiagrammen

Wie bereits zu Beginn des Kapitels erwähnt, umfasst die Toolchain (siehe Abb. 2) die grafische Entwicklungsumgebung Rational Rhapsody Developer 8.0.1 von IBM [Link], das Willert Software Tools RXF (Eval Version V6.01) [Link] und die Embedded-IDE Eval Version Keil-µVision 5 [Link]. In Rhapsody werden UML-Diagramme erstellt und daraus ein ANSI C-Code generiert. Keil µVision wird benötigt, um den Build-Prozess durchzuführen, das ausführbare Model zu flashen oder es in einem Simulator laufen zu lassen. Das RXF von Willert kombiniert die Keil µVision IDE mit Rhapsody. Dieses Zusammenspiel wird in Form der roten Pfeile in Abb. 2 verdeutlicht. (vgl. [2])

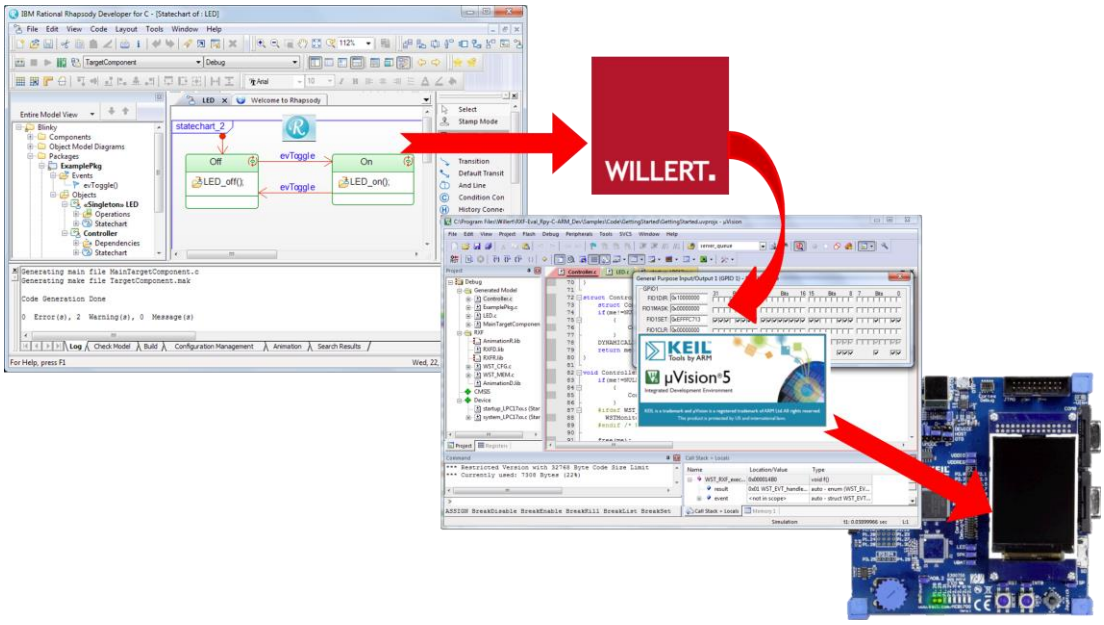


Abb. 2 Toolchain (vgl. [2])

2.2.1 Rhapsody

Rational Rhapsody ist eine graphische Entwicklungsumgebung basierend auf der Unified Modeling Language. Rhapsody kann zur Entwicklung von Embedded-, Echtzeit- oder technischer Anwendungssoftware eingesetzt werden, ist jedoch nicht nur auf diese Bereiche beschränkt (Modellbasierte Systementwicklung mit SysML, …). (vgl. [3])

Für die vorliegende Arbeit wird die Version 8.0.1 von Rhapsody verwendet, welche sich auf einer Demo-DVD befindet. Diese steht auf der Homepage von Willert zum Download zur Verfügung. Dabei besteht auch die Möglichkeit, eine zweimonatige Testlizenz zu beantragen. Alternativ kann auch die Hochschullizenz verwendet werden.

In Rhapsody werden UML-Diagramme erstellt und daraus ein ANSI C-Code generiert. Ohne das Verwenden weiterer Tools kann dieser Code aber nicht direkt auf einem Target verwendet werden, da der Code-generator von Rhapsody beispielsweise keine entsprechenden Notations-elemente für die Benutzung eines Timers auf dem NXP LPC1768 kennt. Hierfür ist ein weiteres Tool notwendig, das Embedded UML RXF von Willert, das in Zusammenarbeit mit Rhapsody einen ausführbaren Code für den NXP LPC1768 erstellt.

Im Folgenden wird das RXF von Willert genauer beschrieben.

2.2.2 Willert Software Tools RXF

Das Willert Software Tools RXF beinhaltet neben dem Embedded UML RXF auch den WSTDeployer. Diese beiden Komponenten sind essentiell für die automatische Codeerzeugung aus UML-Modellen für eine Zielplattform und werden daher im Folgenden genauer vorgestellt.

2.2.2.1 WSTDeployer

Der Deployer ist für die Integration der generierten Dateien und dem RXF in eine IDE zuständig und wird durch das Betätigen des Buttons Run in Rhapsody gestartet. (vgl. [8])

Die Vorgehensweise umfasst üblicherweise folgende Schritte:

1. Code aus UML-Modell in Rhapsody erzeugen.
2. Beim erstmaligen Aufrufen des Deployers muss das Ziel µVision Keil 5 Projekt angegeben werden.
3. Die erzeugten Dateien werden in den vorher ausgewählten Ordner kopiert, in dem sich auch das Keil Projekt befindet.
4. Die benötigten RXF Dateien werden in dasselbe Verzeichnis kopiert.
5. Keil-µVision 5 aktualisiert bei Änderung der UML-Modelle in Rhapsody und anschließend erneuter Codegeneration das Projekt automatisch.

Konfiguration des Deployers

Wird der Deployer zum ersten Mal gestartet, so erscheint folgendes Fenster.

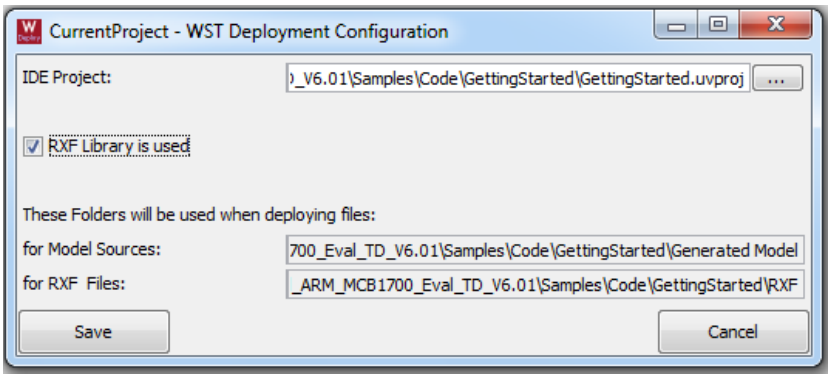


Abb. 3 Deployer Konfiguration

Hier kann über den Browse Button das IDE Projekt gewählt werden. Will man nicht jede einzelne RXF Datei zur IDE hinzufügen, so kann man die Checkbox „RXF Library is used" setzen. Dadurch werden nur wenige Dateien, wie WST_CFG.c, WST_MEM.c, WSTTarget.c, welche beispielsweise plattformabhängige Konfigurationen enthalten, in das Projekt geladen. Dies erfordert jedoch, dass man aus den RXF Dateien eine Bibliothek erstellt und diese manuell dem IDE Projekt hinzufügt. Die Deploy Konfigurationen können nachträglich auch nochmals geändert werden. Dafür steht ein entsprechender Eintrag im Untermenü Tools in Rhapsody zur Verfügung. (vgl. [8])
Im Folgenden sind die Features des Deployers aufgeführt, welche in Verbindung mit Keil-µVision 5 automatisch durchgeführt werden.

- Dateien aus dem Codegenerierungsverzeichnis von Rhapsody automatisch in das Projektzielverzeichnis des IDE Projekts kopieren
- Neue Dateien automatisch zum Keil Projekt hinzufügen
- Automatisches Löschen von umbenannten oder gelöschten Dateien im Keil Projekt (Roundtrip in Rhapsody)
- Wiedergabe der Package und Subpackage Struktur in Keil als Gruppen
- Verwendung von Standardbezeichnungen für die Gruppen, in welche die RXF und generierten Modelldateien eingefügt werden (Ordner „RXF" und „Generated Model")
- Keil µVision5 frägt automatisch nach, ob das Projekt nach dem Ändern der Projektfiles neu geladen werden soll

Der Deployer kann auch für individuelle Zwecke angepasst werden, indem man ihm mitteilt, welche Dateien er zum IDE Projekt hinzufügen soll. Dadurch besteht die Möglichkeit, z.B. nicht alle Dateien des RXF zu laden, um anschließend manuell abgeänderte oder sogar zusätzliche Dateien hinzuzufügen.
Des Weiteren unterstützt er das Entfernen von Dateien aus dem IDE Projekt, nachdem bereits ein Deploy Vorgang durchgeführt wurde. Im Menü Code „Clean Redundant Source Files" wird diese Funktion gestartet, wodurch geänderte oder umbenannte Dateien entfernt werden.

Es bestehen weitere Einstellungsmöglichkeiten für den Deployer. An dieser Stelle sei auf die Dokumentation im Willert Installationsverzeichnis verwiesen. (vgl. [8])

2.2.2.2 Willert Embedded UML RXF

Das Willert Embedded UML RXF ist die Schnittstelle zwischen einem UML-Modell, welches hier in Rhapsody erstellt wird, und einer Zielplattform. Die Bestandteile der Zielplattform sind CPU (ARM Cortex M3), Compiler und Laufzeitsystem (Embedded OO RTX) oder RTOS.
Das RXF beinhaltet einen Adapter, der die Verwendung mit den meist verbreiteten Echtzeitbetriebssystemen ermöglicht. Beispiele für unterstützte Betriebssysteme sind das embOs, OSEK, EUROS und das Keil RTX, welches in dieser Arbeit noch Verwendung findet. Dieser Adapter ermöglicht das Verwenden von Events oder Timern unabhängig vom Betriebssystem und erhöht somit die Portierbarkeit und Wiederverwendbarkeit.
Durch das Embedded UML RXF wird aus dem Rhapsody UML Modell direkt ein ausführbarer Code für ein Target erstellt. Dazu gehört auch das Erstellen entsprechender Makefiles und Compiler Kommandos sowie die Speicherkonfiguration. (vgl. [4])

Folgende Grafik zeigt alle Komponenten, welche zur automatischen Code-generierung benötigt werden.

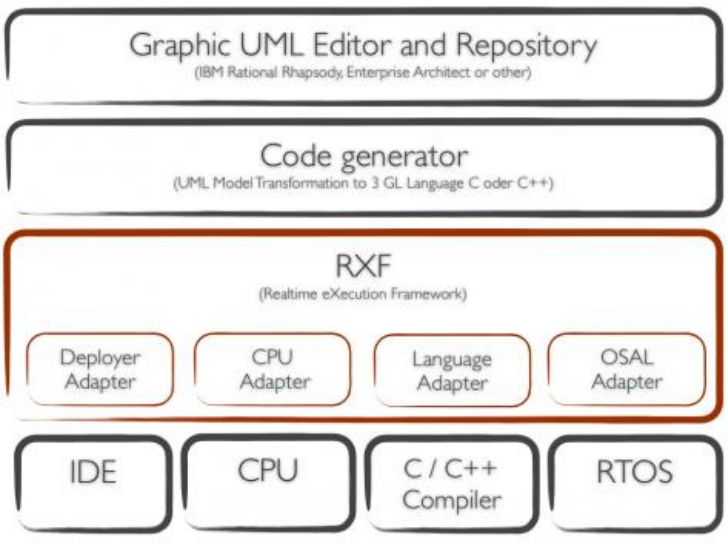


Abb. 4 Komponenten Codegenerierung (vgl. [5])

Als Graphic UML Editor wird Rhapsody zur Modellierung von UML Modellen verwendet. Außerdem beinhaltet Rhapsody den Codegenerator, welcher das UML Modell in eine Hochsprache (ANSI C) übersetzt. Wie bereits in 2.2.1 erwähnt, ist der Codegenerator von Rhapsody aufgrund fehlender Notationselemente nicht in der Lage, direkt einen ausführbaren Code für eine Zielplattform zu generieren. Diese fehlenden Notationselemente werden durch das RXF, welches aus einer Bibliothek mit C Funktionen aufgebaut ist, bereitgestellt. Das RXF ist in verschiedenen Adaptern gekapselt, wodurch das UML Modell weitestgehend von der Hardware Plattform entkoppelt wird. (vgl. [5])
Wird das Embedded UML RXF in Verbindung mit dem OO RTX von Willert verwendet, so kann auf den OSAL Adapter verzichtet werden und es entsteht kein unnötiger Overhead. (vgl. [9])

2.2.3 Keil µVision

Keil µVision ist, wie in Abb. 5 zu sehen, ein Teil des MDK-ARM Micro Controller Development Kit. Es ist eine Embedded IDE und bietet die Möglichkeiten des Projektmanagements, der Quellcodebearbeitung, des Programm Debugging und der Simulation. (vgl. [7])

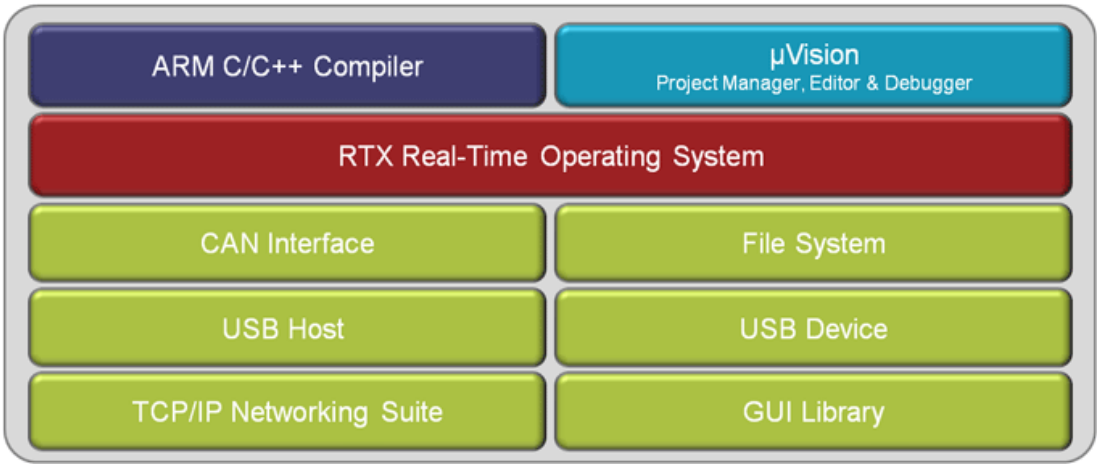


Abb. 5 MDK-ARM Micro Controller Development Kit [7]

Das MDK bietet eine komplette Softwareentwicklungsumgebung für Cortex–M3 Prozessoren basierende Chips wie den NXP LPC1768 auf dem MCB1700.

Ebenfalls werden durch das MDK die entsprechenden ARM C/C++ Compiler bereitgestellt.

Durch den Einsatz von MDK Professional kann auf das Echtzeitbetriebssystem Keil RXT und die RL-ARM zugegriffen werden. In dieser Arbeit wird sowohl eine Evaluationsversion als auch eine Vollversion des MDK verwendet.

Dabei ist zu beachten, dass Keil in der µVision Version 5 das Keil RTX als CMSIS gekapselt hat und diese Schnittstelle noch nicht alle erforderlichen Interfaces für das Willert RXF bereitstellt. Dieses Problem kann jedoch dadurch behoben werden, indem das Keil RTX Legacy Package nachinstalliert wird. (vgl. [10])

Da nun alle verwendeten Komponenten vorgestellt sind, wird im nächsten Kapitel auf die Funktion und Verwendung des Aktivitätsdiagramms genauer eingegangen.

3 Das Aktivitätsdiagramm

In diesem Kapitel wird zunächst auf die allgemeine Verwendung von Aktivitätsdiagrammen eingegangen. Anschließend wird das Konzept der Tokens, welches den Aktivitätsdiagrammen hinterlegt ist, näher erläutert. Schließlich werden alle Notationselemente aufgeführt und die am häufigsten verwendeten näher vorgestellt.

3.1 Allgemeine Verwendung

Aktivitätsdiagramme ermöglichen die Visualisierung von Abläufen, welche zu unterschiedlichen Projektzeitpunkten mit stark variierendem Detailierungsgrad modelliert werden. Dadurch können sie vielfach im Projekt eingesetzt werden, wie beispielsweise zur Geschäftsprozessmodellierung, Use-Case (Anwendungsfall) Beschreibung und Implementierung einer Operation.

3.1.1 Geschäftsprozessmodellierung

Mit UML können neben der Beschreibung von Soft- und Hardwaresystemen auch Geschäftsprozesse modelliert werden. Dies kann sehr gut durch Aktivitätsdiagramme realisiert werden. Da sich diese sehr klar grafisch repräsentieren lassen, werden sie auch gerne zur Optimierung der Prozesse sowie zur Diskussion und Zuordnung von Verantwortlichkeiten benutzt. (vgl. [1] S.273)

3.1.2 Beschreibung von Use-Cases

Wenn Nebenläufigkeiten und Kontrollanweisungen (Schleifen, Entscheidungen, etc.) durch einen Text mit natürlicher Sprache erstellt werden, ist dies zeitaufwändig und nur durch Einrückungen und Gliederungspunkte überhaupt lesbar und verständlich. Ebenso sind nachträgliche Erweiterungen meist nur durch Überarbeitung des gesamten Textes möglich. Diese Nachteile können durch Aktivitätsdiagramme größtenteils eliminiert werden, sodass die Les- und Wartbarkeit deutlich zunehmen. Da Use-Cases ein Ablauf hinterlegt ist, lässt sich dieser durch Aktivitäten beschreiben. (vgl. [1] S.274)

3.1.3 Implementierung einer Operation

Die Realisierung von Algorithmen kann nicht nur in natürlicher Sprache oder in konkreter Programmiersprache dargestellt, sondern in der UML auch durch Aktivitäten modelliert werden. Dadurch wird die Funktionalität einer Operation übersichtlich dargestellt. Dies wird häufig verwendet, wenn aus kompletten UML-Diagrammen Quellcode erzeugt werden soll. (vgl. [1] S.275)

3.2 Das Token-Konzept

Da in Aktivitätsdiagrammen auch komplexe Verhältnisse von nebenläufigen Abläufen erklärt werden müssen, wird dafür das logische Konzept der Tokens verwendet. Ein Token (auch Marke genannt) zeigt den logischen Punkt an, an dem sich ein Ablauf in einem Aktivitätsdiagramm gerade befindet. Es wandert so durch eine Aktivität und repräsentiert damit die Abarbeitung eines Ablaufs. Da in einer Aktivität beliebig viele Tokens unterwegs sein können, kann ein paralleler Ablauf abgebildet werden. (vgl. [1] S.267)
Zur Erklärung sind in der nachfolgenden Abbildung die Tokens durch einen Punkt dargestellt. Dies ist jedoch nicht UML-konform, da Marken in Aktivitätsdiagrammen nicht grafisch repräsentiert werden, sondern nur zur logischen Erklärung dienen.

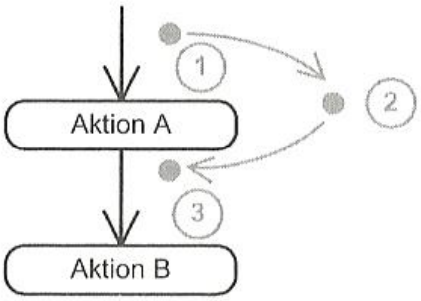


Abb. 6 Token Konzept ([1] S.268)

Eine Aktion wird gestartet, sobald an der eingehenden Kante der Aktion ein Token angeboten und aufgenommen wird (1). Diese Marke wird dann von der Kante weggenommen und verweilt für die Dauer des Ablaufs in der Aktion (2). Wenn der Ablauf abgeschlossen ist, wird das Token über die wegführende Kante der nächsten Aktion angeboten (3).

Tokens verweilen nur in Aktionen und Objektknoten. Da Kanten und Kontrollelemente zeitlos sind, können Marken in ihnen nicht verweilen.

3.3 Notationselemente

Im Bereich der Notationselemente gibt es bei UML 2 im Vergleich zu älteren UML-Versionen viele Neuerungen. Einerseits wurden Elemente aus den vorherigen Versionen zum Teil in ihrer Notationsweise oder in ihrem Inhalt abgewandelt, andererseits sind gänzlich neue hinzugekommen. Diese sind im Wesentlichen:

Tabelle 1 Notationselemente

Aktion	Verzweigungsknoten	Exception-Handler
Aktivität	Verbindungsknoten	Aktivitätsbereich
Objektknoten	Synchronisationsknoten	Strukturierte Knoten
Kanten	Parallelisierungsknoten	Mengenverarbeitungsb.
Startknoten	Parametersatz	Schleifenknoten
Endknoten	Unterbrechungsbereich	Entscheidungsknoten

In den nächsten Abschnitten werden die wichtigsten und die meistgenutzten Elemente genauer beschrieben.

3.3.1 Aktion

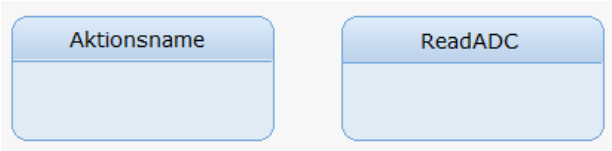


Abb. 7 Aktion

Eine Aktion ist für den Aufruf eines Verhaltens oder die Bearbeitung von Daten verantwortlich. Mit Aktionen werden Einzelschritte beschrieben, die zur Realisierung des durch die Aktivität beschriebenen Verhaltens beitragen. Die Summe aller Aktionen, einschließlich der Reihenfolge ihrer Ausführung und der zur Laufzeit erstellten und verwendeten Daten, realisiert die Aktivität.

Eine Aktion ist das zentrale Element eines Aktivitätsdiagramms. Mit Hilfe aller anderen Elemente werden der Ablauf und die Kontrolle der aufeinander folgenden Aktionen gelenkt und deren Datenaustausch modelliert. (vgl. [1] S.276)
Neben den Standard-Aktionen gibt es auch Sonderformen, die sich mit dem Senden von Signalen und Empfangen von Ereignissen beschäftigen. Diese Sonderformen werden als „SendSignalAction", „AcceptEventAction" und „AcceptTimeEvent" bezeichnet.

3.3.1.1 SendSignalAction (Signalsender)

Abb. 8 Signalsender Aktivitätsdiagramm

Der Signalsender erstellt aus seinen Eingabedaten ein Signal, das an einen Ereignisempfänger gesendet wird. Wenn das Signal gesendet wurde, gilt diese Aktion als beendet und der Kontrollfluss der Aktivität läuft weiter. Im Beispiel hier wird ein Signal „ev_LedOn" versendet, sobald die Aktion ausgeführt wird.

3.3.1.2 AcceptEventAction (Ereignisempfänger)

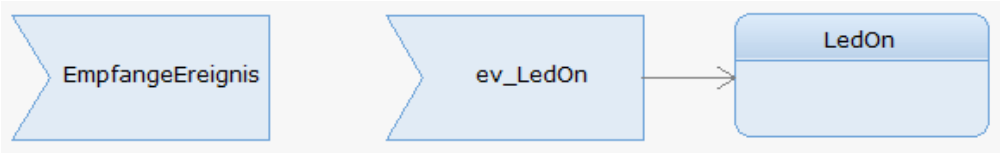


Abb. 9 Ereignisempfänger Aktivitätsdiagramm

Der Ereignisempfänger, das Gegenstück zum Signalsender, wird wie ein „Nutt"-Symbol modelliert. Wenn der Ablauf einen Ereignisempfänger erreicht, verharrt er solange in der Aktion, bis das erwartete Ereignis eintritt. Erst dann wird die Abarbeitung fortgesetzt. Sonderfall hier ist, dass der Ereignisempfänger auch ohne eingehende Kanten modelliert

werden kann. Er wird aktiviert, sobald das Ereignis auftritt. Da die AcceptEventAction immer Ereignisse empfangen kann, sind auch mehrere Ereignisse innerhalb eines Ablaufes einer Aktivität möglich. Im Beispiel wird eine Led eingeschaltet, sobald das Event „ev_LedOn" auftritt.

3.3.1.3 AcceptTimeEvent (Zeitereignis)

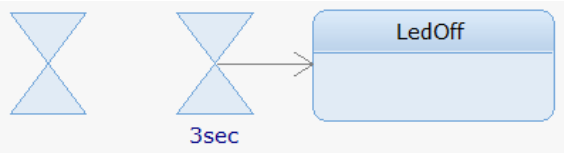


Abb. 10 Zeitereignis Aktivitätsdiagramm

Aktionen können nicht nur durch Ereignisse von anderen Aktionen, sondern auch zeitabhängig gestartet werden. Dies geschieht durch das Symbol, welches eine Eieruhr darstellen soll. Im Beispiel hier wird nach drei Sekunden automatisch eine Led ausgeschaltet.

Folgender Einschub zeigt den Unterschied zwischen Aktion in Aktivitäts- und Zustand in Zustandsdiagrammen bei der Modellierung in Rhapsody auf. Zustandsdiagramme werden im Gegensatz zu Aktivitätsdiagrammen zur ereignisorientierten Modellierung eingesetzt.

3.3.1.4 Vergleich Aktivitätsdiagramm - Zustandsdiagramm

Abb. 11 Zustand

Was im Aktivitätsdiagramm eine Aktion darstellt, kann im Zustandsdiagramm mit einem Zustand (State) verglichen werden. Dieser ist im Wesentlichen gleich aufgebaut, jedoch ist es in Rhapsody möglich, bei einem State einen Code beim Eintritt (Entry) und beim Verlassen (Exit) des Zustandes anzugeben. Bei Aktionen kann nur beim Eintritt in die Aktion ein Code ausgeführt werden.

Ebenso wie im Aktivitätsdiagramm können im Zustandsdiagramm Signale (Events) mit den Elementen „SendSignalAction", „AcceptEventAction" und „AcceptTimeEvent" versendet und empfangen werden. Dies wird jedoch häufig nicht genutzt, da es bei Zustandsdiagrammen im Gegensatz zu Aktivitätsdiagrammen möglich ist, direkt auf der Transition einen Code dafür zu hinterlegen.

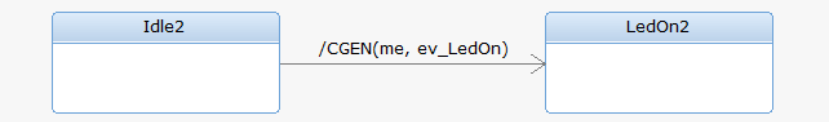


Abb. 12 Signalsender Zustandsdiagramm

So kann im Zustandsdiagramm direkt ein Event (ev_LedOn()) beim Übergang zwischen zwei Zuständen gefeuert werden.

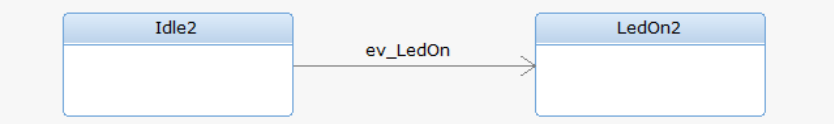


Abb. 13 Ereignisempfänger Zustandsdiagramm

Ebenso kann bei Zustandsdiagrammen das Event, welches als Weiterschaltbedingung erforderlich ist (ev_LedOn), direkt als Trigger in die Transition eingetragen werden.

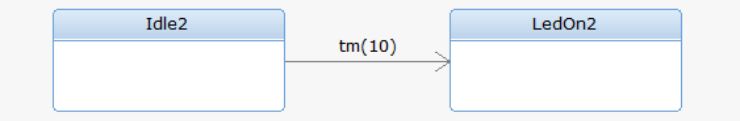


Abb. 14 Zeitereignis Zustandsdiagramm

Ein Zeitereignis wird im Zustandsdiagramm ebenso direkt auf die Transition als Trigger hinterlegt (tm(10)).

Wie hier aufgezeigt, ist ein Zustandsdiagramm für den Einsatz mit Events bestimmt. Da es die Logik von Aktivitätsdiagrammen ist, Modelle zeitkontinuierlich darzustellen, also ohne Events, wirkt sich die

umständliche Modellierung von Events nicht nachteilig auf Aktivitätsdiagramme aus. Der Vergleich zwischen Aktivitäts- und Zustandsdiagrammen ist hiermit abgeschlossen.
Nachfolgend wird die Aktivität, welche einen wichtigen Teil des Aktivitätsdiagramms darstellt, erläutert.

3.3.2 Aktivität

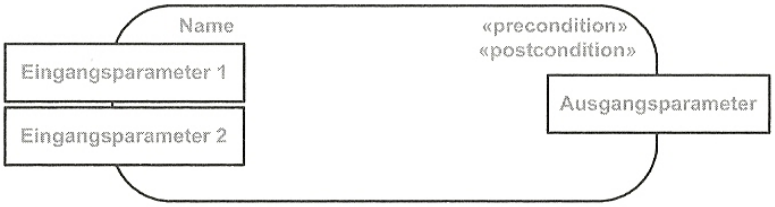


Abb. 15 Aktivität [1] S.281

Eine Aktivität wird durch ein Rechteck mit abgerundeten Ecken dargestellt. Parameter werden in Aktivitäten durch Objektknoten auf den Grenzen des Rechtecks eingezeichnet. Bei dem Begriff Aktivität ist zu beachten, dass dieser zwar aus früheren Versionen von UML übernommen wurde, aber bei UML 2 eine neue Bedeutung bekommen hat. Was früher unter einer Aktivität bekannt war, wird heute als Aktion bezeichnet:

Frühere UML Version: Aktivität **entspricht** UML 2: Aktion

Bei UML 2 wird als Aktivität eine gesamte Einheit, die in einem Aktivitätsmodell modelliert wird, bezeichnet. Dieses besteht aus mehreren Aktionen und weiteren Elementen, wie beispielsweise Kontrollelemente. (vgl. [1] S.281)

Um auch sehr komplexe Systeme darzustellen, ist es möglich, Aktivitäten zu schachteln. Dabei besteht die Möglichkeit, an die Aktivitäten Parameter in Form von Objekten zu übergeben. Dabei legen die Objektknoten an den Rändern die Ein- und Ausgabeparameter für den Beginn und das Ende der Aktivität fest. (vgl. [1] S.281)

3.3.3 Kanten

Abb. 16 Kante ([1] S.291)

Durch Kanten wird der Übergang zwischen zwei Knoten (zum Beispiel Aktionen, Objektknoten etc.) gewährleistet.
Die Kanten sind immer gerichtet und können auch mit Namen versehen werden. Den Begriff Kante gibt es erst seit UML 2. In den früheren Versionen wurden diese Übergänge als Transitionen bezeichnet. Kanten unterteilen sich in Kontrollfluss und Objektfluss. (vgl. [1] S.291)

3.3.3.1 Kontrollfluss

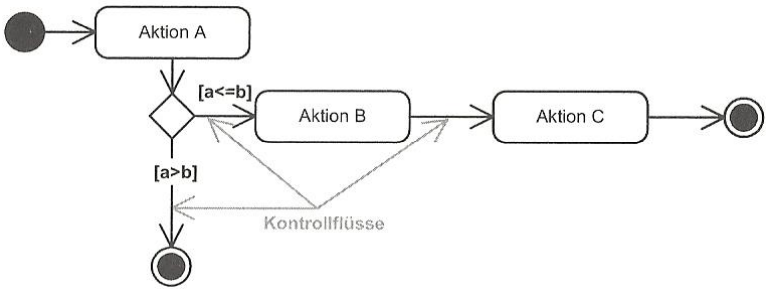


Abb. 17 Kontrollfluss ([1] S.292)

Ein Kontrollfluss stellt eine Kante zwischen zwei Aktionen oder zwischen einer Aktion und einem Kontrollelement dar. Durch Kontrollfluss-Kanten wird die Ausführung einer Aktion erreicht, die verschickten Tokens „tragen" keine Daten. (vgl. [1] S.292)

3.3.3.2 Objektfluss

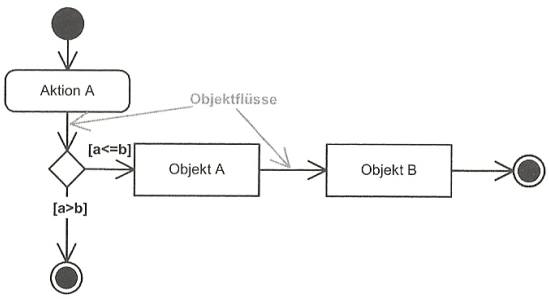


Abb. 18 Objektfluss ([1] S.292)

Bei einer Objektfluss-Kante ist mindestens ein Objektknoten beteiligt. Beim Objektfluss werden über die Kanten Tokens übertragen, welche Daten/Werte zum oder vom Objektknoten transportieren. (vgl. [1] S.292)

3.3.3.3 Bedingungen

Es besteht die Möglichkeit, Kanten mit Bedingungen (Guards) zu belegen. Dabei kann die Kante nur überwunden werden, wenn die Bedingung erfüllt ist. (vgl. [1] S.292)

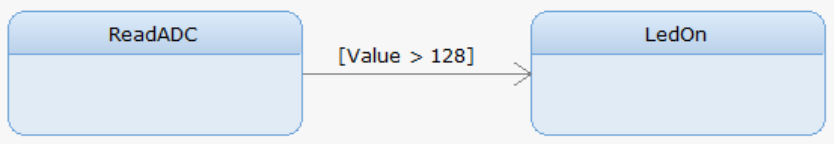


Abb. 19 Kante Bedingung

3.3.3.4 Sprungmarken

Abb. 20 Sprungmarke ([1] S.294)

Müssen Kanten über einen weiten Weg gezogen werden, so ist es aus Gründen der Übersicht und ästhetischer Gestaltung sinnvoll, diese zu unterbrechen und an der entfernten Stelle wieder weiter zu führen. Dafür sind Sprungmarken vorgesehen. Sprungmarken müssen immer paarweise auftreten und mit einem eindeutigen Namen gekennzeichnet sein. (vgl. [1] S.294)
Es folgt ein weiterer Einschub, welcher den Unterschied zwischen Kanten in Aktivitäts- und Transitionen in Zustandsdiagrammen darstellt.

3.3.3.5 Vergleich Aktivitätsdiagramm - Zustandsdiagramm

In Zustandsdiagrammen wird die Verbindung zwischen Zuständen mit Transitionen modelliert. Wie schon in 3.3.1.4 gezeigt, können Transitionen nicht wie in Aktivitätsdiagrammen nur mit einer Bedingung

(Guard), sondern zusätzlich auch mit einem Trigger (Event) und einem auszuführenden Code dargestellt werden.

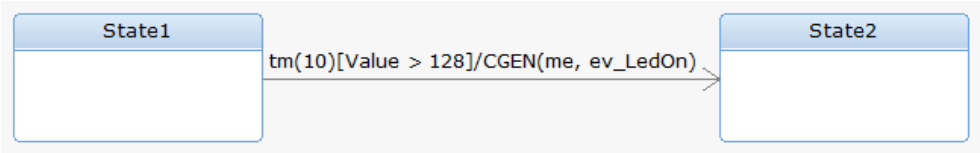


Abb. 21 Transition Zustandsdiagramm

Ein wesentlicher Unterschied zwischen Kante und Transition bei der Modellierung in Rhapsody wurde aufgezeigt; nachfolgend werden Kontrollelemente eines Aktivitätsdiagramms vorgestellt.

3.3.4 Kontrollelemente

Kontrollelemente sind häufig verwendete Elemente in Aktivitätsdiagrammen, mit denen der Weg der Tokens gesteuert wird. In allen Kontrollelementen gilt, dass Tokens in ihnen nicht verweilen dürfen. Muss ein Token beispielsweise an einem Synchronisationsknoten auf ein anderes Token warten, so verbleibt das Token vollständig in der vorhergehenden Aktion und wartet nicht im Kontrollelement. Folgende Kontrollelemente können in Aktivitätsdiagrammen verwendet werden. (vgl. [1] S.296)

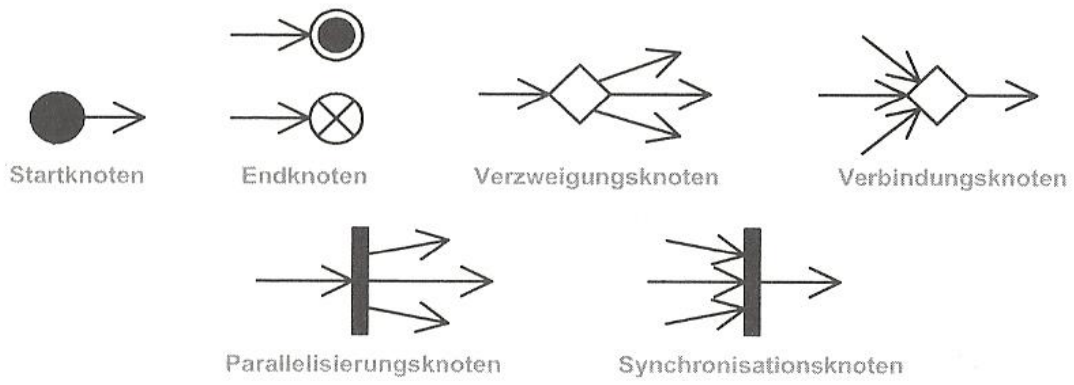


Abb. 22 Kontrollelemente ([1] S.294)

3.3.4.1 Startknoten und Endknoten

Startknoten

Durch den Startknoten (ausgefüllter Kreis) wird der Startpunkt eines Ablaufs bei Aktivierung der Aktivität dargestellt. Eine Aktivität kann beliebig viele Startknoten haben. Jeder Startknoten dagegen kann wiederum beliebig viele wegführende Kanten besitzen. Beim Start einer Aktivität erstellt der Startknoten Tokens (Anzahl der Tokens gleich Anzahl der Kanten).

Endknoten

Durch einen Endknoten für Aktivitäten (Punkt mit einem umschließenden Ring, vgl. Zielscheibe, im Englischen „bull's eye") wird die gesamte Aktivität beendet, sobald er von einem Token erreicht wird. Innerhalb einer Aktivität können mehrere Endknoten existieren. In diesem Fall wird die Aktivität beendet, sobald einem dieser Endknoten ein Token übergeben wird. (vgl. [1] S.299)

3.3.4.2 Verzweigungs- und Verbindungsknoten

Um Kanten zu spalten und anschließend wieder zusammen zu führen werden in der UML der Verzweigungs- und der Verbindungsknoten verwendet. Beide werden durch eine Raute symbolisiert.

Verzweigungsknoten

Durch einen Verzweigungsknoten wird eine Kante in mehrere Alternativen aufgespaltet. Dies wird besonders dann genutzt, wenn ein Ablauf in der Aktivität von bestimmten Bedingungen abhängig ist. Wenn ein Token einen Verzweigungsknoten erreicht, dann passiert dieses nur eine ausgehende Kante, das Token wird also nicht dupliziert. Diese ausgehende Kante wird durch die Bedingungen, welche eindeutig beschrieben sind und sich nicht überschneiden, festgelegt. In der Bedingung wird definiert, welche Anforderungen ein Token (oder die Daten/Werte des Tokens) erfüllen muss, um die Kante zu passieren.

Verbindungsknoten

Der Verbindungsknoten ist das Gegenstück zum Verzweigungsknoten, da bei ihm Kanten zusammengeführt werden. Allerdings findet hier keine Synchronisierung der eingehenden Kanten statt, die Tokens werden also nicht miteinander verschmolzen. Wenn mehrere Tokens parallel an den Kanten anliegen, werden diese serialisiert und an der ausgehenden Kante angeboten. (vgl [1] S.302)

Abb. 23 Verbindungsknoten ODER/UND

Wie im oberen Beispiel dargestellt, ist darauf zu achten, ob die Kanten getrennt oder gemeinsam über einen Verbindungsknoten in die nächste Aktion übergehen. Wird ein Verbindungsknoten verwendet, stellt dies eine „Explizite ODER" Verbindung dar. Dies bedeutet, dass die Aktion „LedOn" ausgeführt wird, wenn entweder ein Token aus der Aktion „Switch1On" oder ein Token aus der Aktion „Switch2On" bereit steht. Wenn jedoch kein Verbindungsknoten verwendet wird, stellt dies ein „Implizites UND" dar; das heißt die Aktion „LedOn" wird nur ausgeführt, wenn Tokens aus den Aktionen „Switch1On" und „Switch2On" bereit stehen.

3.3.4.3 Parallelisierungs- und Synchronisationsknoten

In der UML ist es möglich, Abläufe zu mehreren Flüssen zu parallelisieren oder zu einem Fluss zu synchronisieren. Dazu werden Parallelisierungs- und Synchronisationsknoten verwendet. Beide Knoten werden durch einen Balken dargestellt.

Parallelisierungsknoten

Durch einen Parallelisierungsknoten wird ein Ablauf in mehrere parallele Abläufe aufgeteilt. Das eingehende Token wird dabei dupliziert und an jede ausgehende Kante angeboten. Beide Kanten, sowohl die eingehende als auch die ausgehende, können zusätzlich mit Bedingungen versehen werden.

Synchronisationsknoten

Mit einem Synchronisationsknoten werden die eingehenden Abläufe zu einem gemeinsamen Ablauf zusammengeführt. Um dies zu realisieren, müssen an allen eingehenden Kanten gleichzeitig Tokens (Kontroll- oder Daten-Tokens) angeboten werden. Kontroll-Eingangs-Tokens werden dabei zu einem gemeinsamen Ausgangs-Token verschmolzen. Daten-Tokens, die Objekte mit gleichem Wert oder das gleiche Objekt beinhalten, werden vor Weiterleitung zu einem einzigen Daten-Token zusammengefasst. Zu beachten ist außerdem, wenn am Eingang des Synchronisationsknoten Daten- und Kontroll-Tokens anliegen, werden nur die Daten-Tokens weitergeleitet, die Kontroll-Tokens werden vernichtet.

Der nachfolgende Einschub erklärt die Verwendung von Kontroll-elementen in Aktivitäts- und Zustandsdiagrammen.

3.3.4.4 Vergleich Aktivitätsdiagramm - Zustandsdiagramm

Ebenso wie in Aktivitätsdiagrammen ist es auch in Zustandsdiagrammen möglich, Kontrollelemente einzufügen. Sie dienen für den Start und das Ende eines Zustandsdiagramms, das Verzweigen und Verbinden oder Parallelisieren und Synchronisieren von Transitionen.

Im weiteren Verlauf wird aufgezeigt, wann Aktivitätsdiagramme oder Zustandsdiagramme zur Modellierung eingesetzt werden sollen.

3.4 Verwendung Aktivitäts- / Zustandsdiagramm

Aktivitätsdiagramme werden zur Modellierung von Abläufen verwendet. Mit Aktivitätsdiagrammen können komplexe Verläufe unter Berücksichtigung von Nebenläufigkeiten, alternativen Entscheidungswegen und ähnlichem modelliert und nachvollzogen werden. Anhand eines Aktivitätsdiagramms wird die Frage beantwortet:

> „Wie realisiert mein System ein bestimmtes Verhalten?"([1] S.265)

Im Gegensatz dazu wird das Zustandsdiagramm zum Modellieren von Zuständen, welche ein System einnehmen kann, benutzt. Die Übergänge zwischen Zuständen werden durch interne oder externe Ereignisse ausgelöst. Mit einem Zustandsdiagramm wird somit die Frage beantwortet:

> „Wie verhält sich das System in einem bestimmten Zustand bei gewissen Ereignissen?"([1] S.335)

Um den Verwendungszweck von Aktivitätsdiagrammen und deren Vorteile zu verdeutlichen, wird das folgende Beispiel angeführt.

Das Beispiel weist folgende Funktion auf. Es gibt drei Tasten (mode, plus, minus), mit denen die Helligkeit einer Led und eines Glcd Displays verstellt werden kann. Mit der Modus-Taste kann zwischen der Bedienung von Helligkeit der Led und Helligkeit des Glcds umgeschaltet werden. Mit der Plus- und Minus-Taste kann die Helligkeit entsprechend eingestellt werden.

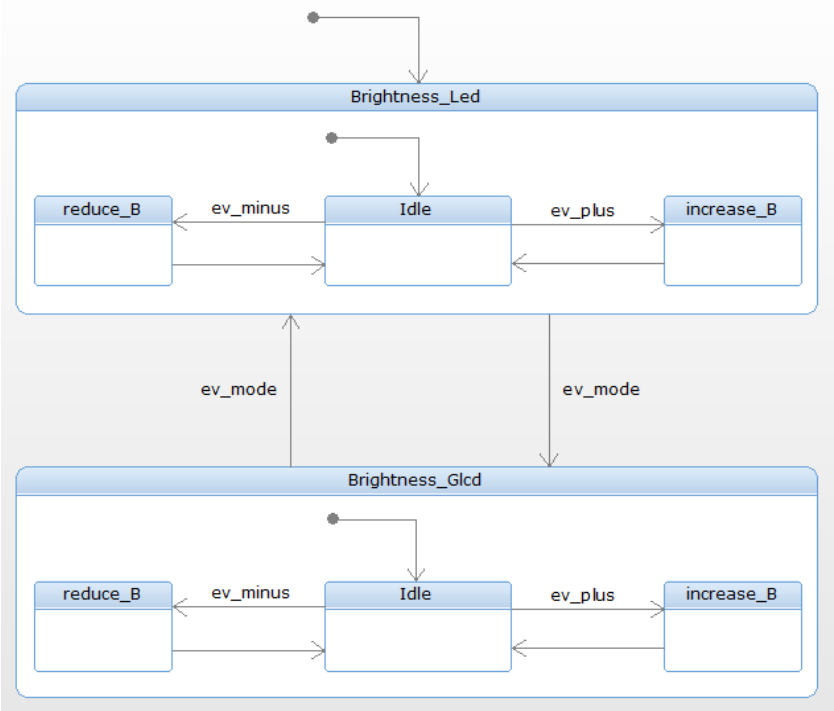


Abb. 24 Zustandsübergänge Ereignisse

Das in Abb. 24 gezeigte Zustandsdiagramm stellt die zuvor beschriebene Funktion mittels Ereignissen (Events) dar. Entsprechend den Tasten werden Ereignisse in Form von Events „ev_mode", „ev_plus" und „ev_minus" erzeugt.

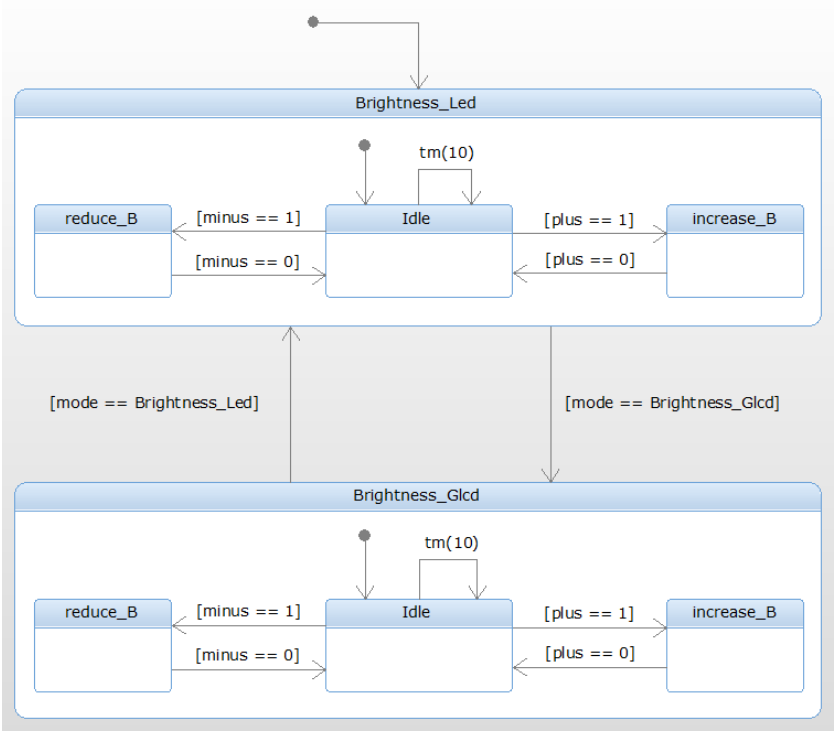


Abb. 25 Zustandsübergänge Guards

Das Zustandsdiagramm in Abb. 25 zeigt ebenfalls die genannte Funktion auf, allerdings sind die Übergänge mittels Guards realisiert. Entsprechend den Tasten werden die Variablen „mode", „plus" und „minus" gesetzt.

Oberflächlich betrachtet weisen die Zustandsdiagramme in Abb. 24 und Abb. 25 dasselbe Verhalten auf.

Nun soll allerdings ein besonderer Fall betrachtet werden und es wird angenommen, dass man sich im Modus „Brightness_Led" befindet und ein Bediener die Helligkeit des Glcds erhöhen möchte. Er wird also zuerst die Modus-Taste und anschließend die Plus-Taste drücken. Da die CPU des MCB1700 momentan sehr beschäftigt ist und die Tasten sehr schnell hintereinander gedrückt werden, sind beide Tasten vor dem nächsten Aufruf des Zustandsdiagrammes durch die CPU betätigt worden.

Beim Zustandsdiagramm in Abb. 24 werden die Ereignisse (Events) im Hintergrund in einer sogenannten „Queue" nach ihrer aufgetretenen Reihenfolge abgelegt und abgearbeitet. Dadurch ist ein eindeutiger Ablauf gewährleistet.
In dem in Abb. 25 gezeigten Zustandsdiagramm werden kontinuierliche Signale (aus zeitlicher Sicht kontinuierlich gültige Werte z.B. in Form von Variablen) als Übergangsbedingung verwendet. Hier gibt es keine Information, in welcher Reihenfolge die Variablen geändert wurden. Wird nun das Zustandsdiagramm durch die CPU ausgeführt, so wurden durch den schnellen Tastendruck zwei Variablen geändert und es sind zwei Übergangsbedingungen erfüllt. Dadurch kann es vorkommen, dass entweder zuerst der Modus gewechselt und dann die Helligkeit des Glcds erhöht wird oder zuerst die Helligkeit der Led erhöht und dann erst der Modus gewechselt wird. Somit ist kein eindeutiger Funktionsablauf garantiert.

Um einen eindeutigen Funktionsablauf mit kontinuierlichen Signalen zu gewährleisten, ist es ratsam, ein Aktivitätsdiagramm zu verwenden, welches den Ablauf eindeutig wiedergibt.

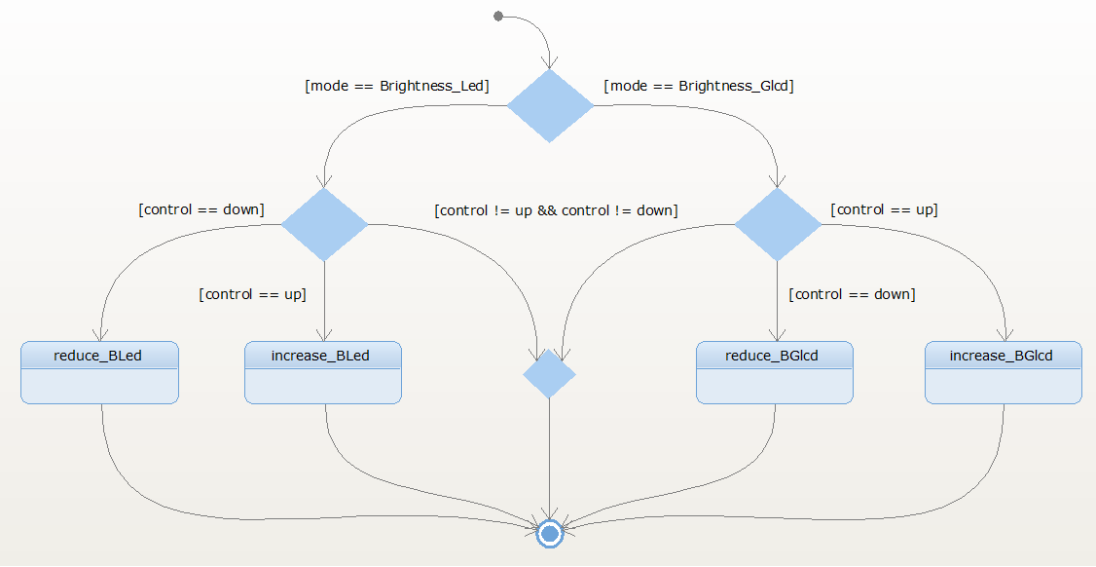


Abb. 26 Aktivitätsdiagramm kontinuierliche Signale

Durch die ablauforientierte Modellierung im Aktivitätsdiagramm ist auch bei gleichzeitiger Veränderung zweier Variablen ein eindeutiger Ablauf, wie in Abb. 26 zu sehen, vorgegeben. (vgl. [14])

Folgendes Kapitel zeigt die Vorgehensweise der Modellierung von Aktivitätsdiagrammen in Rhapsody und den daraus automatisch generierten Code anhand von Beispielen auf.

4 Modellierung Aktivitätsdiagramme in Rhapsody

In diesem Kapitel wird die Modellierung eines Aktivitätsdiagrammes beschrieben. Zu Beginn wird die für dieses Projekt verwendete Software und das verwendete Echtzeitsystem vorgestellt. Danach wird beschrieben, wie Aktivitätsdiagramme in Rhapsody modelliert werden, um diese schließlich mit der Willert-Software zu übersetzen und daraus einen Code zu erzeugen. Zunächst wird der generierte Code aus den wesentlichen Elementen eines Aktivitätsdiagramms dargestellt und erklärt, um anschließend diese Elemente in einem Beispiel zu verwenden. Die dort aufgetretenen Probleme werden erklärt und schließlich eine verbesserte Version des Beispiels gezeigt, die ohne Fehler auf dem Board ausgeführt werden kann. Abschließend wird ein Fazit aus der Verwendung von Aktivitätsdiagrammen in Rhapsody mit der Willert Software gezogen.

4.1 Verwendete Software und Echtzeitsystem

Zur Modellierung der Aktivitätsdiagramme wird das Softwarepackage Rpy_C_OO RTX_Keil_ARM_MCB1700_Eval_TD_V6.01 von Willert verwendet. Dieses beinhaltet Rhapsody 8.0.1 als UML-Modellierungswerkzeug, das Embedded UML RXF von Willert, Keil µVision 4 als IDE und das Echtzeitbetriebssystem Willert OO RTX, wie folgende Tabelle verdeutlicht.

Tabelle 2 Software und Echtzeitsystem

RXF	Willert OO RTX
IDE	Keil µVision 4
Modellierung	IBM Rational Rhapsody
Sprache	ANSI C
RTOS	Willert OO RTX
Compiler und Zielplattform	Keil MDK-ARM
Board	Keil MCB 1700

Zu Beginn der Arbeiten wurde Keil µVision 4 als IDE verwendet, jedoch später durch Keil µVision 5 mit dem Keil RTX Legacy Package ersetzt, wodurch Keil µVision 5 die gleiche Funktionalität wie Keil µVision 4 bietet.

4.1.1 Beschränkung UML Modellierung durch Willert RXF

Das RXF von Willert unterstützt die Codegenerierung von Rhapsody, indem ein Code für fehlende Notationselemente, wie beispielsweise Timer, in Form einer Bibliothek zur Verfügung gestellt wird. Jedoch gibt es einige Funktionen in Rhapsody, welche das RXF nicht unterstützt. Diese sind im Folgenden aufgelistet.

Aktivitätsdiagramme in Verbindung mit Operationen

Es wird nur aus Aktivitätsdiagrammen und Zustandsdiagrammen, welche mit einer Klasse oder einem Objekt verbunden sind, ein C Code generiert. Zwar können Aktivitätsdiagrammen Operationen zugewiesen werden, aber daraus wird kein Code generiert.

TCP/IP

Aufgrund des großen Overheads unterstützt das Embedded RXF keine Kommunikation via TCP/IP.

Vererbung

Für Vererbung in C müssten alle Funktionen als Callbacks oder als virtuelle Funktionstabelle implementiert sein. Das würde die Codegröße sehr stark anwachsen lassen und deswegen ist in Rhapsody C keine Vererbung möglich.

Aktive Klassen (Tasks)

Aufgrund des RTOS OO RTX können keine aktiven Klassen angelegt werden. Wie in Abb. 27 zu sehen ist, kann im Eigenschaftenfenster einer Klasse die Eigenschaft „Concurrency" von „Sequential" auf „Active" umgestellt werden. Dies würde bedeuten, dass für die Klasse ein eigener Task angelegt wird. Eine genauere Beschreibung folgt im nächsten Abschnitt. (vgl. [8])

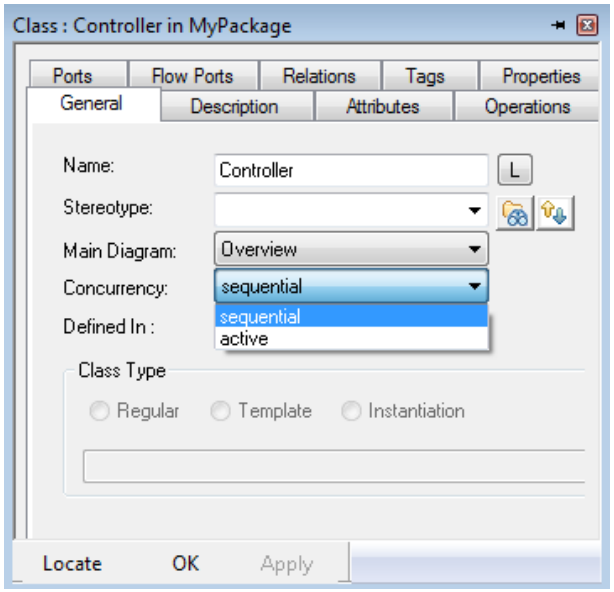


Abb. 27 Klasse Sequential - Active

4.1.2 OO RTX

Das OO RTX von Willert ist ein Echtzeitbetriebssystem, das seine Anwendung im Einsatz mit UML-modellierten Systemen findet.
Es unterstützt alle notwendigen Betriebssystemdienste wie Timer, Events, Messages und Tasks und der Scheduler arbeitet nach dem Motto:

> *„Wer nicht kommt zur rechten Zeit, der muss sehen was übrig bleibt*" [9]

Das OO RTX basiert auf einer Single Task Architektur. Dies bedeutet, dass präemptives Scheduling und reales Umschalten von Tasks nicht möglich ist und dadurch keine aktiven Klassen (Tasks) modelliert werden können. Trotzdem besteht die Möglichkeit durch Zustandsdiagramme und Events, welche in einer Event Queue verwaltet werden, ein Scheduling zu modellieren.

Das OO RTX unterstützt keine Semaphores und Mutexes. Timeouts können mit Hilfe einer Timeout Transition z. B. tm(Zeit in Millisekunden) modelliert werden. Aufgrund der eventgetriebenen Software-Architektur ergeben sich Probleme wie Deadlocks nicht. (vgl. [8])

4.2 Aktivitätsdiagramme in Rhapsody mit Willert RXF

Aktivitätsdiagramme in Rhapsody sind nicht tokenbasiert, sondern zustandsorientiert. Dies bedeutet, dass jede Aktion in Aktivitätsdiagrammen wie ein Zustand im Zustandsdiagramm behandelt wird. Dieses Verhalten von Rhapsody wird durch einen Vergleich des Codes aus einem Aktivitätsdiagramm und Zustandsdiagramm in Abschnitt 4.2.1 aufgezeigt.
Die wichtigsten und am häufigsten verwendeten Elemente, welche auch in einem späteren Beispiel Verwendung finden, werden nun betrachtet und der daraus generierte Code genauer beleuchtet.

4.2.1 Aktion und Kante

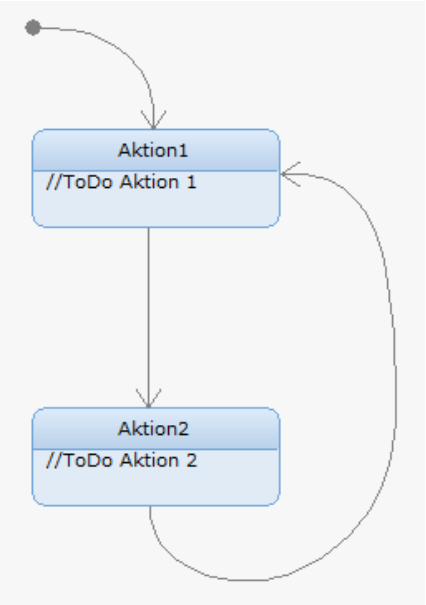


Abb. 28 Aktionen mit Kanten

Aus dem UML Modell wird folgender Code generiert.

```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           WST_FSM_pushNullConfig(&(me->ric_reactive));
6           me->rootState_subState = LED_Aktion1;
7           me->rootState_active = LED_Aktion1;
8           me->ric_reactive.currentState = me->rootState_active;
9           {
10              /*#[ state Aktion1.(Entry) */
11              //ToDo Aktion 1;
12              /*#]*/
13          }
14      }
15  }
16
17  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
18
19      LED * const me = (LED *)void_me;
20      RiCTakeEventStatus res = eventNotConsumed;
21      switch (me->rootState_active) {
22          /* State Aktion1 */
23          case LED_Aktion1:
24          {
25              /*## transition 1 */
26              if(id == Null_id)
27                  {
28                      WST_FSM_popNullConfig(&(me->ric_reactive));
29                      WST_FSM_pushNullConfig(&(me->ric_reactive));
30                      me->rootState_subState = LED_Aktion2;
31                      me->rootState_active = LED_Aktion2;
32                      me->ric_reactive.currentState = me->rootState_active;
33                      {
34                          /*#[ state Aktion2.(Entry) */
35                          //ToDo Aktion 2;
36                          /*#]*/
37                      }
38                      res = eventConsumed;
39                  }
40          }
41          break;
42          /* State Aktion2 */
43          case LED_Aktion2:
44          {
45              /*## transition 2 */
46              if(id == Null_id)
47                  {
48                      WST_FSM_popNullConfig(&(me->ric_reactive));
49                      WST_FSM_pushNullConfig(&(me->ric_reactive));
50                      me->rootState_subState = LED_Aktion1;
51                      me->rootState_active = LED_Aktion1;
52                      me->ric_reactive.currentState = me->rootState_active;
53                      {
54                          /*#[ state Aktion1.(Entry) */
55                          //ToDo Aktion 1;
56                          /*#]*/
57                      }
58                      res = eventConsumed;
59                  }
60          }
61          break;
62          default:
63              break;
64      }
65      return res;
66  }
```

In Abb. 28 ist ein Teil eines einfachen Aktivitätsdiagramms mit zwei Aktionen dargestellt. Durch das Verbinden von Aktion1 mit der Initialisierungskante (Default Transition) wird Aktion1 als root State gesetzt. Aktion1 ist mit Aktion2 über eine Kante ohne Bedingung verbunden.

In den Codezeilen 1-15 ist die Codegenerierung des root States ersichtlich. Dieser Programmabschnitt wird bei Abarbeitung des Aktivitätsdiagramms zuerst ausgeführt. In der Funktion „rootState_entDef" wird die Aktion1 als aktive Aktion übernommen (Zeile 6f). In Zeile 11 ist dargestellt, welcher Code beim Eintritt (Entry) in die Aktion ausgeführt wird. Bei Zuständen in Zustandsdiagrammen besteht die Möglichkeit, einen Code sowohl beim Eintritt in den Zustand, als auch beim Verlassen des Zustandes auszuführen. Im Gegensatz dazu kann in Aktivitätsdiagrammen bei einer Aktion nur beim Eintritt eine Aktion eingegeben werden.

Nach dem Aufruf der „rootState_entDef" Funktion wird nun die Funktion „rootState_dispatchEvent" aufgerufen und dort das generierte Aktivitätsdiagramm ausgeführt. Anhand einer Switch-Case Anweisung (Zeile 21), welche den aktuellen „rootState_active" überprüft, wird die aktuell aktive Aktion ermittelt. Wie in Zeile 22 zu sehen ist, wird Aktion1 als Zustand (State Aktion1) übersetzt. Da sich der Ablauf des Aktivitätsdiagramms zu Beginn im root State (Aktion1) befindet, wird der erste Case (Zeile 23) erfüllt. Anschließend wird in Zeile 26 überprüft, welches Event zum Aufruf der „rootState_dispatchEvent" Funktion geführt hat (Eventgetriebene Softwarearchitektur). Dieser Funktion werden als Parameter ein void-Pointer, welcher auf die Anfangsadresse des Objekts „itsLED" zeigt, sowie die ID des aufrufenden Events übergeben. Da Kanten nur mit Bedingungen (Guard), aber nicht mit Events (Trigger) modelliert werden können, ist die übergebene ID des aufrufenden Events immer eine „Null_id". Dadurch wird die Bedingung in Zeile 26 erfüllt, in die Aktion2 gewechselt und der Code beim Eintritt in die Aktion (Zeile 35) ausgeführt. Abschließend wird als Ergebnis „eventConsumend" zurückgegeben. Schließlich wird beim nächsten Aufruf

der „rootState_dispatchEvent" Funktion der Case „LED_Aktion2" erfüllt und nach gleichem Ablauf wie in Aktion1 beschrieben, vorgegangen.
Um nun die Codegeneration aus Aktivitäts- und Zustandsdiagrammen vergleichen zu können, erfolgt folgender Einschub.

Vergleich Aktivitäts-/Zustandsdiagramm

Wie bereits in Abschnitt 4.2 erwähnt, ist das Aktivitätsdiagramm in Rhapsody zustandsorientiert. Um dies zu verdeutlichen, wurde folgendes Zustandsdiagramm modelliert und der Code betrachtet. Das Zustandsdiagramm in folgender Abbildung hat die gleiche Funktionalität wie das Aktivitätsdiagramm in Abb. 28.

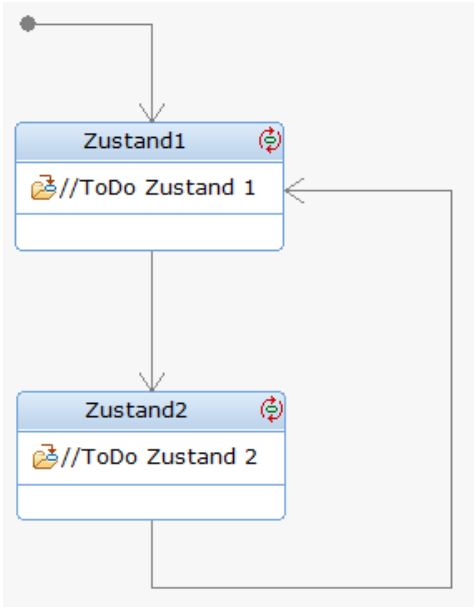


Abb. 29 Zustände/Transitionen Zustandsdiagramm

Aus dem UML Modell in Abb. 29 wird folgender Code generiert.

```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           WST_FSM_pushNullConfig(&(me->ric_reactive));
6           me->rootState_subState = LED_Zustand1;
7           me->rootState_active = LED_Zustand1;
8           me->ric_reactive.currentState = me->rootState_active;
9           {
10              /*#[ state Zustand1.(Entry) */
11              //ToDo Zustand 1;
12              /*#]*/
13          }
14      }
15  }
16
```

```
17  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
18
19      LED * const me = (LED *)void_me;
20      RiCTakeEventStatus res = eventNotConsumed;
21      switch (me->rootState_active) {
22          /* State Zustand1 */
23          case LED_Zustand1:
24          {
25              /*## transition 1 */
26              if(id == Null_id)
27                  {
28                      WST_FSM_popNullConfig(&(me->ric_reactive));
29                      WST_FSM_pushNullConfig(&(me->ric_reactive));
30                      me->rootState_subState = LED_Zustand2;
31                      me->rootState_active = LED_Zustand2;
32                      me->ric_reactive.currentState = me->rootState_active;
33                      {
34                          /*#[ state Zustand2.(Entry) */
35                          //ToDo Zustand 2;
36                          /*#]*/
37                      }
38                      res = eventConsumed;
39                  }
40          }
41          break;
42          /* State Zustand2 */
43          case LED_Zustand2:
44          {
45              /*## transition 2 */
46              if(id == Null_id)
47                  {
48                      WST_FSM_popNullConfig(&(me->ric_reactive));
49                      WST_FSM_pushNullConfig(&(me->ric_reactive));
50                      me->rootState_subState = LED_Zustand1;
51                      me->rootState_active = LED_Zustand1;
52                      me->ric_reactive.currentState = me->rootState_active;
53                      {
54                          /*#[ state Zustand1.(Entry) */
55                          //ToDo Zustand 1;
56                          /*#]*/
57                      }
58                      res = eventConsumed;
59                  }
60          }
61          break;
62          default:
63              break;
64      }
65      return res;
66  }
```

Wird der generierte Code aus dem Aktivitätsdiagramm und Zustandsdiagramm verglichen, so stellt man fest, dass dieser funktional vollkommen identisch ist.

Ebenso ist der generierte Code aus den Diagrammen, welche in Abb. 30, Abb. 31 und Abb. 32 dargestellt sind, funktional gleich.

Signalsender

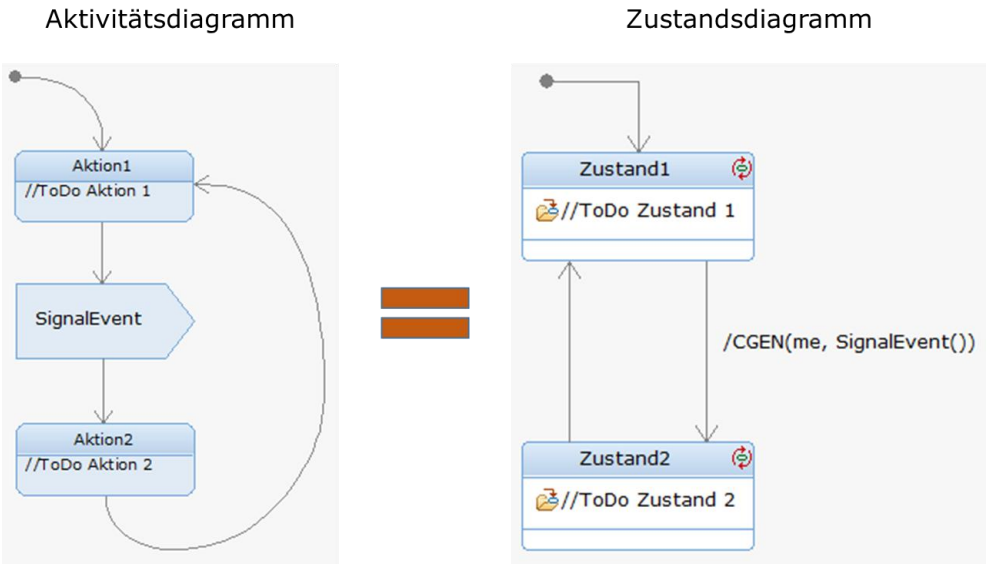


Abb. 30 Vergleich Signalsender

Ereignisempfänger

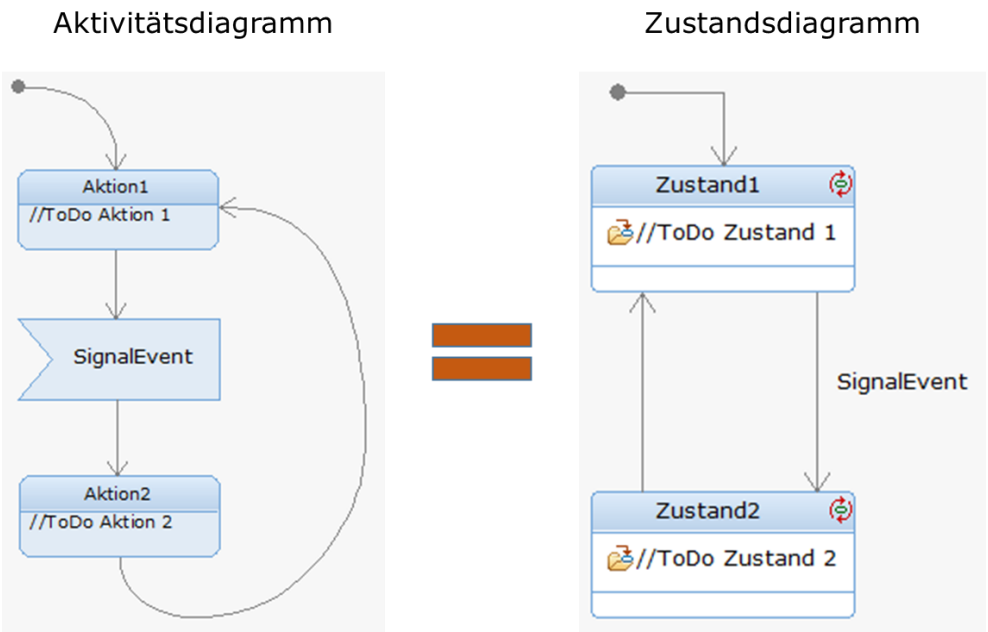


Abb. 31 Vergleich Ereignisempfänger

Zeitereignis

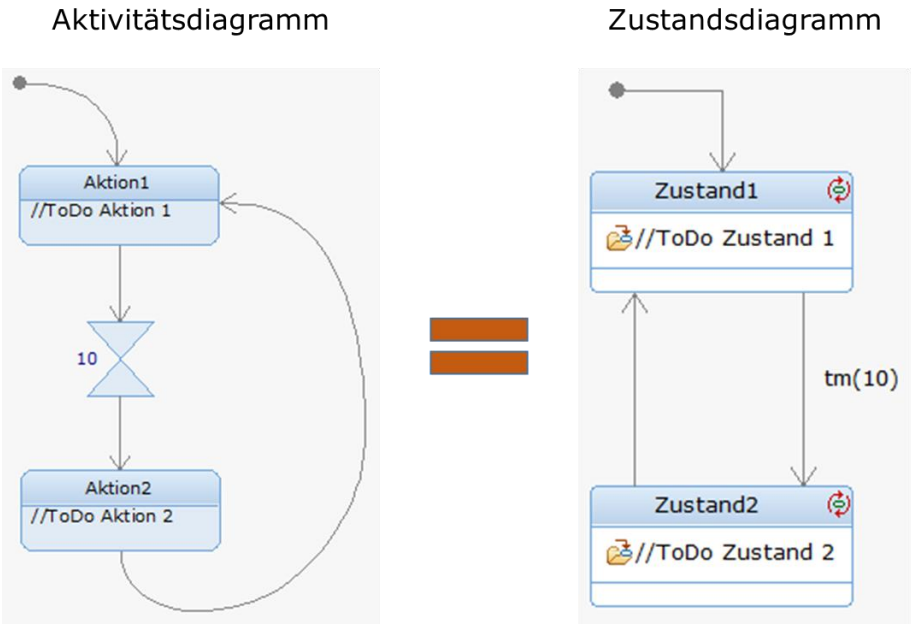


Abb. 32 Vergleich Zeitereignis

4.2.2 Zeitereignis

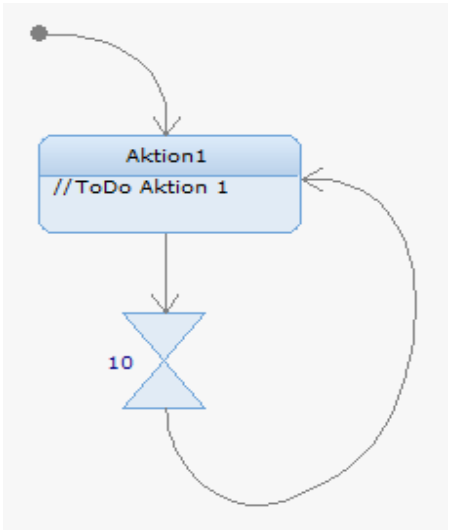


Abb. 33 Zeitereignis

Aus dem UML-Modell in Abb. 33 wird folgender Code generiert.

```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           me->rootState_subState = LED_Aktion1;
6           me->rootState_active = LED_Aktion1;
7    me->ric_reactive.currentState = me->rootState_active;
8           {
9               /*#[ state Aktion1.(Entry) */
10              //ToDo Aktion 1;
11              /*#]*/
12          }
13          WST_DUMMY_TASK_schedTm(me->ric_reactive.myTask, 10, LED_Timeout_Aktion1_id,
14  &(me->ric_reactive), NULL);
15      }
16  }
17
18  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
19
20      LED * const me = (LED *)void_me;
21      RiCTakeEventStatus res = eventNotConsumed;
22      switch (me->rootState_active) {
23          /* State Aktion1 */
24          case LED_Aktion1:
25          {
26              /*## transition 1 */
27              if(id == Timeout_id)
28                  {
29                      if(RiCTimeout_getTimeoutId((RiCTimeout*) me->
30                  ric_reactive.current_event) == LED_Timeout_Aktion1_id)
31                          {
32                              WST_DUMMY_TASK_unschedTm(me->
33          ric_reactive.myTask, LED_Timeout_Aktion1_id, &(me->ric_reactive));
34                              WST_FSM_pushNullConfig(&(me->ric_reactive));
35                              me->rootState_subState = LED_Zeitereignis;
36                              me->rootState_active = LED_Zeitereignis;
37                              me->ric_reactive.currentState = me->
38                                rootState_active;
39                              res = eventConsumed;
40                          }
41                  }
42          }
43          break;
44          /* State Zeitereignis */
45          case LED_Zeitereignis:
46          {
47              /*## transition 2 */
48              if(id == Null_id)
49                  {
50                      WST_FSM_popNullConfig(&(me->ric_reactive));
51                      me->rootState_subState = LED_Aktion1;
52                      me->rootState_active = LED_Aktion1;
53                         me->ric_reactive.currentState = me->rootState_active;
54                      {
55                          /*#[ state Aktion1.(Entry) */
56                          //ToDo Aktion 1;
57                          /*#]*/
58                      }
59                      WST_DUMMY_TASK_schedTm(me->ric_reactive.myTask, 10,
60                                  LED_Timeout_Aktion1_id, &(me->ric_reactive),
61                                          NULL);
62                      res = eventConsumed;
63                  }
64          }
65          break;
66          default:
67              break;
68      }
69      return res;
70  }
```

Das Zeitereignis stellt eine Sonderform der Aktion dar. In Abb. 33 ist ein Aktivitätsdiagramm mit einer Aktion und einem Zeitereignis zu sehen. Diese sind durch zwei Kanten in einem Kreislauf miteinander verbunden. Durch ein Zeitereignis können Verzögerungen (hier 10 ms) modelliert werden.

Auf die genaue Beschreibung der „rootState_entDef" Funktion und das Ermitteln der aktuell aktiven Aktion wird an dieser Stelle verzichtet, da dies bereits in Abschnitt 4.2.1 bei Aktionen und Kanten ausführlich dargestellt wurde. Hier wird gesondert auf die Codezeilen 13 und 14 eingegangen, welche durch das Verwenden eines Zeitereignisses entstehen. Aufgrund dieser Codezeilen wird nach der angegebenen Zeit, welche durch das Zeitereignis definiert wird (hier 10 ms), ein Event mit der ID „LED_Timeout_Aktion1_id" generiert.

Allgemein wird nach Ablauf der Zeit ein Event mit der ID „Timeout_id" generiert. Diese Event-ID wird als Weiterschaltbedingung (Transition 1 Zeile 27) geprüft. In Zeile 29f wird schließlich überprüft, ob das Event mit der ID „LED_Timeout_Aktion1_id" vorliegt. Ist dies der Fall, so wird das Zeitereignis, welches als eigene Aktion (Zeile 45) generiert ist, aktiv geschaltet (Zeile 35f). In der Aktion Zeitereignis wird zunächst die Aktion1 aktiv geschaltet (Zeile 51f) und anschließend wieder die Generierung eines Timeout Events angestoßen (Zeile 59f), wodurch der Ablauf von neuem beginnt.

4.2.3 Verzweigungs- und Verbindungsknoten

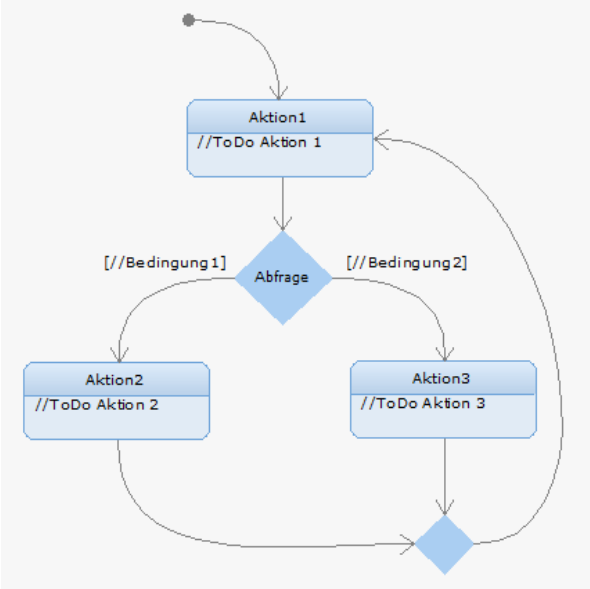


Abb. 34 Verzweigungs- und Verbindungsknoten

Aus dem UML-Modell in Abb. 34 wird folgender Code generiert. Aus Gründen der Übersichtlichkeit wurden die Funktion „rootState_entDef" sowie der generierte Code aus „LED_Aktion2" nicht mit aufgeführt.

```
1   static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
2
3       LED * const me = (LED *)void_me;
4       RiCTakeEventStatus res = eventNotConsumed;
5       switch (me->rootState_active) {
6           /* State Aktion1 */
7           case LED_Aktion1:
8           {
9               if(id == Null_id)
10                  {
11                      /*## transition 2 */
12                      if(//Bedingung1)
13                          {
14                           ...//hier fehlt Code aus Gründen der Übersichtlichkeit!!!
15                          }
16                      else
17                          {
18                              /*## transition 3 */
19                              if(//Bedingung2)
20                                  {
21                                   ... //hier fehlt Code aus Gründen der
22                                       //Übersichtlichkeit!!!
23                                  }
24                          }
25                  }
26          }
27          break;
28          /* State Aktion3 */
29          case LED_Aktion3:
30          {
31              /*## transition 4 */
```

```
32              if(id == Null_id)
33                  {
34                      WST_FSM_popNullConfig(&(me->ric_reactive));
35                      WST_FSM_pushNullConfig(&(me->ric_reactive));
36                      me->rootState_subState = LED_Aktion1;
37                      me->rootState_active = LED_Aktion1;
38                      me->ric_reactive.currentState = me->rootState_active;
39                      {
40                          /*#[ state Aktion1.(Entry) */
41                          //ToDo Aktion 1;
42                          /*#]*/
43                      }
44                      res = eventConsumed;
45                  }
46          }
47          break;
48          default:
49              break;
50      }
51      return res;
52  }
```

Das in Abb. 34 erstellte Aktivitätsdiagramm stellt neben Aktionen auch Verzweigungs- und Verbindungsknoten dar, welche als Kontrollelemente eingesetzt werden. Die Funktion „rootState_entDef" wird hier nicht beschrieben, da die Aufgaben der Funktion bereits in Abschnitt 4.2.1 detailliert aufgeführt sind. Der Verzweigungsknoten, welcher dazu dient, eine Kante in mehrere Alternativen aufzuspalten, wird als if-else Bedingung im Code dargestellt. Diese if-else Bedingung ist in den Codezeilen 12-24 zu sehen. Zuerst wird überprüft, ob Bedingung 1 erfüllt ist; wenn nicht, wird anschließend mit einer erneuten if-Abfrage Bedingung 2 überprüft. Der Code, welcher sich innerhalb der if-Bedingungen befindet, wurde aus Gründen der Übersichtlichkeit hier vernachlässigt.

Der Verbindungsknoten stellt ein „explizites ODER" dar. Im Code ist dieser nicht ersichtlich. Würde man den Verbindungsknoten entfernen und Aktion3 und Aktion2 jeweils direkt über eine Kante mit Aktion1 verbinden, so würde dies eine „implizite UND" Verbindung darstellen. Diese wird bei der Codegenerierung ebenfalls nicht berücksichtigt (Zeile 29-47). Somit wird kein Unterschied zwischen einem „expliziten ODER" und einem „impliziten UND" ersichtlich.

Im Vorfeld wurden lediglich die Notationselemente beschrieben und genauer erklärt, welche auch im folgenden Beispiel Verwendung finden.
Die anderen, jedoch ebenfalls wichtigen Elemente sowie der aus ihnen erzeugte Quellcode, sind im Anhang hinterlegt.

4.3 Komparator als einfaches Beispiel

Aktivitätsdiagramme werden eingesetzt, um ein kontinuierliches Verhalten zu modellieren. Dies bedeutet, dass Aktionen keine Zustände wie im Zustandsdiagramm darstellen. Wenn eine Aktion ausgeführt ist, folgt anschließend ohne Unterbrechung die nächste Aktion.
Des Weiteren wird beschrieben, wie ein Projekt in Rhapsody angelegt wird, wie das modellierte Aktivitätsdiagramm aussieht, und welcher Code daraus generiert wird. Abschließend wird das Ergebnis dieses ersten Beispiels vorgestellt.

4.3.1 Anlegen eines Projekts unter Rhapsody

Im Folgenden wird aufgezeigt, wie das Rhapsody-Projekt Komparator angelegt wird. Dieses Projekt stellt eine einfache Verwendung von Aktivitätsdiagrammen dar. Die Stellung des Potentiometers des MCB1700 Boards soll erfasst und anhand dieser eine entsprechende Led eingeschaltet werden.
Über Menü File/New kann ein neues Projekt angelegt werden. Es erscheint folgendes Fenster.

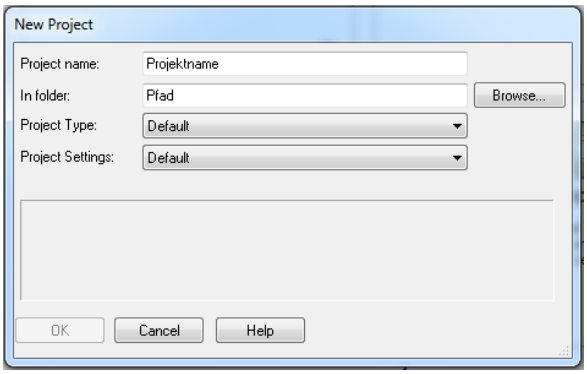


Abb. 35 Neues Projekt anlegen

Hier wird der Projektname eingetragen sowie der Pfad für das Projekt gewählt und anschließend mit OK bestätigt. Die nächste Abfrage wird ebenfalls mit JA bestätigt. Anschließend erscheint der in nachfolgender Abbildung dargestellte Bildschirm.

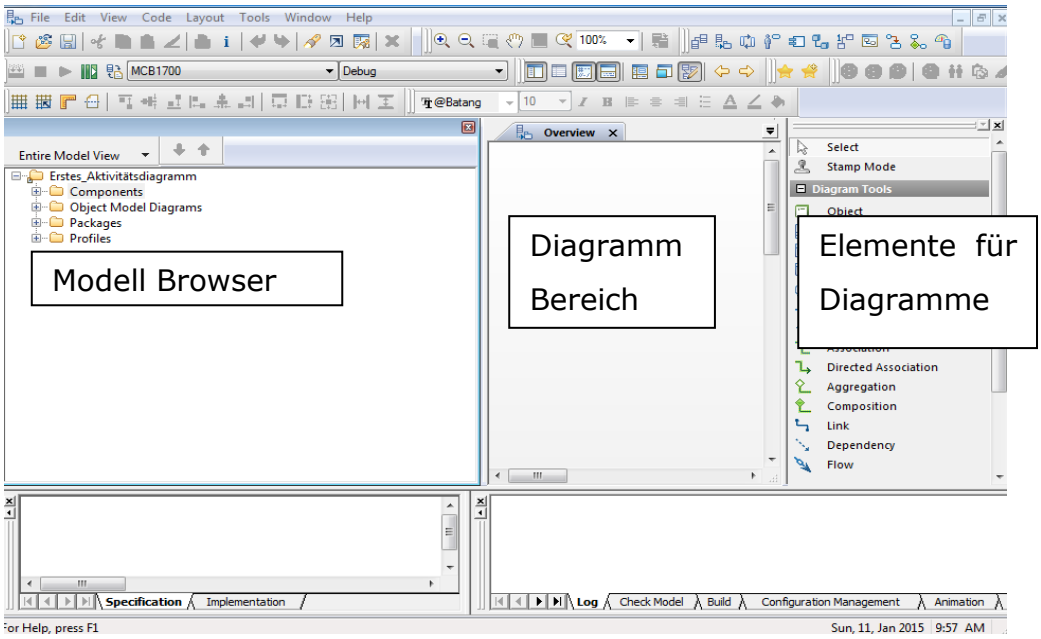


Abb. 36 Rhapsody Fenster

Im nächsten Schritt werden Profile hinzugefügt. Mit Profilen kann man zusätzliche Einstellungsmöglichkeiten zu Rhapsody anfügen und die Codegeneration unter Rhapsody wird beeinflusst, sodass diese mit dem Framework von Willert zusammenarbeitet.

Im Menü File/Add Profile to Model…/WST_RXF_V6 müssen Profile hinzugefügt werden.

Das „Rpy_C_OO RTX_Keil_ARM_MCB1700_Eval_TC_Profile.sbs" gehört zum Willert Software Tools RXF und muss in alle Modelle inkludiert werden.

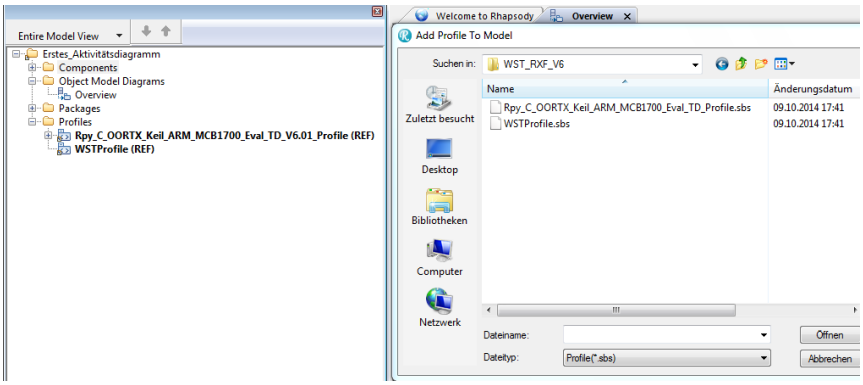


Abb. 37 Profile hinzufügen

Dieses Profil ermöglicht die Verwendung weiterer Stereotypen. Das „WSTProfile.sbs" ist ein optionales Profil. Es ändert die Einstellungen von Rhapsody hinsichtlich des „Look and Feel".

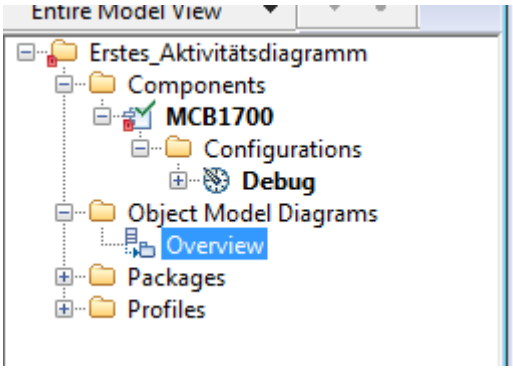


Abb. 38 Einstellungen Modell Browser

Im Modell Browser können nun einige Einstellungen geändert werden. Die „DefaultComponent" kann zu MCB1700, „DefaultConfig" zu Debug und „Model1" zu Overview umbenannt werden.
Durch einen Doppelklick auf das MCB1700 öffnet sich folgendes Fenster. Hier muss der Stereotype „RXFComponent" gewählt und als Standard Headers „<lpc17xx.h>" eingetragen werden.

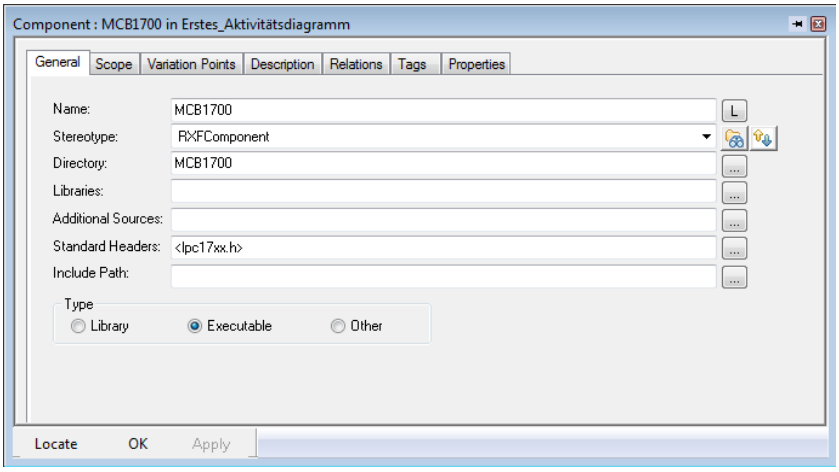


Abb. 39 Einstellungen Component

Nun kann das OMD (Object Model Diagramm) Overview geöffnet und dort eine Klasse mit dem zugehörigen Aktivitätsdiagramm angelegt werden.

Um die Erklärung des generierten Codes möglichst einfach zu gestalten, gibt es in dem Programm nur eine Klasse mit der Bezeichnung „LED". Diese Klasse besitzt drei Attribute (BitNr, Delay, Value) und vier Operationen (Init, off, on, Sampling). Zusätzlich ist der Klasse LED das Aktivitätsdiagramm (Activity) hinzugefügt.

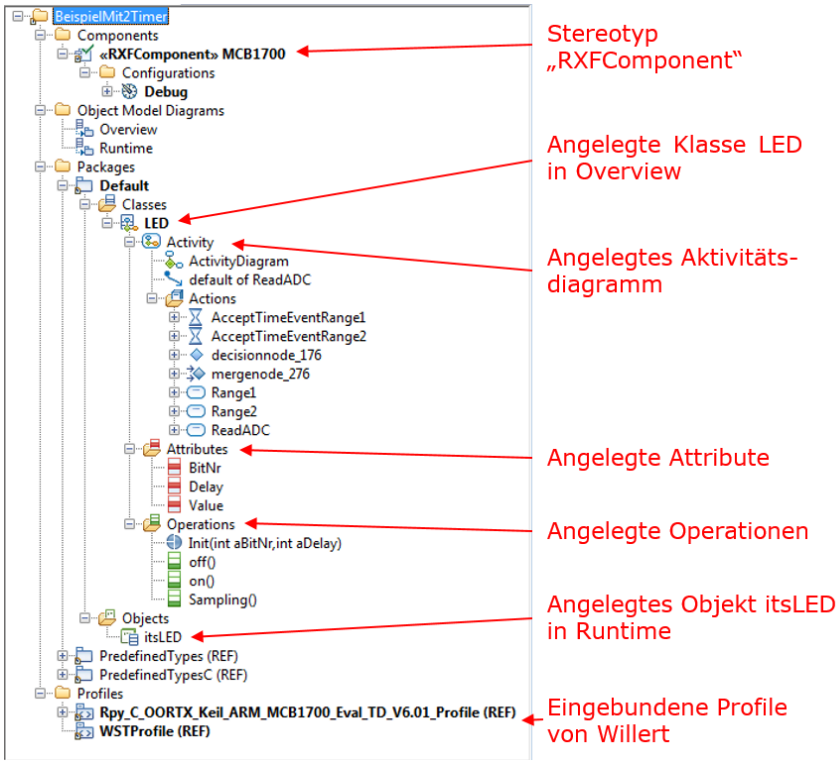


Abb. 40 Beschreibung Modell Browser

In der nachfolgenden Abbildung sind die angelegte Klasse „LED" im OMD Overview und das Objekt „itsLED" im OMD Runtime abgebildet. Die Argumente des Initializers werden beim Anlegen des Objekts „itsLED" durch den Benutzer eingegeben.

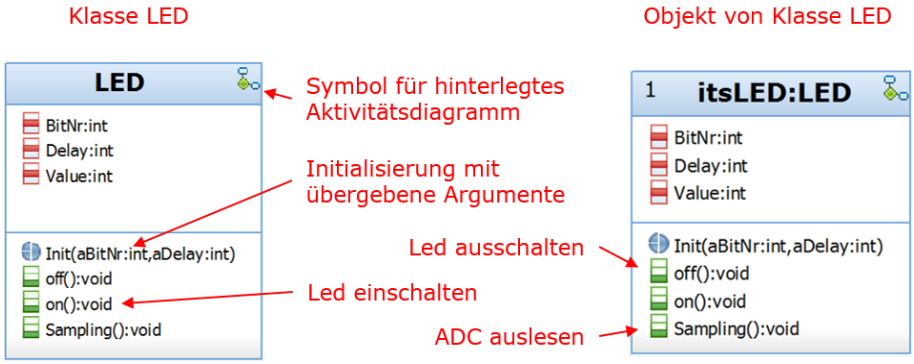


Abb. 41 OMD Overview und Runtime

Mit dem Attribut „BitNr" wird festgelegt, welche Led ein- bzw. ausgeschaltet werden soll. In das Attribut „Value" wird der ausgelesene Wert des AD-Wandlers geschrieben. Das Attribut „Delay" wird in diesem Beispiel noch nicht verwendet.
Der Code, welcher durch die Operationen der Klasse entsteht, kann in Anhang 8.2.1 eingesehen werden.

4.3.2 Aktivitätsdiagramm Klasse LED

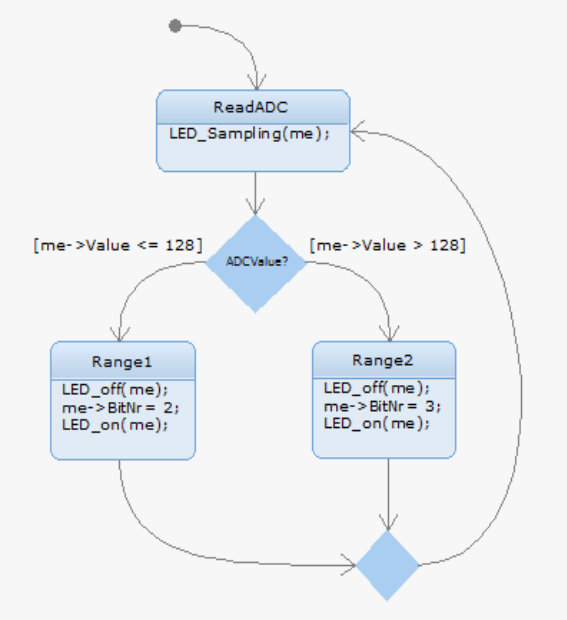


Abb. 42 Beispiel Komparator

Das in Abb. 42 dargestellte Aktivitätsdiagramm soll folgende Aufgabe erfüllen. Die Analogspannung des Potentiometers, welches sich auf dem MCB1700 befindet, soll kontinuierlich digital gewandelt werden und je nachdem, welcher Wert sich daraus ergibt, eine entsprechende Led einschalten.
Diese Aufgabe soll dauerhaft ausgeführt werden.
Nach dem Start der Aktivität wird als erstes die Aktion „ReadADC" angelaufen. In dieser Aktion wird die Operation „Sampling" aufgerufen. Wenn der Ablauf der Aktion fertig durchlaufen ist, folgt ein Verzweigungsknoten, an dem der AD-Wandler Wert, der im Attribut „Value" gespeichert ist, überprüft wird. Ist dieser Wert kleiner oder gleich 128, wird als nächstes die Aktion „Range1" angelaufen, ist der Wert größer als 128, wird „Range2" angelaufen. In den Aktionen werden durch die Operation „off" zunächst die aktuelle Led ausgeschaltet, dann in das Attribut „BitNr" die einzuschaltende Led-Nummer gesetzt und anschließend durch die Operation „on" die jeweilige Led eingeschaltet. Schließlich werden die ausgehenden Kanten der Aktionen „Range1" und „Range2" wieder durch einen Verbindungsknoten zusammengefasst. Die ausgehende Kante des Knotens läuft wieder zurück in die Aktion „ReadADC", sodass ein Kreislauf entsteht.

4.3.3 Generierter Code aus Aktivitätsdiagramm

Im Folgenden ist ein Auszug aus dem Code des Aktivitätsdiagrammes aus Abschnitt 4.3.2 dargestellt. Der vollständige Code kann in Anhang 8.2.7 eingesehen werden.

Wie man im UML-Diagramm in Abb. 42 sehen kann, ist eine Aktion mit dem Namen „ReadADC" angelegt worden. Diese Aktion wird wie folgt im Code dargestellt.

```
1   /* State ReadADC */
2           case LED_ReadADC:
3           {
4               if(id == Null_id)
5                   {
6                       /*## transition 0 */
7                       if(me->Value > 128)
8                           {
9                               WST_FSM_popNullConfig(&(me->ric_reactive));
10                              WST_FSM_pushNullConfig(&(me->ric_reactive));
11                              me->rootState_subState = LED_Range2;
12                              me->rootState_active = LED_Range2;
13                              me->ric_reactive.currentState = me->rootState_active;
14                              {
15                                  /*#[ state Range2.(Entry) */
16                                  LED_off(me);
17                                  me->BitNr = 3;
18                                  LED_on(me);
19                                  /*#]*/
20                              }
21                              res = eventConsumed;
22                          }
```

Es erfolgt in Zeile 2, wie in einem Zustandsdiagramm, eine Switch-Case Abfrage, um die aktuell aktive Aktion zu ermitteln. Anschließend wird in Zeile 4 die Event ID überprüft.

Da in Aktivitätsdiagrammen nur Null-Transitionen modelliert werden können, ist diese ID immer eine „Null_id". In Aktivitätsdiagrammen treten normalerweise keine Events auf, die das Wechseln in die nächste Aktion auslösen.

Null-Transitionen sind Transitionen, bei denen nur ein Guard und kein Trigger (Event) angegeben werden kann. Dies bedeutet, dass der Ablauf nicht durch das Auftreten von äußeren Ereignissen (Events) beeinflusst werden kann.

Die if-Abfrage in Zeile 7 gibt den Verzweigungsknoten aus dem UML Modell wieder. Dieser überprüft, in welchem Bereich, Range1 oder Range2, sich der gewandelte Wert befindet. Ist der Wert kleiner als 128, so wird in die Aktion Range1 gewechselt.

4.3.4 Ergebnis

Aus dem in Abschnitt 4.3.2 erstellten Aktivitätsdiagramm wurde ein Code generiert und auf das MCB1700 Board gespielt.

Als Ergebnis war zu sehen, dass nach dem Start des Programmes auf dem Board genau die Led geleuchtet hat, in deren Bereich das Potentiometer zu Beginn gestellt war. Befand sich das Potentiometer in „Range1", so hat die entsprechende Led geleuchtet.

Befand sich das Potentiometer beim Start im Bereich „Range2", so hat die Led für den Bereich Range2 geleuchtet.

Wurde das Potentiometer bei laufendem Programm in einen anderen Bereich verstellt, so hat sich die Led für diesen Bereich nicht eingeschaltet und die Led für den aktiven Bereich nicht ausgeschaltet. Aufgrund dieser Tatsache wurde das Programm im Debug-Modus in Keil gestartet.

Dabei wurde ein Breakpoint an folgender Codezeile (siehe Anhang 8.2.7 Codezeile 17) gesetzt.

```
•  static RiCTakeEventStatus rootState_dispatchEvent
(void * const void_me, short id)
```

Diese Zeile wird bei jedem Aufruf des Aktivitätsdiagrammes ausgeführt. Das Programm wurde mit RUN gestartet und hat an dem dargestellten Breakpoint angehalten. Es war zu beobachten, dass das Programm genau sieben Mal dort stehen blieb und anschließend den Haltepunkt nicht mehr erreichte. Daraufhin wurde das Programm gestoppt und man konnte im Disassembly Fenster in Keil erkennen, dass das Programm in einer Error-Schleife des OO RTX steht. Diese ist als Endlosschleife realisiert, sodass nichts anderes mehr ausgeführt wird.

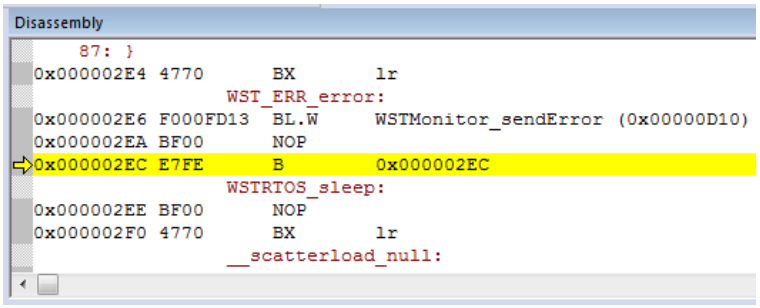


Abb. 43 Errorhandler

Aufgrund dieses Verhaltens wurde das Unternehmen Willert per Email kontaktiert, worauf Herr Walter van den Heiden antwortete, dass das System die Null-Transitionen erfasst. Wenn sieben hintereinander auftreten, wird der Errorhandler aktiv und das Programm läuft in die Endlosschleife. Dies soll verhindern, dass an irgendeiner Stelle im Programm Endlosschleifen realisiert sind.

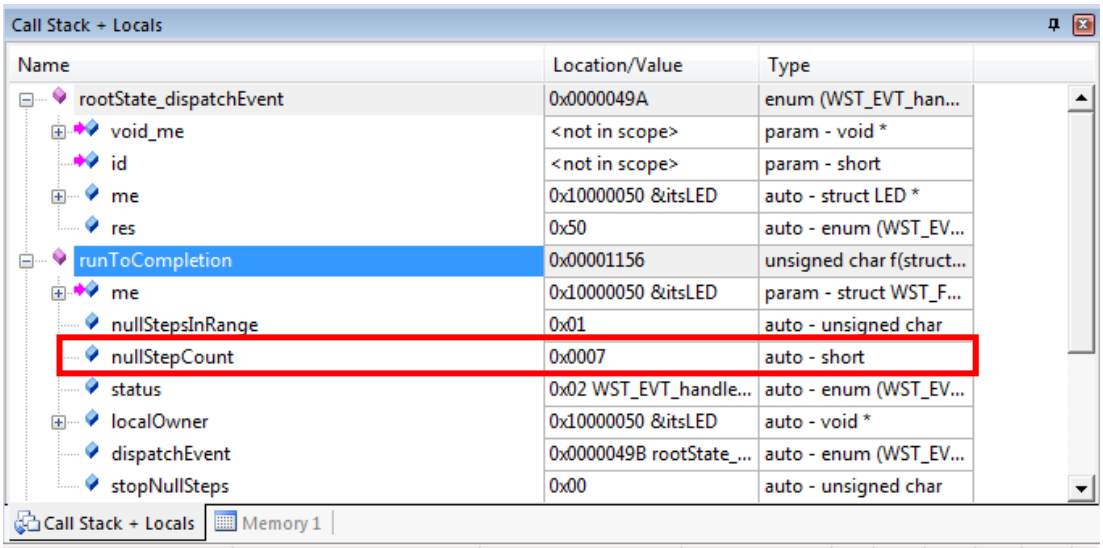


Abb. 44 Null-Transitionen Count

Während des Debuggens besteht auch die Möglichkeit, die Anzahl der bereits erfassten Null-Transitionen zu beobachten. Die zu beobachtende Variable ist in Abb. 44 hervorgehoben. Immer wenn eine Null-Transition auftritt, wird die Variable „nullStepCount" inkrementiert, solange bis sie den Wert sieben erreicht hat. Dann wird in die Errorschleife gesprungen. Aufgrund dieser Beschränkung ist ein kontinuierliches Modellieren mit Aktivitätsdiagrammen, wie zu Beginn angenommen, nicht gegeben.

In Absatz 4.4 wird beschrieben, wie man diese Einschränkung durch einen Workaround umgehen kann.

4.4 Komparator mit Workaround

Anhand dieses Beispiels wird erläutert, wie die in Abschnitt 4.3.4 festgestellte Beschränkung durch eine Änderung in der Modellierungsweise umgangen werden kann.
Es wird zunächst das UML Modell vorgestellt, anschließend der Code und die Änderung des UML Modells und abschließend das Ergebnis dieses Beispiels dargestellt.

4.4.1 Aktivitätsdiagramm der Klasse LED

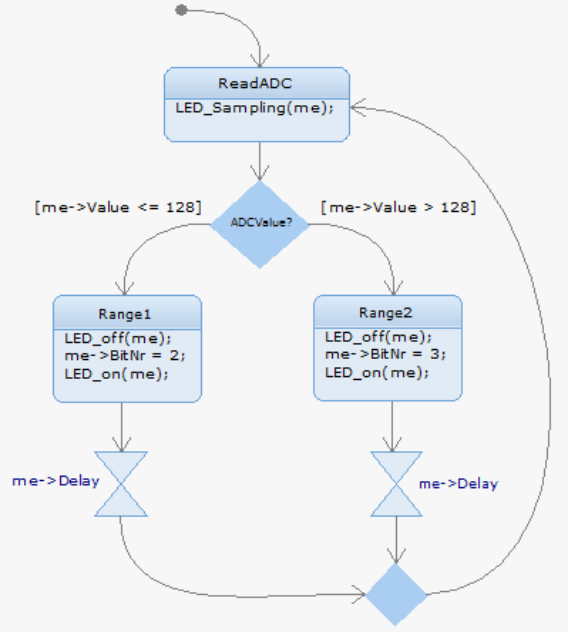


Abb. 45 Komparator mit Workaround

Das in Abb. 45 dargestellte Aktivitätsdiagramm besitzt die gleiche Funktionalität wie das Diagramm in Abschnitt 4.3.2.
Diesem Modell wurden zusätzlich Accept Time Events hinzugefügt. Dies soll verhindern, dass der Errorhandler des OO RTX aufgerufen wird und das Programm in die Endlosschleife läuft. Warum dies der Fall ist, wird im nächsten Abschnitt anhand des Codes verdeutlicht. Wird das Zeitereignis vor der Decision-Node gesetzt, so wird der gleiche Effekt erzielt und der Errorhandler nicht aufgerufen. Platziert man das Zeitereignis jedoch nach der Merge-Node, so erkennt der Codegenerator von Rhapsody das Zeitereignis nicht und generiert daraus keinen Code.

4.4.2 Generierter Code aus Aktivitätsdiagramm

Im Folgenden ist ein Codeausschnitt des Aktivitätsdiagrammes aus Abschnitt 4.4.1 dargestellt. Der Code ist bis auf die Zeilen 20 und 21 identisch mit dem in Absatz 4.3.3 vorgestellten Code.

```
1   /* State ReadADC */
2           case LED_ReadADC:
3           {
4               if(id == Null_id)
5                   {
6                       /*## transition 0 */
7                       if(me->Value > 128)
8                           {
9                               WST_FSM_popNullConfig(&(me->ric_reactive));
10                              me->rootState_subState = LED_Range2;
11                              me->rootState_active = LED_Range2;
12                              me->ric_reactive.currentState = me->rootState_active;
13                              {
14                                  /*#[ state Range2.(Entry) */
15                                  LED_off(me);
16                                  me->BitNr = 3;
17                                  LED_on(me);
18                                  /*#]*/
19                              }
20                              WST_DUMMY_TASK_schedTm(me->ric_reactive.myTask, me->
21  Delay, LED_Timeout_Range2_id, &(me->ric_reactive), NULL);
22                              res = eventConsumed;
23                          }
```

Die Änderung des Codes bewirkt, dass ein Timeout Event mit der Event ID „Timeout_id" generiert wird (Codezeile 20, 21).

```
1   /* State Range2 */
2           case LED_Range2:
3           {
4               /*## transition 5 */
5               if(id == Timeout_id)
6                   {
7                       if(RiCTimeout_getTimeoutId((RiCTimeout*) me->
8   ric_reactive.current_event) == LED_Timeout_Range2_id)
9                           {
10                              WST_DUMMY_TASK_unschedTm(me->ric_reactive.myTask,
11  LED_Timeout_Range2_id, &(me->ric_reactive));
12                              WST_FSM_pushNullConfig(&(me->ric_reactive));
13                              me->rootState_subState = LED_AcceptTimeEventRange2;
14                              me->rootState_active = LED_AcceptTimeEventRange2;
15                              me->ric_reactive.currentState = me->rootState_active;
16                              res = eventConsumed;
17                          }
18                  }
19          }
20          break;
```

Durch diese Modellierungsweise wird eine Nicht Null-Transition (siehe Codezeile 5) erzeugt.

4.4.3 Ergebnis

Aus dem Aktivitätsdiagramm wurde ein Code generiert und auf das MCB1700 gespielt.
Das Programm wurde gestartet und es leuchtete die entsprechende Led für den Bereich. Auch bei Verstellen des Potentiometers in den anderen Bereich wird die Led gewechselt.
Um genauer nachvollziehen zu können, warum dieses Programm funktionsfähig ist, wurde es im Debug Modus gestartet und ein Breakpoint an der Codestelle (siehe Anhang 8.2.8 Zeile 17)

- `static RiCTakeEventStatus rootState_dispatchEvent (void * const void_me, short id)`

erstellt. Es ist zu beobachten, dass die Variable „nullStepCount" (siehe Abb. 44), welche die Anzahl der aufgetretenen Null-Transitionen erfasst, auf zwei erhöht wird, aber nach dem Erreichen einer Nicht-Null-Transition wieder auf eins zurückgesetzt wird. Dies ermöglicht die Verwendung von Aktivitätsdiagrammen.

4.5 Fazit

In diesem Kapitel wurde anhand des Beispiels Komparator getestet, wie die Codegenerierung von Aktivitätsdiagrammen in Rhapsody mit der Willert Software aussieht. Die Aufgabe des Aktivitätsdiagrammes dort war es, kontinuierlich eine Analogspannung digital zu wandeln und das Ergebnis dieser Wandlung optisch an den Benutzer auszugeben.
Dabei wurde das Problem aufgedeckt, dass das Echtzeitbetriebssystem Null-Transitionen, welche bei jeder Modellierung einer Kante entstehen, erfasst und nach siebenmaligem Auftreten der Errorhandler aktiv wird.
Aufgrund dieses Problems wurde ein zweites Beispiel mit einem Workaround vorgestellt. Bei diesem Aktivitätsdiagramm wird durch Einsatz eines Accept Time Events eine Nicht-Null Transition erzeugt, die bewirkt, dass die Variable, welche die Anzahl der bereits aufgetreten Null Transitionen erfasst, zurückgesetzt wird und dadurch der Errorhandler nicht eingreift.
Wegen der Beschränkung des Willert OO RTX ist, wie zu Beginn der Arbeit angenommen, eine kontinuierliche Modellierung mit Aktivitätsdiagrammen nicht möglich.
Aus diesem Grund wurde die Bearbeitung dieser Thematik beendet und ein neues Thema, welches im weiteren Verlauf dieser Arbeit vorgestellt wird, aufgenommen.

5 Einbinden der Ethernet Schnittstelle

Dieses Kapitel befasst sich mit dem Einbinden der Ethernet Schnittstelle des MCB1700 Boards in ein UML-Modell, sodass Nachrichten gesendet und empfangen werden können.
Zuerst werden die verwendete Software und das eingesetzte Echtzeitsystem vorgestellt. Anschließend wird nochmals detaillierter auf das Echtzeitsystem eingegangen.
In Abschnitt 5.2 wird das UML-Modell, welches die Ethernet Schnittstelle des MCB1700 verwendet, vorgestellt. Dort wird aufgezeigt, wie ein Projekt in Rhapsody angelegt wird; zudem werden die Funktionen der einzelnen Klassen vorgestellt und wichtige Einstellungsmöglichkeiten aufgezeigt. Abschließend wird ein Fazit über die Einbindung der Ethernet Schnittstelle gezogen.

5.1 Verwendete Software und Echtzeitsystem

Zur Modellierung wird das Softwarepackage Rpy_C_KeilRTX_Keil5_CM3_MCB1700_RC_V6.00 von Willert verwendet. Dieses beinhaltet Rhapsody 8.0.1 als UML-Modellierungswerkzeug, das Embedded UML RXF von Willert, Keil µVision 5 als IDE und verwendet das Echtzeitsystem Keil RTX, wie folgende Tabelle verdeutlicht.

Tabelle 3 Software und Echtzeitsystem

RXF	Willert RXF
IDE	Keil µVision 5
Modellierung	IBM Rational Rhapsody
Sprache	ANSI C
RTOS	Keil RTX (RL-ARM)
Compiler und Zielplattform	Keil MDK-ARM
Board	Keil MCB 1700

Grund für die Verwendung einer anderen Software als im ersten Teil der Arbeit ist, dass hier das Echtzeitsystem Keil RTX in Verbindung mit dem Willert RXF verwendet werden kann und man die RL-ARM Library von Keil einsetzen kann. Das Keil RTX unterstützt im Gegensatz zum Willert OO RTX die Kommunikation über TCP/IP, welche in diesem Teil der Arbeit benötigt wird. Um die RL-ARM Library verwenden zu können, wird eine Vollversion von Keil µVision 5 benötigt und es muss zusätzlich das Keil RTX Legacy Package nachinstalliert werden, da das Willert RXF noch nicht alle Interfaces des CMSIS von Keil µVision 5 verwenden kann. (s. Abschnitt 2.2.3)

Keil RTX

Das Keil RTX Real-Time-Operating System (RTX-RTOS) ist sehr leistungsstark und kann für Mikrocontroller, welche auf einem Cortex-M CPU Kern basieren, eingesetzt werden. Das RTX Kernel kann man für Applikationen, die mehrere Aufgaben gleichzeitig ausführen müssen (Multitasking), verwenden. (vgl. [11])

Die RL-TCPnet Library ist eine Komponente der RL-ARM Library und ermöglicht das einfache Verwenden von UDP Socket und das Senden und Empfangen von Nachrichten via User Datagram Protocol (UDP). Das UDP wird aufgrund der einfacheren Implementierung und geringeren zu sendenden Datenmenge anstelle von TCP/IP verwendet.

Folgende UDP Funktionen werden durch die RL-TCPnet Library zur Verfügung gestellt. (vgl. [12])

Tabelle 4 UDP Routinen

Funktion	**Beschreibung**
udp_get_socket	Allokiert einen UDP Socket
udp_open	Öffnet einen UDP Socket für die Kommunikation
udp_close	Schließt einen UDP Socket
udp_release_socket	Gibt UDP Socket frei
udp_get_buf	Allokiert Speicher für einen UDP send buffer
udp_send	Sendet ein UDP Packet

Eine weitere wichtige Funktion, welche für die Ethernet Kommunikation via UDP Protokoll benötigt wird, ist die „main_TcpNet". Diese Funktion ist ein Teil der RL-TCPnet Library. Sie pollt den Ethernet Kontroller des NXP LPC1768 auf empfangene Daten.
Stellt die „main_TcpNet" Funktion fest, dass Daten empfangen wurden, so wird eine entsprechende Funktion aufgerufen, um die Daten zu verarbeiten. Anschließend werden diese an die User-Applikation weitergereicht. Wichtig ist, dass die „main_TcpNet" Funktion zyklisch aufgerufen wird, da ansonsten das TCPnet System nicht funktioniert. Aus diesem Grund wird diese Funktion, wie später zu sehen ist, in einem extra dafür angelegten Task zyklisch ausgeführt. (vgl. [13])
Eine genauere Darstellung der Funktionen, welche für die Ethernet Kommunikation benötigt werden, erfolgt in Abschnitt 5.2.5.

5.2 UML-Modell Ethernet Schnittstelle

Durch ein in Rhapsody erstelltes UML-Modell soll es möglich sein, zwischen MCB1700 Boards per Ethernet Daten auszutauschen. Dazu wird die auf den Boards vorhandene Ethernet Schnittstelle verwendet. Um Daten zwischen den Boards zu transportieren, ist entweder eine direkte Verbindung miteinander über ein gekreuztes Ethernet Kabel notwendig, oder die MCB1700 Boards werden über ein Ethernet Patch Kabel an ein Netzwerk angeschlossen, beispielsweise an einen Switch. Das erstellte UML-Modell erfüllt dann folgende Funktionen.

- Bei Betätigung des Joysticks wird auf dem über Ethernet verbundenen MCB1700 Board eine bestimmte Led eingeschaltet.
- Ebenso ist es möglich, vier Leds gleichzeitig einzuschalten.
- Die Übertragung der Daten soll in beide Richtungen möglich sein, das heißt jedes Board soll als Sender und als Empfänger verwendet werden können.

Mit dem folgenden Bild soll die Funktionsweise des Modells nochmals näher erläutert werden.

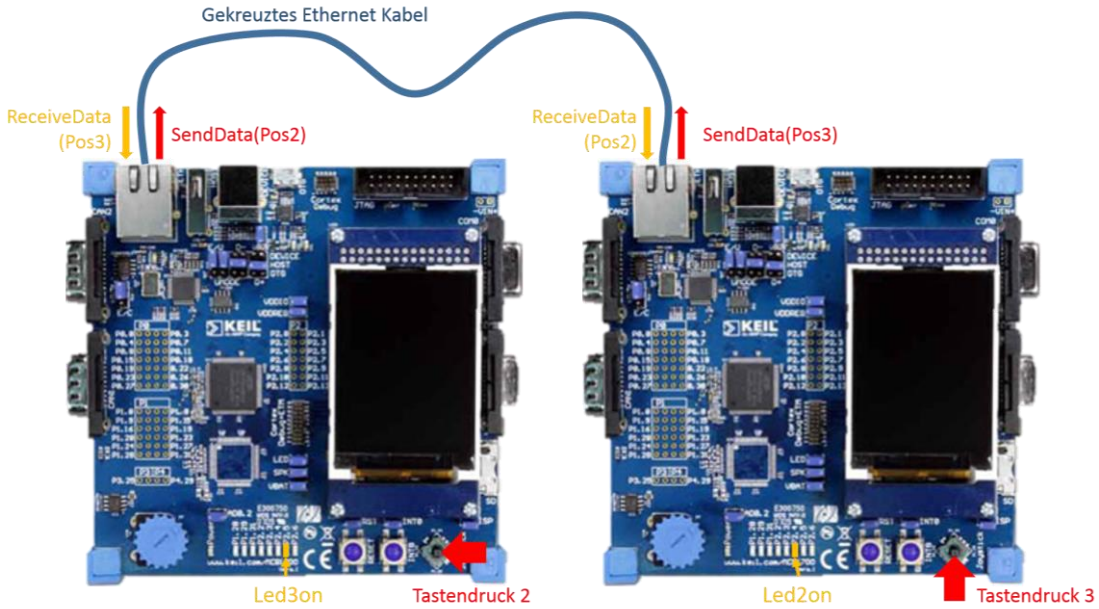


Abb. 46 Kommunikation MCB1700 Boards

In Abb. 46 sind zwei MCB1700 Boards dargestellt, welche über ein gekreuztes Ethernet Kabel miteinander verbunden sind. Um nun Leds auf dem verbundenen Board über Ethernet einzuschalten, muss der Joystick betätigt werden. Für jede Richtung, in die der Joystick geschaltet wird, wird die jeweilige Joystickposition versendet (das Schalten des Joysticks wird im Bild durch einen dicken roten Pfeil verdeutlicht). Wenn die Daten auf der Empfängerseite angekommen sind, werden sie ausgewertet und eine entsprechende Led eingeschaltet. Der Joystick auf dem Board ist dabei mit folgenden Funktionen belegt.

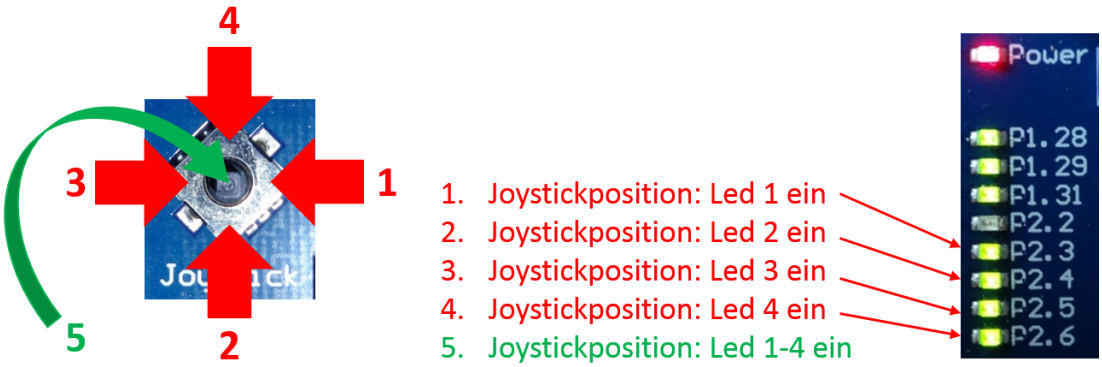


Abb. 47 Bedeutung Position Joystick und LedBar

Nach Beschreibung der Funktionsweise des Systems wird nun das UML-Modell genauer erläutert.

5.2.1 Anlegen eines Projektes

Im Folgenden wird beschrieben, wie ein Projekt unter Rhapsody angelegt wird. Dieses soll die in Abschnitt 5.2 beschriebene Funktionalität realisieren. Der Ablauf des Anlegens ist größtenteils identisch mit dem im Abschnitt 4.3.1. Der einzige Unterschied besteht darin, dass dem Projekt andere Profile hinzugefügt werden müssen. Wie in Abb. 48 zu sehen ist, handelt es sich um die folgenden drei Profile, welche über File/ Add Profile to Model…/WST_RXF_V6 angefügt werden.

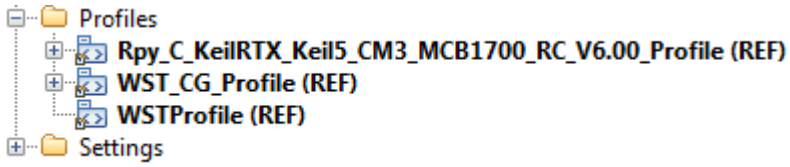


Abb. 48 Modell Browser Profiles

Das Rpy_C_KeilRTX_Keil5_CM3_MCB1700_RC_V6.00_Profile gehört zum Willert RXF und ermöglicht die Verwendung weiterer Stereotypen. Das WST_CG_Profile beinhaltet Target spezifische Stereotypen, welche die Codegeneration modifizieren. Das WST Profile ist wiederum optional. Es ändert die Einstellungen von Rhapsody hinsichtlich des „Look and Feel".

5.2.2 Verwendete Konstanten und Events

In Abb. 49 sind die verwendeten Types (Konstanten) und Events, welche im Rhapsody UML-Modell verwendet werden, zu sehen.

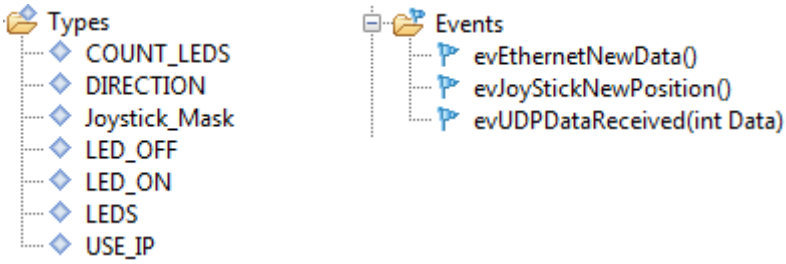


Abb. 49 Verwendete Konstanten und Events

Im Folgenden werden die Konstanten und Events genauer beschrieben.

Konstante COUNT_LEDS

„COUNT_LEDS" ist ein Define, welcher die Anzahl der Leds der LedBar vorgibt.

```
#define COUNT_LEDS 4
```

Konstante LED_OFF und LED_ON

„LED_OFF" und „LED_ON" sind ebenfalls Defines zum Vergleichen des aktuellen Status einer Led.

```
#define LED_ON 1
#define LED_OFF 0
```

Konstante Joystick_Mask

„Joystick_Mask" ist ein Define, welcher für das Auslesen des Joysticks benötigt wird.

```
#define Joystick_Mask 0x79
```

Konstante DIRECTION

„DIRECTION" ist eine Konstante vom Typ Enumeration, durch welche die ausgelesene Position des Joysticks einfach mit ihren Items verglichen werden kann. Sie ist folgendermaßen aufgebaut.

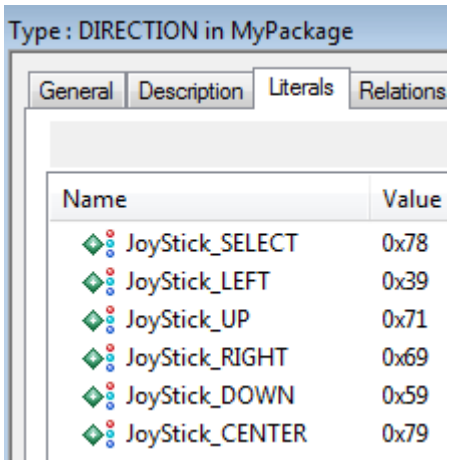


Abb. 50 Literals DIRECTION

Konstante LEDS

„LEDS" ist eine Konstante vom Typ Enumeration und sieht wie folgt aus.

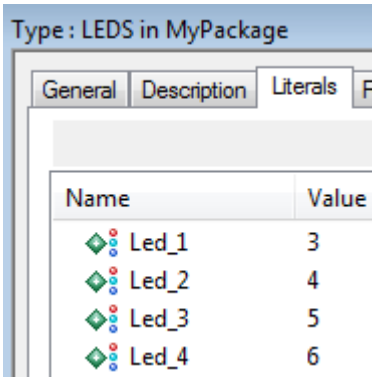


Abb. 51 Literals LED

Das Verwenden dieser Items ermöglicht es, der LedBar einfach mitzuteilen, welche Led eingeschaltet werden soll.

Konstante USE_IP

„USE_IP" ist eine Konstante vom Typ Enumeration und kann zum Erkennen des Empfängers verwendet werden.

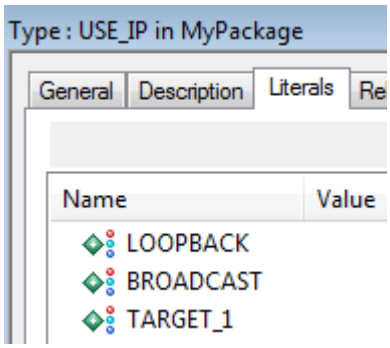


Abb. 52 Literals USE_IP

evEthernetNewDate()

Dieses Event wird von der Klasse „Ethernet" an die Klasse „Controller" gefeuert. Dadurch wird der Klasse „Controller" mitgeteilt, dass neue Daten empfangen wurden.

evJoyStickNewPosition()

Die Klasse „JoyStick" feuert dieses Event bei einer neuen Position des Joysticks an die Klasse „Controller". Somit kann die Klasse „Controller" die neue Position per Ethernet an das andere MCB1700 Board versenden.

evUDPDataReceived(int Data)

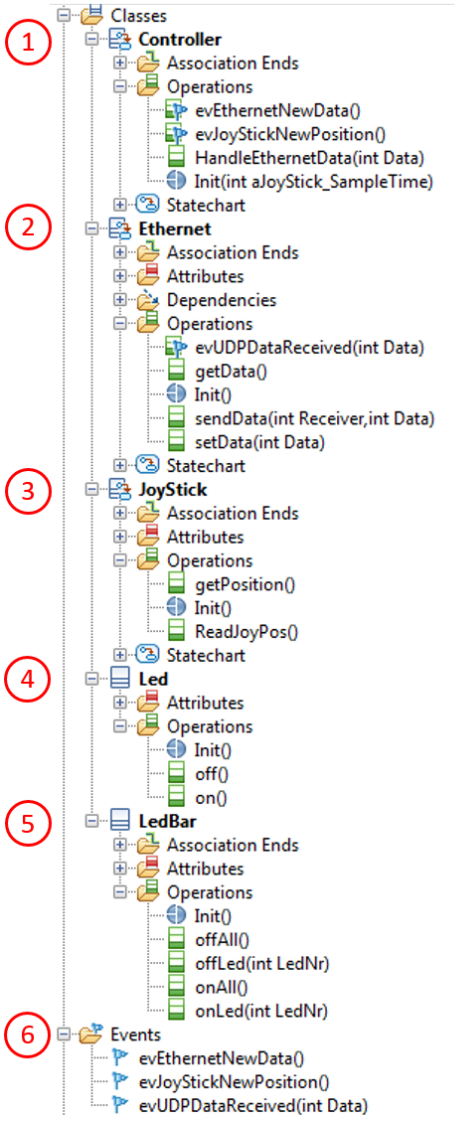
Dieses Event wird aus der Library „LibEthernet" an die Klasse „Ethernet" im UML-Modell gefeuert. Es stellt die Schnittstelle zwischen Library und UML-Modell dar. Die Library verwaltet die Ethernet Funktionalität. Beim Empfang von Daten wird das Event gefeuert und als Parameter die empfangenen Daten übergeben. Deswegen ist es wichtig, dass es in Rhapsody erstellt und von der Library verwendet werden kann. Die Verwendung des Events kann sowohl in Abb. 58, als auch im Code der Library in Anhang 8.3 eingesehen werden.

Die Rahmenbedingungen für das Projekt sind nun vollständig. Es wurden alle notwendigen Profile hinzugefügt und alle Konstanten und Events sind definiert. Nun können die OMDs modelliert werden.

5.2.3 OMD Overview

Um eine übersichtliche Darstellung zu erreichen, werden zwei OMDs angelegt. In dem OMD Overview werden alle benötigten Klassen mit ihren Attributen und Operationen modelliert. Im OMD Runtime wird ein Objekt aus der zentralen Klasse erzeugt.

Um eine Übersicht über alle in diesem Projekt angelegten Klassen, deren Operationen und die verwendeten Events zu geben, wird in nachfolgender Abbildung ein Teil des Model Browsers gezeigt.



1. Klasse Controller in Overview

2 x Eingehende Events

2 x Operationen

Hinterlegtes Statechart

2. Klasse Ethernet in Overview

1 x Eingehende Events

4 x Operationen

Hinterlegtes Statechart

3. Klasse Joystick in Overview

3 x Operationen

Hinterlegtes Statechart

4. Klasse Led in Overview

3 x Operationen

5. Klasse LedBar in Overview

5 x Operationen

6. Angelegte Events

Abb. 53 Model Browser Projekt Ethernet

Das UML-Modell wird aus den fünf Klassen „Controller", „Ethernet", „Joystick", „Led" und „LedBar" aufgebaut. Wie in obenstehender Abbildung deutlich dargestellt, besitzt jede Klasse eigene Operationen. An die Klassen „Controller" und „Ethernet" werden zusätzlich angelegte Events gesendet, welche dann von der jeweiligen Klasse empfangen und weiter verarbeitet werden. Außerdem ist den Klassen „Controller", „Ethernet" und „Joystick" ein Statechart hinterlegt, welches weitere Aufgaben und Funktionen übernimmt.

Im Folgenden ist das komplette UML-Model des OMDs Overview abgebildet.

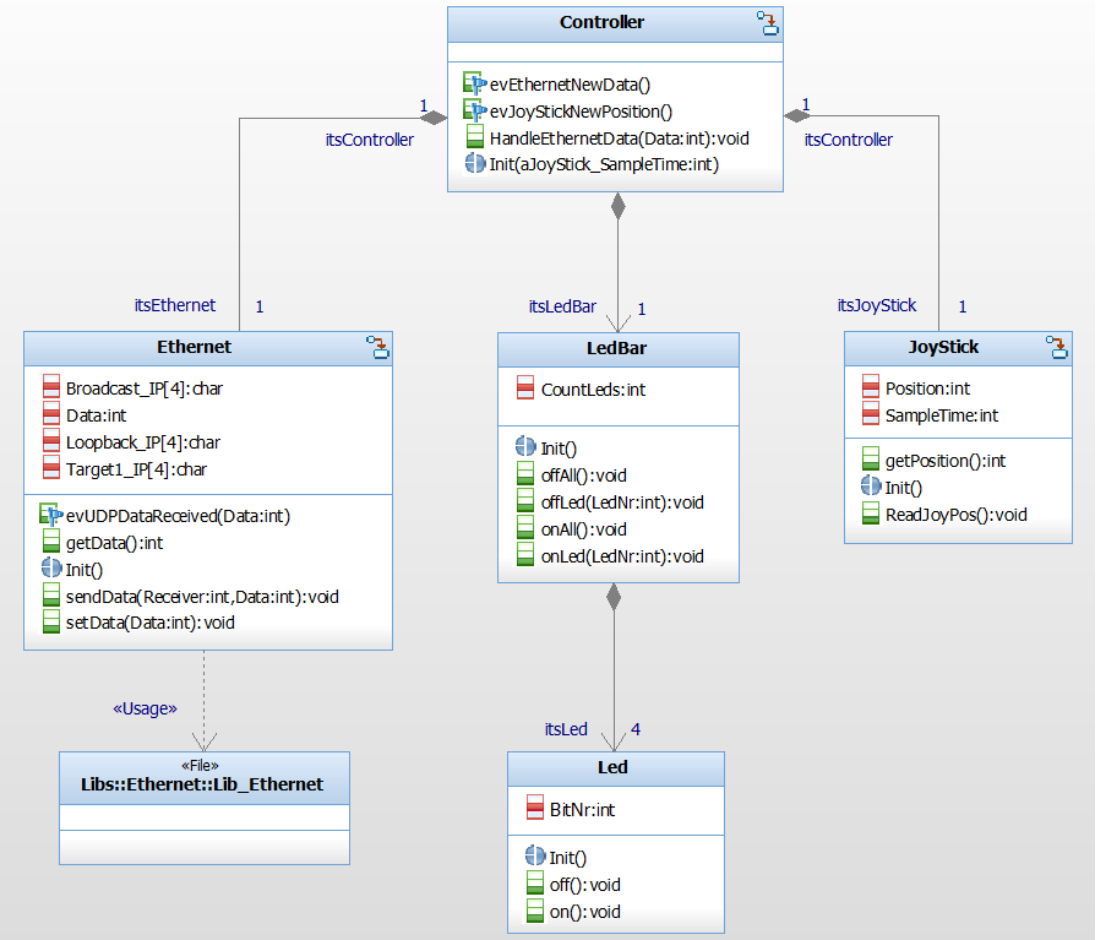


Abb. 54 OMD Overview

An der Struktur erkennt man, dass die Klasse „Controller" eine übergeordnete Klasse darstellt. Sie verwaltet die Objekte der „Ethernet"-, „Joystick"- und „LedBar"-Klasse. Die Klasse „LedBar" wiederum beinhaltet die Klasse „Led". Diese Struktur ermöglicht der Klasse „Controller", auf alle Operationen des UML-Modells zuzugreifen. Die Klasse „Ethernet" benutzt außerdem das File „Lib_Ethernet", in dem Funktionen für die Ethernet Kommunikation bereitgestellt werden. Dies wird in Abschnitt 5.2.5 näher erläutert.

Im Folgenden werden die Klassen und deren wichtigsten Funktionen vorgestellt und erklärt. In den Abbildungen der Klassen sind auf der linken Seite jeweils die Klassendiagramme und auf der rechten Seite jeweils die Struktur im Modell Browser dargestellt.

5.2.3.1 Klasse Controller

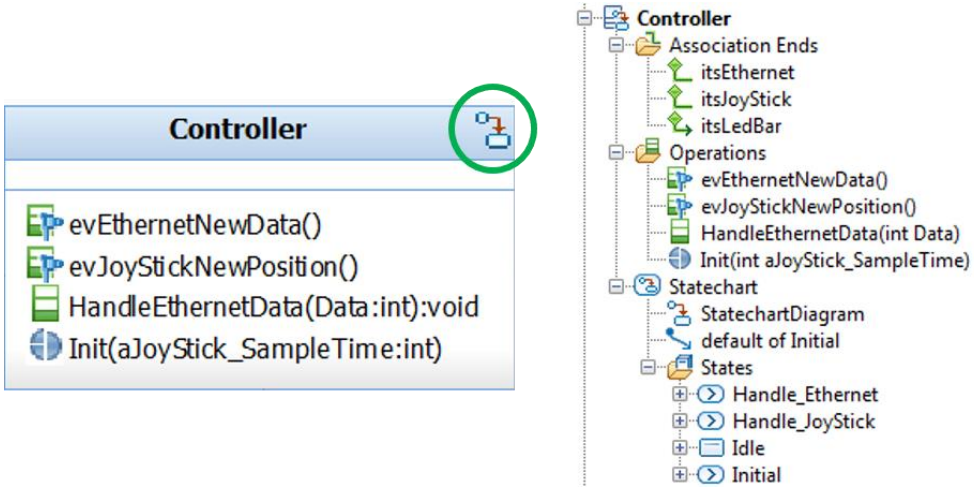


Abb. 55 Klasse Controller

In Abb. 55 ist die „Controller" Klasse jeweils als Klassendiagramm und als Struktur im Modell Browser dargestellt. Wie bereits erwähnt, ist die Klasse „Controller" die zentrale Klasse des UML-Modells. Die Klassen „Ethernet" und „JoyStick" teilen der Klasse „Controller" durch die Events „evEthernetNewData()" und „evJoyStickNewPosition()" mit, wenn Daten empfangen wurden oder sich die Position des Joysticks geändert hat. Zudem besitzt die Klasse „Controller" die Operation „Init (aJoyStick_SampleTime:int)" und „HandleEthernetData(Data:int):void".
Die Methode „HandleEthernetData(Data:int):void" befasst sich mit der Auswertung der empfangenen Daten und ist unabhängig von der Empfangsroutine per UDP. Sie stellt eine beispielhafte Verarbeitung der empfangenen Daten dar und kann auch anderweitig realisiert werden.
Außerdem ist der Klasse „Controller" ein Statechart hinterlegt, welches nicht nur im Modell Browser ersichtlich ist, sondern auch durch das Symbol (grüner Kreis) im Klassendiagramm angezeigt wird.

Im Folgenden werden das Zustandsdiagramm und die Operation „HandleEthernetData(Data:int):void" genauer betrachtet und beschrieben.

Zustandsdiagramm Klasse Controller

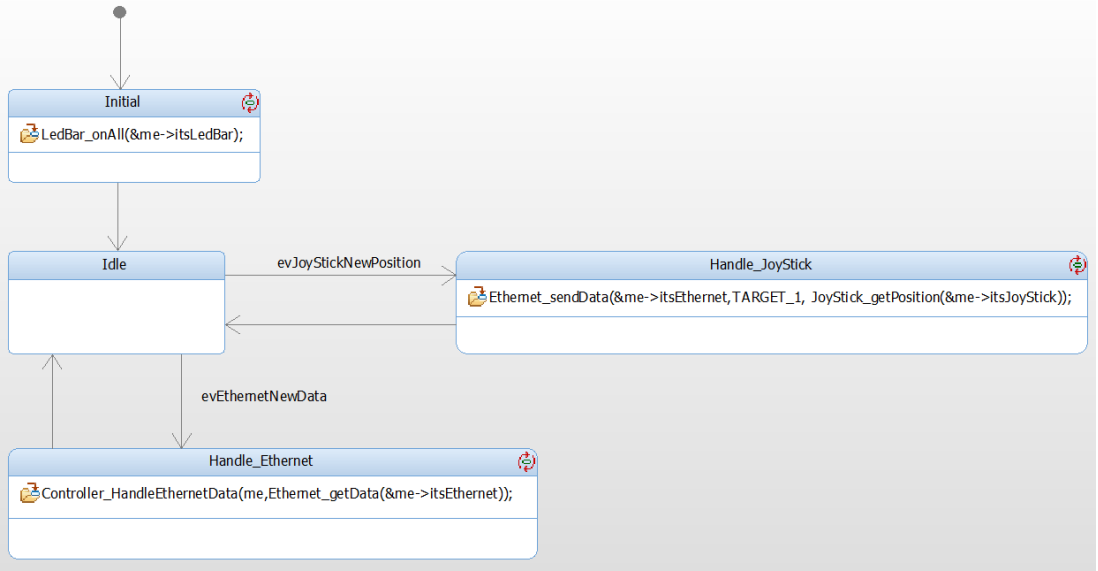


Abb. 56 Zustandsdiagramm Klasse Controller

In Abb. 56 ist das Statechart der Klasse „Controller" dargestellt. Zunächst werden im State „Initial" alle vier Leds der LedBar eingeschaltet, bevor in den Zustand „Idle" gewechselt wird. Dort verharrt das Zustandsdiagramm bis ein Ereignis auftritt. Wenn das Event „evJoyStickNewPosition()" auftritt, wird in den State „Handle_JoyStick" gewechselt. Hier wird die Operation „sendData()" der Klasse „Ethernet" aufgerufen. Zusätzlich werden die Parameter „TARGET_1" für den Empfänger und der Rückgabewert (Integer) der Funktion „getPosition()" der Klasse „Joystick" für die aktuelle Joystick Position übergeben. Danach wird automatisch wieder zurück in den Zustand „Idle" gesprungen.

Falls das Event „evEthernetNewData()" auftritt, wechselt das Zustandsdiagramm in den State „Handle_Ethernet". Dort wird die Operation „HandleEthernetData()" der Klasse „Controller" ausgeführt. Als Parameter wird der Rückgabewert (Integer) der Funktion „getData()" der Klasse „Ethernet" übergeben. Anschließend wird wieder zurück in den State „Idle" gewechselt und auf das nächste Ereignis gewartet.

HandleEthernetData(Data:int):void

Im Folgenden ist die Implementierung der Operation „HandleEthernetData(Data:int):void" dargestellt.

```
1   // Über Ethernet empfangene Daten,
2   // welche Joystickposition enthalten, auswerten
3   switch(Data){
4       case JoyStick_LEFT:{
5           // Alle Leds der LedBar ausschalten
6           LedBar_offAll(&me->itsLedBar);
7           // Led 1 einschalten
8           LedBar_onLed(&me->itsLedBar, Led_1);
9       break;
10      }
11      case JoyStick_UP:{
12          // Alle Leds der LedBar ausschalten
13          LedBar_offAll(&me->itsLedBar);
14          // Led 2 einschalten
15          LedBar_onLed(&me->itsLedBar, Led_2);
16          break;
17      }
18      case JoyStick_RIGHT:{
19          // Alle Leds der LedBar ausschalten
20          LedBar_offAll(&me->itsLedBar);
21          // Led 3 einschalten
22          LedBar_onLed(&me->itsLedBar, Led_3);
23      break;
24      }
25      case JoyStick_DOWN :{
26          // Alle Leds der LedBar ausschalten
27          LedBar_offAll(&me->itsLedBar);
28          // Led 4 einschalten
29          LedBar_onLed(&me->itsLedBar, Led_4);
30      break;
31      }
32      case JoyStick_SELECT :{
33          // Alle Leds der LedBar ausschalten
34          LedBar_offAll(&me->itsLedBar);
35          // Alle Leds einschalten
36          LedBar_onAll(&me->itsLedBar);
37      break;
38      }
39      default:
40          break;
41  }
```

Die Methode ist aus einer Switch-Case Abfrage aufgebaut, welche den übergebenen Parameter ausliest. Es wird überprüft, welche Position des Joysticks über Ethernet gesendet wurde. Wenn beispielsweise der Übergabeparameter mit dem Item „JoyStick_LEFT" (Codezeile 4) der Konstanten „DIRECTION" übereinstimmt, werden zunächst alle Leds ausgeschaltet (Codezeile 6; Funktion „offAll" der Klasse „LedBar" wird aufgerufen) und dann die erste Led eingeschaltet (Codezeile 8; Funktion „onLed" der Klasse „LedBar" wird aufgerufen und als Parameter das Item

„Led_1" der Konstanten „LEDS" für die erste Led übergeben). Falls der Joystick nach oben, rechts oder unten gedrückt wird, werden die gleichen Funktionen aufgerufen, allerdings wird jeweils eine andere Led eingeschaltet. Wird auf den Joystick gedrückt, werden alle Leds der „LedBar" eingeschaltet (Codezeile 36; Funktion „onAll" der Klasse „LedBar" wird ausgeführt).

5.2.3.2 Klasse Ethernet

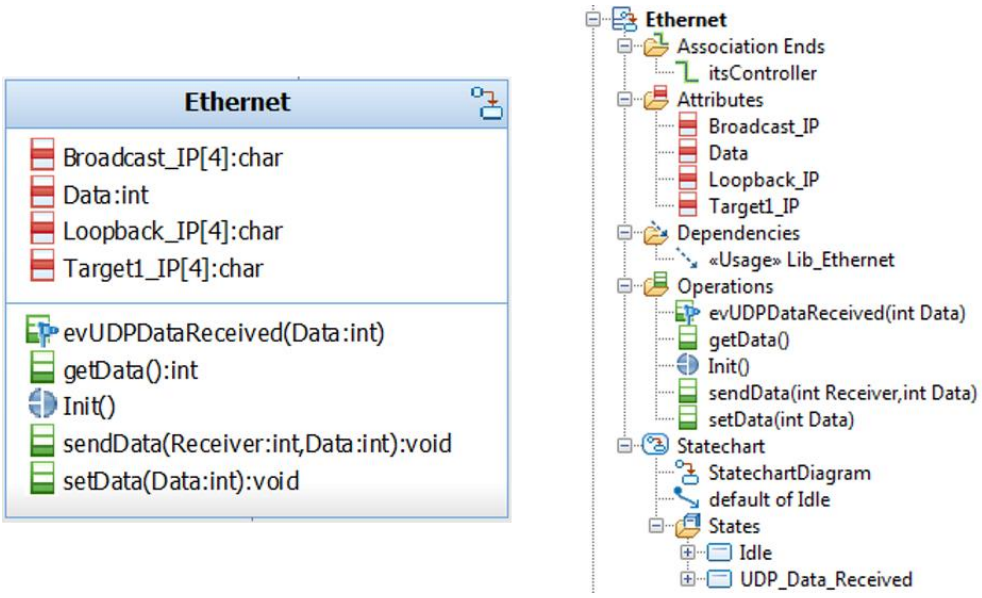


Abb. 57 Klasse Ethernet

In Abb. 57 ist die Klasse „Ethernet" mit Klassendiagramm und Struktur im Modell Browser dargestellt. Diese Klasse hat vier Attribute, davon sind drei Charakter-Arrays mit vier Feldern („Broadcast_IP", „Loopback_IP", „Target1_IP") und eines eine normale Integer-Variable („Data"). Die Klasse „Ethernet" empfängt das Event „evUDPDataReceived(Data:int)" und besitzt vier Operationen. Diese setzen sich aus einer „Init()" Operation und aus den Methoden „getData():int", „sendData(Receiver:int,Data:int):void" und „setData(Data:int):void" zusammen. Ebenso, wie der Klasse „Controller", ist der Klasse „Ethernet" ein Zustandsdiagramm hinterlegt. Die im OMD Overview erkennbare Dependency „Usage" ist ebenfalls im Modell Browser erkennbar.

Im Folgenden werden neben dem Statechart die Operationen „Init()" und „sendData(Receiver:int,Data:int):void" vorgestellt.

Zustandsdiagramm Klasse Ethernet

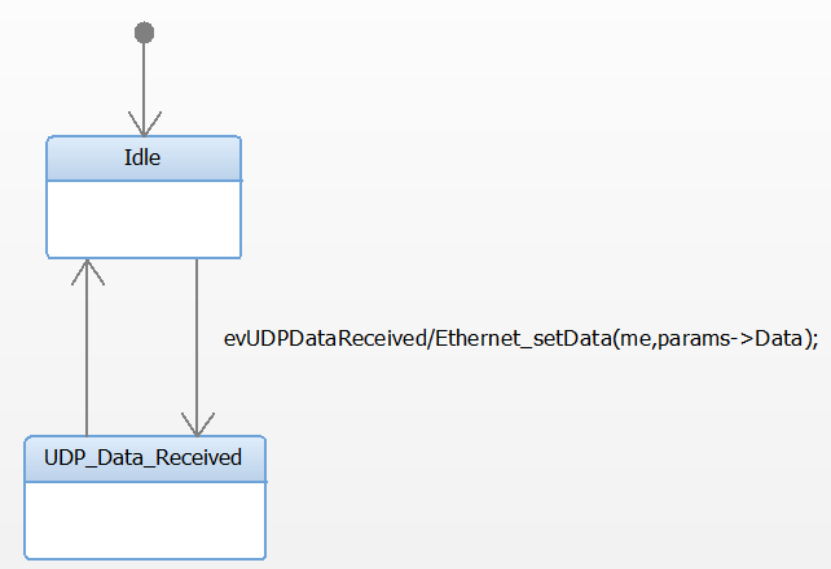


Abb. 58 Zustandsdiagramm Klasse Ethernet

In Abb. 58 ist das Statechart der Klasse „Ethernet" abgebildet. Diese besteht aus zwei leeren Zuständen. Nach Beginn wird im State „Idle" verharrt, bis das Event „evUDPDataReceived(Data:int)" auftritt. Nach Eintreten dieses Ereignisses wird die Funktion „setData()" der Klasse „Ethernet" aufgerufen. Ebenso wird dieser Methode als Parameter der Parameter des Events (Data:int), welcher die empfangenen Daten beinhaltet, übergeben. Danach wird in den State „UDP_Data_Received" gewechselt und anschließend wieder in den State „Idle" zurück gesprungen.

Init()

Mit dem folgenden Code wird die Implementierung der Funktion „Init()" aufgezeigt. Die verwendeten IP-Adressen können durch den Benutzer frei gewählt werden.

```
1  // Anlegen des t_tcp_main Task mit niedrieger Prio
2  // und void me-Pointer als Übergabeparamter
3  os_tsk_create_ex ( t_tcp_main, WST_RTOS_TASK_PRIORITY_LOW,(void*)me );
4
```

```
5   // Broadcast_IP = 192.168.0.255
6   me->Broadcast_IP[0] = 192;
7   me->Broadcast_IP[1] = 168;
8   me->Broadcast_IP[2] = 0;
9   me->Broadcast_IP[3] = 255;
10
11  // Target1_IP = 192.168.0.110
12  me->Target1_IP[0] = 192;
13  me->Target1_IP[1] = 168;
14  me->Target1_IP[2] = 0;
15  me->Target1_IP[3] = 110;
16
17  // Loopback_IP = 127.0.0.1
18  me->Loopback_IP[0] = 127;
19  me->Loopback_IP[1] = 0;
20  me->Loopback_IP[2] = 0;
21  me->Loopback_IP[3] = 1;
```

In dieser „Init()" Methode wird zunächst der „t_tcp_main" Task angelegt (Codezeile 3). Dieser Task hat eine niedrige Priorität („WST_RTOS_TASK _PRIORITY_LOW") und enthält als Übergabeparameter einen void me-Pointer, welcher auf die Klasse „Ethernet" zeigt. Die genaue Funktion des Tasks wird in Abschnitt 5.2.5 erläutert. Außerdem werden in der „Init()" Operation die drei Attribute mit den Charakter-Arrays initialisiert. So erhält beispielsweise das Attribut „Target1_IP" die IP-Adresse „192.168.0.110" (Codezeilen 6-9). Ebenso werden den Attributen „Broadcast_IP" und „Loopback_IP" ihre IP-Adressen zugewiesen.

sendData(Receiver:int, Data:int):void

Im Folgenden findet sich die Implementierung der Methode "sendData(Receiver:int,Data:int):void".

```
1   // Auswahl des Empfängers
2   switch(Receiver){
3       case BROADCAST:{
4           // Aufruf der Funktion aus Lib_Ethernet
5           UDP_send_data_joystick(me->Broadcast_IP,Data);
6           break;
7       }
8       case LOOPBACK:{
9           // Aufruf der Funktion aus Lib_Ethernet
10          UDP_send_data_joystick(me->Loopback_IP,Data);
11          break;
12      }
13      case TARGET_1:{
14          // Aufruf der Funktion aus Lib_Ethernet
15          UDP_send_data_joystick(me->Target1_IP,Data);
16          break;
17      }
18      default:
19          break;
20  }
```

Diese Methode besteht aus einer Switch-Case Abfrage. Darin wird der Übergabeparameter, welcher die Information über den Empfänger (Receiver) enthält, ausgelesen. Für die einfache Ermittlung des Empfängers wurde die Konstante „USE_IP" angelegt. Die Items der Konstante können mit dem Übergabeparameter „Receiver" verglichen werden. Wenn nun beispielsweise der Übergabeparameter der Operation „Receiver" mit „TARGET_1" übereinstimmt (Codezeile 13), so wird die Funktion „UDP_send_data_joystick(me->Target1_IP,Data)" (Codezeile 15) aus der Library „Lib_Ethernet" aufgerufen. Als Parameter werden dieser Funktion die Empfänger IP (Target1_IP) und Daten über die Joystick Position (Data) übergeben. Der Ablauf bei den restlichen Switch-Case Abfragen ist identisch; es ändert sich lediglich der Empfänger (Loopback_IP oder Broadcast_IP).

5.2.3.3 Klasse Joystick

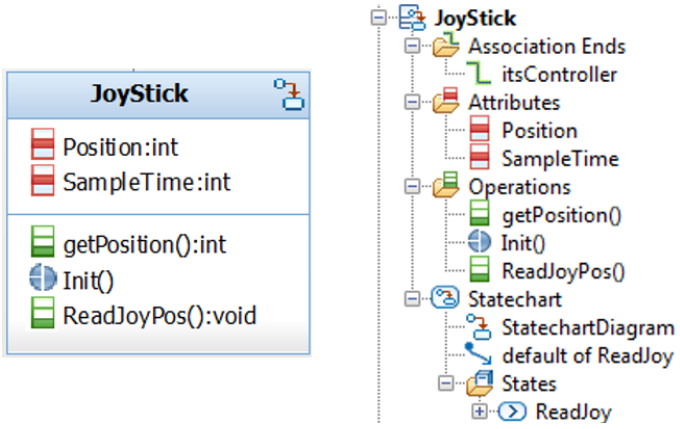


Abb. 59 Klasse JoyStick

In Abb. 59 ist die Klasse „JoyStick" als Klassendiagramm sowie deren Struktur im Modell Browser dargestellt. Die Klasse besitzt die beiden Attribute „Position" und „SampleTime" als einfache Integer-Variablen, aber auch die drei Operationen „getPosition():int", „Init()" und „ReadJoyPos():void". Zudem ist der Klasse „JoyStick", genauso wie den Klassen „Controller" und „Ethernet", ein Statechart hinterlegt.

Im Folgenden werden das Zustandsdiagramm und die Methode „ReadJoyPos():void" genauer betrachtet.

Zustandsdiagramm Klasse JoyStick

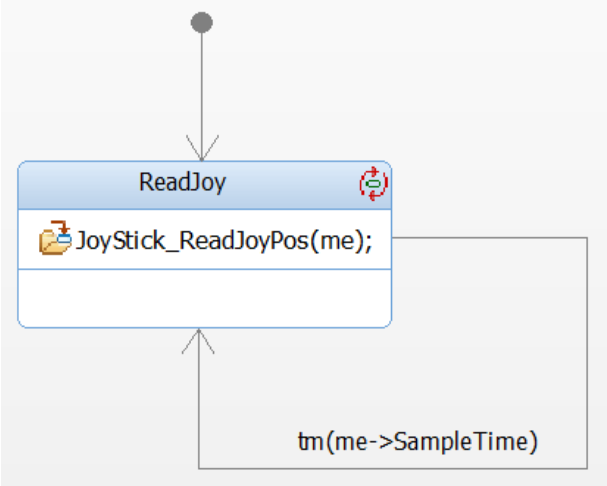


Abb. 60 Zustandsdiagramm Klasse JoyStick

In Abb. 60 ist das Statechart der Klasse „JoyStick" dargestellt. Dabei handelt es sich um ein einfaches Zustandsdiagramm mit nur einem State „ReadJoy", welcher zyklisch mit dem durch die Variable „SampleTime" eingestellten Wert aufgerufen wird. Somit wird auch die Funktion „ReadJoyPos()" der Klasse „JoyStick" zyklisch ausgeführt.

ReadJoyPos():void

Die Implementierung der Operation „ReadJoyPos():void" wird anhand des folgenden Codes dargestellt.

```
1   // Längere SampleTime
2   int SampleWaitTime = 1000; // in ms
3
4   // Joystickposition auslesen
5   int position =(LPC_GPIO1->FIOPIN >> 20) & Joystick_Mask;
6
7   // Alle Positionen außer Center erfassen
8   if(position != JoyStick_CENTER)
9   {
10      me->Position = position;
11      // Event JoyStickNewPosition feuern
12      CGEN (me->itsController, evJoyStickNewPosition() );
13      // Pause im Sample um mehrfaches Senden von Daten zu verhindern
14      me->SampleTime = SampleWaitTime;
15  }
16
```

```
17  else if(me->SampleTime == SampleWaitTime)
18  {
19      // SampleTime wieder zurück auf 200 setzen
20      me->SampleTime = 200; // in ms
21  }
```

Zu Beginn wird eine lokale Variable „SampleWaitTime" mit dem Wert 1000 angelegt (Codezeile 2). Danach wird die Joystickposition ausgelesen und in eine lokale Variable „position" gespeichert (Codezeile 5). Daraufhin folgt eine Überprüfung, ob die ausgelesene Joystickposition ungleich der Ruhestellung des Joysticks ist (Codezeile 8). Falls ja, wird zunächst die aktuelle Joystickposition in das Attribut „Position" gespeichert, dann ein Event „evJoyStickNewPosition()" an die Klasse „Controller" gefeuert und schließlich der Wert des Attributs „SampleTime" auf 1000 (SampleWaitTime) gesetzt (Codezeile 10-14). Die Erhöhung der Samplezeit auf den Wert 1000 verhindert ein mehrfaches Senden bei längerer Betätigung des Joysticks (so wird nur einmal anstatt fünfmal pro Sekunde gesendet). Durch eine anschließende if-else-Abfrage wird abgefragt, ob der Wert des Attributs „SampleTime" gleich 1000 ist (Codezeile 17). Ist dies der Fall, wird der Wert des Attributs „SampleTime" auf 200 zurückgesetzt (Codezeile 20).

5.2.3.4 Klasse LedBar

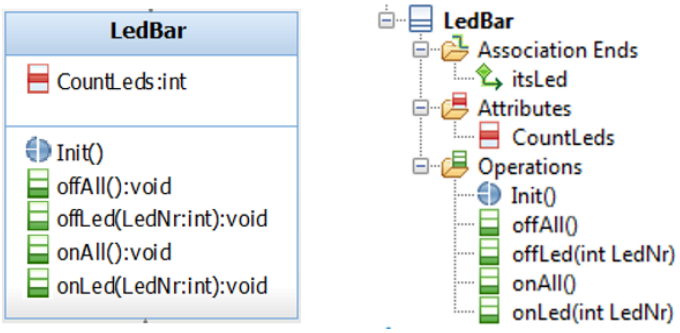


Abb. 61 Klasse LedBar

In Abb. 61 ist die Klasse „LedBar" mit Klassendiagramm und Struktur im Modell Browser abgebildet. Sie hat ein Attribut „CountLeds" als einfache Integer-Variable, in welcher die Anzahl der vorhandenen Leds abgelegt ist. Zudem besitzt die Klasse „LedBar" fünf Operationen. Mit „Init()"

werden die Bitnummern der Leds gesetzt und der Port GPIO2 als Ausgang deklariert. Mit „offAll():void" werden alle vier Leds ausgeschaltet und mit „offLed(LedNr:int):void" wird die durch den Übergabeparameter „LedNr" bestimmte Led ausgeschaltet. Analog zu den angeführten Funktionen können die Leds entweder gemeinsam oder einzeln eingeschaltet werden.

5.2.3.5 Klasse Led

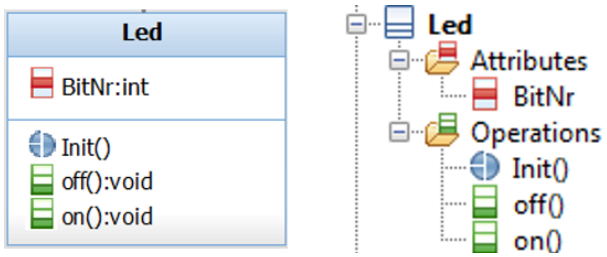


Abb. 62 Klasse Led

In Abb. 62 wird die Klasse „Led" mit Klassendiagramm und Modell Browser Struktur aufgezeigt. Sie hat ein Attribut „BitNr" als Integer-Variable, mit welcher die Led ausgewählt wird, die ein- oder ausgeschaltet werden soll. Zudem besitzt die Klasse „Led" drei Operationen. Neben der Operation „Init()", mit der die Leds initialisiert werden, handelt es sich um die Methoden „off():void" und „on():void, mit welchen die Led aus- bzw. eingeschaltet wird. Somit handelt es sich bei der Klasse „Led" um eine sehr übersichtliche Klasse.

5.2.4 OMD Runtime

Folgende Abbildung zeigt das OMD Runtime.

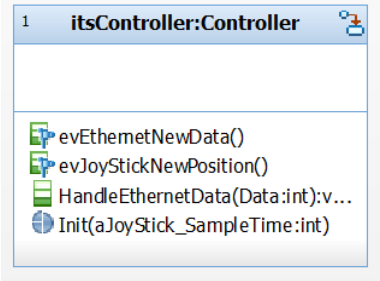


Abb. 63 OMD Runtime

Hier wird nur ein Objekt der Klasse „Controller" angelegt. Diese einfache Struktur erreicht man dadurch, dass die Klasse „Controller" die übergeordnete Klasse des gesamten UML-Modells bildet und deshalb alle Objekte der anderen Klassen verwaltet. Daher muss auch nur ein Objekt der Klasse „Controller" angelegt werden, welches man an der Bezeichnung „itsController:Controller" erkennt. Dies bedeutet, dass ein Objekt mit dem Namen „itsController" von der Klasse „Controller" vorliegt.

5.2.5 Einbinden und Funktion der Lib_Ethernet Library

Für die Ethernet Kommunikation werden spezielle Funktionen zum Senden und Empfangen der Daten benötigt. Diese werden durch die „Lib_Ethernet" Library zur Verfügung gestellt. Um sie dem Projekt in Rhapsody hinzuzufügen, müssen folgende Schritte ausgeführt werden.

- Rechtsklick auf „Packages" und Hinzufügen eines neuen Packages „Libs"
- Erstellen eines neuen Packages Ethernet im Package „Libs" und Hinzufügen eines Files mit dem Namen „Lib_Ethernet"

Der Modell Browser muss nun wie folgt aussehen.

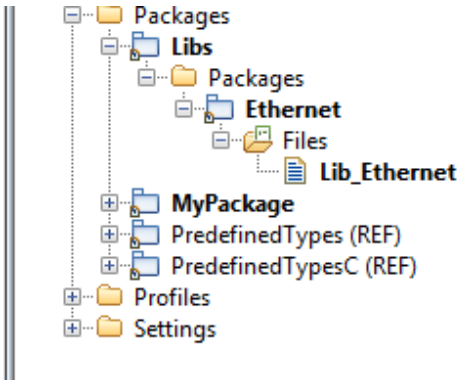


Abb. 64 Modell Browser Lib

Um die Funktionen nutzen zu können, muss das File in das OMD Overview gezogen und mit einer Dependency mit der Klasse Ethernet

verbunden werden. Anschließend ist der Stereotyp auf „Usage" zu setzen, was sich wie folgt darstellt.

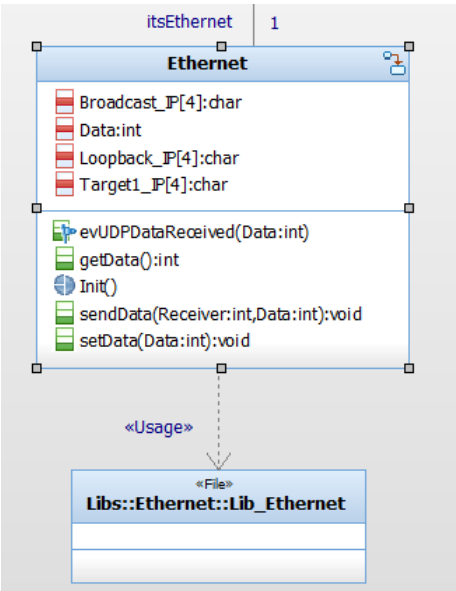


Abb. 65 Usage Lib

Ist dies abgeschlossen, müssen die „Lib_Ethernet.h" und „Lib_Ethernet.c" Dateien, welche sich im Ordner „Ethernet_Lib" auf dem Desktop befinden, in das Debug Verzeichnis des Rhapsody-Projektes eingefügt und ersetzt werden. Alternativ können zuvor auch die „Lib_Ethernet" Dateien im Debug Verzeichnis gelöscht werden. Die Library ermöglicht die Verwendung nachfolgender Funktion.

```
void UDP_send_data_joystick (char Target_IP[4], int Data).
```

Als Übergabeparameter müssen die Ziel IP-Adresse, welche durch die Attribute der Klasse Ethernet wiedergegeben wird, und die Daten übergeben werden. Sie enthalten Information über die Position des Joysticks. Folgende Daten können übermittelt werden (Items der Konstante „DIRECTION").

- JoyStick_LEFT
- JoyStick_RIGHT
- JoyStick_UP
- JoyStick_DOWN
- JoyStick_SELECT

Der Realisierung der Funktion sieht folgendermaßen aus.

```
1  void UDP_send_data_joystick (char Target_IP[4], int Data){
2
3    U8 *sendbuf;
4    U8 IP_send[4];
5
6    // Ziel IP Adresse
7    IP_send[0] = (Target_IP[0]);
8    IP_send[1] = (Target_IP[1]);
9    IP_send[2] = (Target_IP[2]);
10   IP_send[3] = (Target_IP[3]);
11 /*-------------------------------------------------------------------
12   Speicher (1 Byte) für das zu sendende Datenpacket allokieren
13 -------------------------------------------------------------------
14   U8* udp_get_buf (
15     U16 size);    Number of bytes to be sent
16   Return Value: Pointer auf allokierten Speicher
17 -------------------------------------------------------------------*/
18   sendbuf = udp_get_buf (1);
19 /*-------------------------------------------------------------------
20    Anhand von Data, welche die Position des Joysticks enthält,
21    werden die zu sendenden Daten ermittelt.
22    Dies entspricht einem Charakter (1 Byte) für die entsprechende
23    Position:
24       'L' = Links
25       'R' = Rechts
26       'U' = Oben
27       'D' = Unten
28       'S' = Select
29 -------------------------------------------------------------------*/
30   switch(Data){
31    case JoyStick_LEFT:{
32     sendbuf[0] = 'L';
33    break;
34    }
35    case JoyStick_RIGHT:{
36     sendbuf[0] = 'R';
37    break;
38    }
39    case JoyStick_UP:{
40     sendbuf[0] = 'U';
41    break;
42    }
43    case JoyStick_DOWN:{
44     sendbuf[0] = 'D';
45    break;
46    }
47    case JoyStick_SELECT:{
48     sendbuf[0] = 'S';
49    break;
50    }
51    default:
52     break;
53   }
54 /*-------------------------------------------------------------------
55  Daten senden via UDP
56 -------------------------------------------------------------------
57  udp_send (
58   U8  socket,     UDP socket to send the data packet from
59   U8* remip,      Pointer to the IP address of the remote machine
60   U16 remport,    Port number of remote machine to send the data to
61   U8* buf,        Pointer to buffer containing the data to send
62   U16 dlen );     Number of bytes of data to send
63 -------------------------------------------------------------------*/
64     udp_send (socket_udp, IP_send, PORT_NUM_send, sendbuf, SENDLEN);
65 }
```

Zu Beginn wird die Ziel IP-Adresse in das Array „IP_send[4]" übertragen (Codezeile 7-10), da diese im Datenformat „U8*" vorliegen muss. Anschließend wird Speicher für die zu sendenden Daten allokiert (Codezeile 18). Seine Größe beträgt lediglich ein Byte, da ein Charakter für die jeweilige Position über UDP übertragen wird (Codezeile 30-53). Abschließend werden die Daten mit dem Aufruf der Funktion „udp_send" (Codezeile 64) gesendet.
Des Weiteren ermöglicht die Library das Anlegen des folgenden Tasks.

```
__task void t_tcp_main (void* void_me)
```

Als Übergabeparameter muss ein void-Pointer auf die Ethernetklasse aus dem UML-Modell übergeben werden. Dadurch ist es möglich, Events beim Empfang von Daten aus der Library „Lib_Ethernet" an die Klasse „Ethernet" zu feuern. Dieses wird durch das Makro „CGEN" erreicht und ist in der Funktion „procrec" in der Library folgendermaßen realisiert.

```
CGEN(me, evUDPDataReceived (JoyStick_LEFT));
```

Durch den Befehl wird ein Event „evUDPDataReceived" an die Klasse „Ethernet" gefeuert, welches als Parameter Information über die Position des Joysticks (empfangene Daten) hat. Voraussetzung dafür ist das Anlegen des Events im Rhapsody-UML Modell.

Die Realisierung des vorher erwähnten Tasks stellt sich wie folgt dar.

```
1  __task void t_tcp_main (void* void_me) {
2
3   /* cast void-Pointer void_me zu Ethernet-Pointer */
4   me = (Ethernet *)void_me;
5
6   /* Initialize the TcpNet */
7   init_TcpNet ();
8
9  /*---------------------------------------------------------------------
10  Initialize UDP Socket and start listening
11 ---------------------------------------------------------------------
12  U8 udp_get_socket (
13     U8   tos,           Type Of Service
14     U8   opt,           Option to calculate or verify the checksum
15     U16 (*listener));   Function to call when TCPnet receives a data
16                         packet
17 ---------------------------------------------------------------------*/
18    socket_udp = udp_get_socket (0, UDP_OPT_SEND_CS | UDP_OPT_CHK_CS,
19 udp_callback);
20
21    if (socket_udp != 0) {
22      /*-----------------------------------------------------------------
23       Open UDP port PORT_NUM_empf for communication
24      -----------------------------------------------------------------
25      udp_open (
26      U8  socket,      Socket handle to use for communication
27      U16 locport);    Local port to use for communication
28      -----------------------------------------------------------------*/
29      udp_open (socket_udp, PORT_NUM_empf);
30    }
31
32    while (1) {
33      /* Run main TcpNet 'thread' */
34      main_TcpNet();
35    }
36 }
```

In Codezeile 7 wird das TcpNet initialisiert. Anschließend wird ein UDP Socket angelegt und geöffnet (Codezeile 18-29). Beim Anlegen des Sockets wird die Callback-Funktion, welche beim Empfang von Daten aufgerufen wird, als Parameter übergeben (Codezeile 18,19). Die Hauptaufgabe des Tasks ist es, die „main_TcpNet" Funktion zyklisch auszuführen, sodass das TcpNet System funktioniert (Codezeilen 32-34).
Die wichtigsten Funktionalitäten der Library sind nun vorgestellt. Um tiefere Einblicke zu erhalten, kann die „Lib_Ethernet.c" Datei in Keil µVision geöffnet oder in Anhang 8.3 eingesehen werden. Alle wichtigen Codezeilen sind ausführlich kommentiert.

Im weiteren Verlauf wird die Deploy-Konfiguration vorgestellt.

5.2.6 Deploy-Konfiguration

Wird der Deployer zum ersten Mal gestartet, erscheint folgendes Fenster.

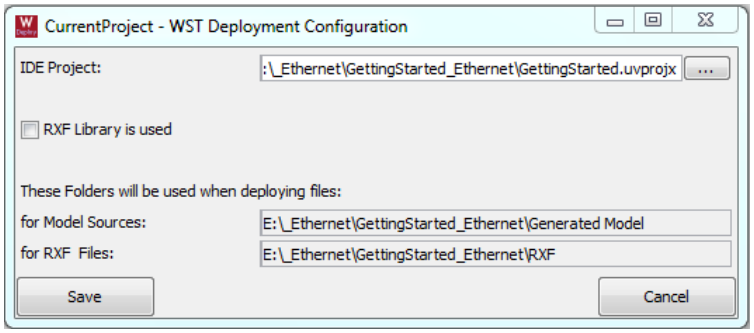


Abb. 66 Deployer Konfiguration Fenster Ethernet

Auf dem Desktop der virtuellen Maschine wird ein Ordner „Ethernet_Lib" bereitgestellt. In diesem befinden sich die „Lib_Ethernet.h" und „Lib_Ethernet.c", sowie ein weiterer Ordner „GettingStartedEthernet". Der Ordner „GettingStartedEthernet" beinhaltet ein Keil µVision 5 Projekt. Die Struktur des µVision Projektes ist speziell an die Verwendung der Ethernet Schnittstelle angepasst und wurde im Rahmen dieser Studienarbeit erstellt. Dem Keil µVision 5 Projekt wurden Dateien aus früher durchgeführten Studienarbeiten hinzugefügt, welche die Konfiguration der Ethernet Schnittstelle des MCB1700 vornehmen. Daher werden diese Dateien nicht genauer vorgestellt. Der Ordner „GettingStartedEthernet" sollte in das Rhapsody Projektverzeichnis kopiert werden, sodass das Original auf dem Desktop stets als leeres Startprojekt verwendet werden kann.

Im Deploy Konfigurations Fenster (Abb. 66) kann über den Button „Browser" das zuvor in das Rhapsody Projektverzeichnis kopierte Keil µVision 5 Projekt gewählt werden. Die Checkbox „RXF Library is used" darf nicht gesetzt werden. Abschließend werden die Einstellungen durch das Betätigen des Buttons „Save" übernommen.

Die Deploy Konfigurationen können nachträglich nochmals geändert werden. Dafür steht ein entsprechender Eintrag im Untermenü Tools in Rhapsody zur Verfügung.

5.2.7 Festlegung eigener IP-Adresse

In Abschnitt 5.2.3.2 wurde die durch das Attribut „Target1_IP" die IP Adresse festgelegt, an welche die Daten gesendet werden.

Bevor man Daten empfangen kann, muss die eigene IP Adresse festgelegt werden. Dies geschieht nach dem Deploy Vorgang direkt in Keil µVision. Dafür wird, wie in Abb. 67 zu sehen ist, die „Net_Config.c" Datei geöffnet und auf den Configuration Wizard umgeschaltet. Anschließend kann man die eigene IP Adresse (rot umrahmt) festlegen. In untenstehender Darstellung ist die IP Adresse 192.168.0.110 definiert.

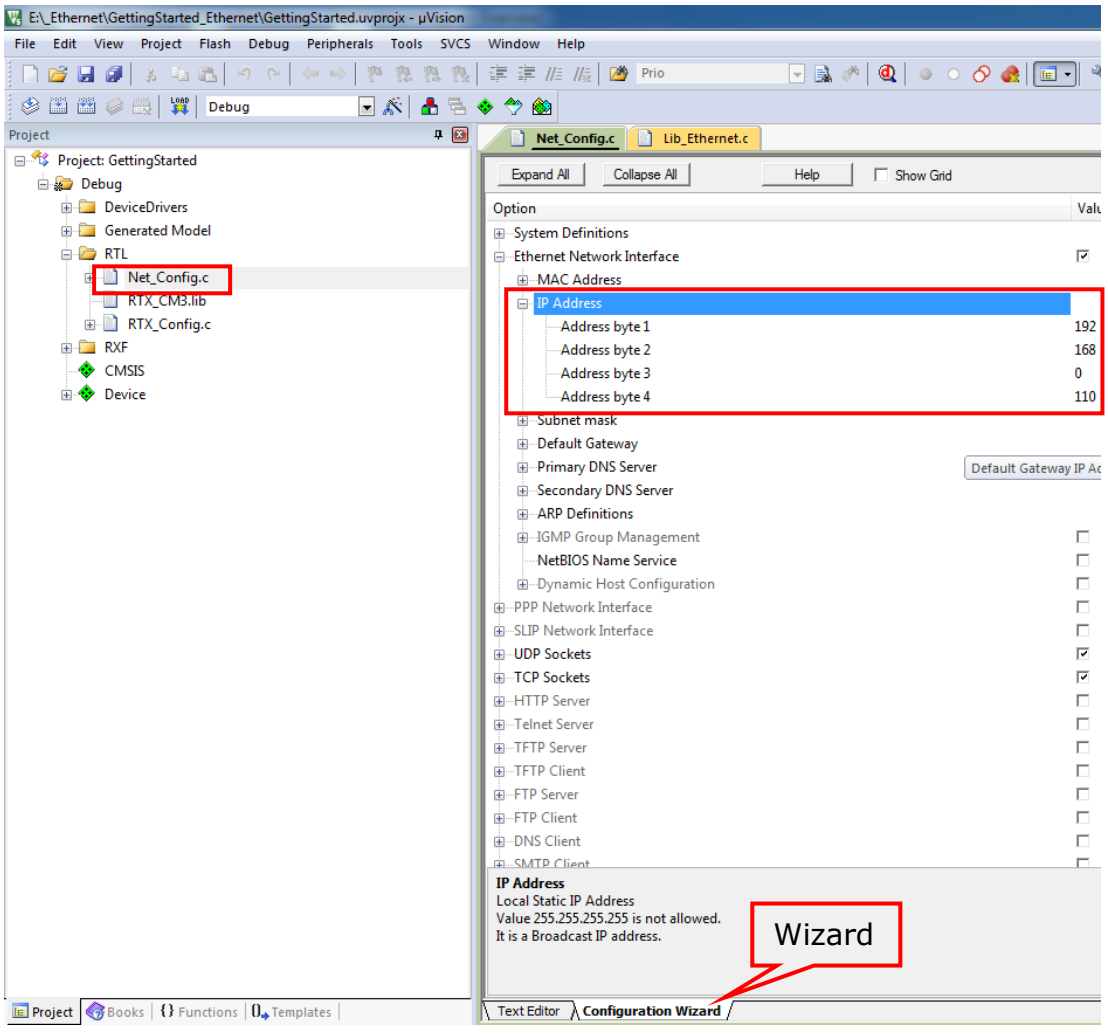


Abb. 67 Einstellung IP Adresse

5.3 Fazit

In diesem Kapitel wurde beschrieben, wie man die Ethernet Schnittstelle des MCB1700 Boards in einem UML-Modell in Rhapsody verwenden kann. Durch das Hinzufügen der Library „Lib_Ethernet" ist eine einfache Realisierung möglich. Zudem wurde ein Keil µVision Projekt angelegt, welches entsprechende Dateien zur Konfiguration des Ethernet Controllers des NXP LPC1768 enthält.

Ein weiterer wichtiger Geschichtspunkt war die Verwendung einer anderen Software von Willert. Diese hat den Einsatz des Keil RTX in Verbindung mit dem Embedded RXF von Willert ermöglicht. Das RTX von Keil unterstützt im Gegensatz zum OO RTX von Willert, welches bei der Modellierung von Aktivitätsdiagrammen eingesetzt wurde, die Kommunikation via TCP/IP. Ohne die Bereitstellung der Software durch das Unternehmen Willert wäre die Umsetzung der Aufgabe nicht gelungen.

Aufgabe war es, eine Ethernet Kommunikation zwischen zwei MCB1700 Boards herzustellen, sowie die jeweilige Position des Joysticks an das andere Board zu versenden und sie optisch auszugeben.

Diese Funktionalität konnte mittels des in 5.2.3 vorgestellten UML-Modells realisiert werden. Aufgrund der zentralen Klasse „Controller" handelt es sich um ein sehr übersichtliches und verständliches Modell. Alle mitwirkenden Komponenten sind als Klassen dargestellt und ihre Funktionalitäten als Zustandsdiagramme modelliert. Auch die Verwendung der „Lib_Ethernet" ist zu sehen.

Somit kann die Ethernet Schnittstelle des MCB Board 1700 eingebunden werden und die Laborübungen der Vorlesung Embedded Systems lassen sich um eine Ethernet Aufgabe erweitern.

6 Resümee und Ausblick

Ziel dieser Arbeit war zunächst die Codegenerierung aus UML-Aktivitätsdiagrammen in Rhapsody. Später erfolgte zudem die Einbindung der auf dem MCB1700 Board vorhandenen Ethernet Schnittstelle unter UML in Rhapsody.

Die Modellierung von Aktivitätsdiagrammen in Rhapsody lässt sich einfach und komfortabel durchführen. Zudem sind alle für die Modellierung wichtigen Notationselemente vorhanden. Das Übersetzen und Überspielen auf das MCB1700 Board erfolgt ebenso problemlos. Es wurde jedoch festgestellt, dass Aktivitätsdiagramme in Rhapsody nicht wie üblich tokenorientiert, sondern statebasiert übersetzt werden. Dies bedeutet, dass der generierte Code aus funktional gleichem Aktivitäts- und Zustandsdiagramm völlig gleich ist. Da aber in Aktivitätsdiagrammen nur Transitionen mit Guards benutzt und diese als Null-Transitionen angesehen werden (Transition hat kein Event entspricht Null-Transition), läuft das System nach der siebten nacheinander aufgetretenen Null-Transition in den Errorhandler. Somit sind Aktivitätsdiagramme, die nur mit Null-Transitionen modelliert sind, nicht ausführbar. Das Problem kann durch Einfügen von Zeitereignissen umgangen werden, welche Nicht-Null-Transitionen erzeugen. Da jedoch durch Aktivitätsdiagramme Systeme zeitkontinuierlich modelliert werden, wäre dies der falsche Ansatz, weswegen die Arbeiten an der Codegenerierung aus Aktivitätsdiagrammen an dieser Stelle gestoppt wurden.

Ändert sich die Codegenerierung von Rhapsody oder das Framework von Willert, so kann die Thematik der Aktivitätsdiagramme nochmals aufgegriffen werden.

Im zweiten Teil der Arbeit wurde die Einbindung der Ethernet Schnittstelle unter UML in Rhapsody betrachtet, um die Funktion zukünftig auch bei Laborübungen für die Vorlesung „Embedded Systems" anbieten zu können. Hierfür wird jedoch die Vollversion von „Keil RTX" benötigt; zudem ist die zusätzliche Installation des „Keil Legacy Package" notwendig, um eine Ethernet Kommunikation zwischen MCB1700 Boards aufzubauen. Durch Einbinden des Files „Lib_Ethernet" in das UML-Modell stehen benötigte Funktionen bereit, wie beispielsweise das Senden über UDP. Wenn auf einem Board der Joystick aus der Mitte bewegt wird, wird die Veränderung der Joystickposition registriert und die neue Position über Ethernet an das entfernte MCB1700 Board geschickt. Nach dem Empfang wird dort die neue Position ausgelesen und eine entsprechende Led eingeschaltet. Diese Kommunikation ist in beide Richtungen möglich, das heißt jedes Board dient als Sender und Empfänger.

Die Struktur des UML-Modells wurde so einfach wie möglich gehalten, sodass auch das Einbinden weiterer Funktionen keine großen Änderungen mit sich bringt. Das Modell stellt somit eine solide und grundlegende Basis dar, mit welcher weitere Funktionalitäten einfach zu realisieren sind. So kann in Zukunft auch das auf den MCB1700 Boards vorhandene LCD-Display in das UML-Modell mit aufgenommen werden. Dabei kann eine zusätzliche Klasse „Display" mit eingefügt werden, welche ebenso wie alle anderen Klassen von der zentralen Klasse „Controller" verwaltet wird. Die Aufnahme des Potentiometers in das UML-Modell erfolgt analog zu der eben beschriebenen Vorgehensweise.

Durch die vorliegende Studienarbeit wird ein Einblick in die Autocode-generation aus Aktivitätsdiagrammen gegeben und die Verwendung der Ethernet Schnittstelle des MCB1700 Boards mit einem UML-Modell ermöglicht. Die gewonnenen Erkenntnisse können für weitere Studien-arbeiten eingesetzt und weitergeführt werden.

7 Literaturverzeichnis

[1] Rupp, Chris; Hahn, Jürgen; Queins, Stefan; Jeckle, Mario; Zengler, Barbara: UML 2: Praxiswissen für die UML-Modellierung und –Zertifizierung. München: HANSER, 2005. ISBN 3-446-22952-3

[2] Matuschek, Marco: UML - Getting Started. Elektronisches Dokument, Willert Software Tools GmbH

[3] Entwicklung und Validierung von Embedded-Software mit einer Plattform für modellorientierte Entwicklung. URL: *http://www-03.ibm.com/software/products/de/ratirhap* [Link], Letzter Zugriff: 03.01.2015

[4] Embedded UML RXF. Elektronisches Dokument, Willert Software Tools GmbH

[5] Willert Embedded UML RXF. URL: *http://www.willert.de/produkte/-model-driven-sw-engineering/willert-embedded-uml-rxf/* [Link], Letzter Zugriff: 03.01.2015

[6] Matuschek, Macro: KEIL µVision IDE & MCB1700. Elektronisches Dokument, Willert Software Tools GmbH

[7] MDK-ARM Microcontroller Development Kit, URL: *http://www.keil.-com/arm/mdk.asp* [Link], Letzter Zugriff: 01.01.2015

[8] Willert RXF Documentation, Elektronische Dokumentation zur Software Rpy_C_OO RTX_Keil_ARM_MCB1700_Eval_TD_V6.01, Auffindbar in %Installationspfad%/Doc/Help/ RXF Documentation.hta

[9] Embedded OO RTX. Elektronisches Dokument, Willert Software Tools GmbH

[10] Neue RXF Releases. URL: *http://www.willert.de/neue-rxf-releases/* [Link], Letzter Zugriff: 22.12.2014

[11] RL-RTX. URL: *http://www.keil.com/support/man/docs/rlarm/-rlarm_ar_artxarm.htm* [Link], Letzter Zugriff: 19.02.2015

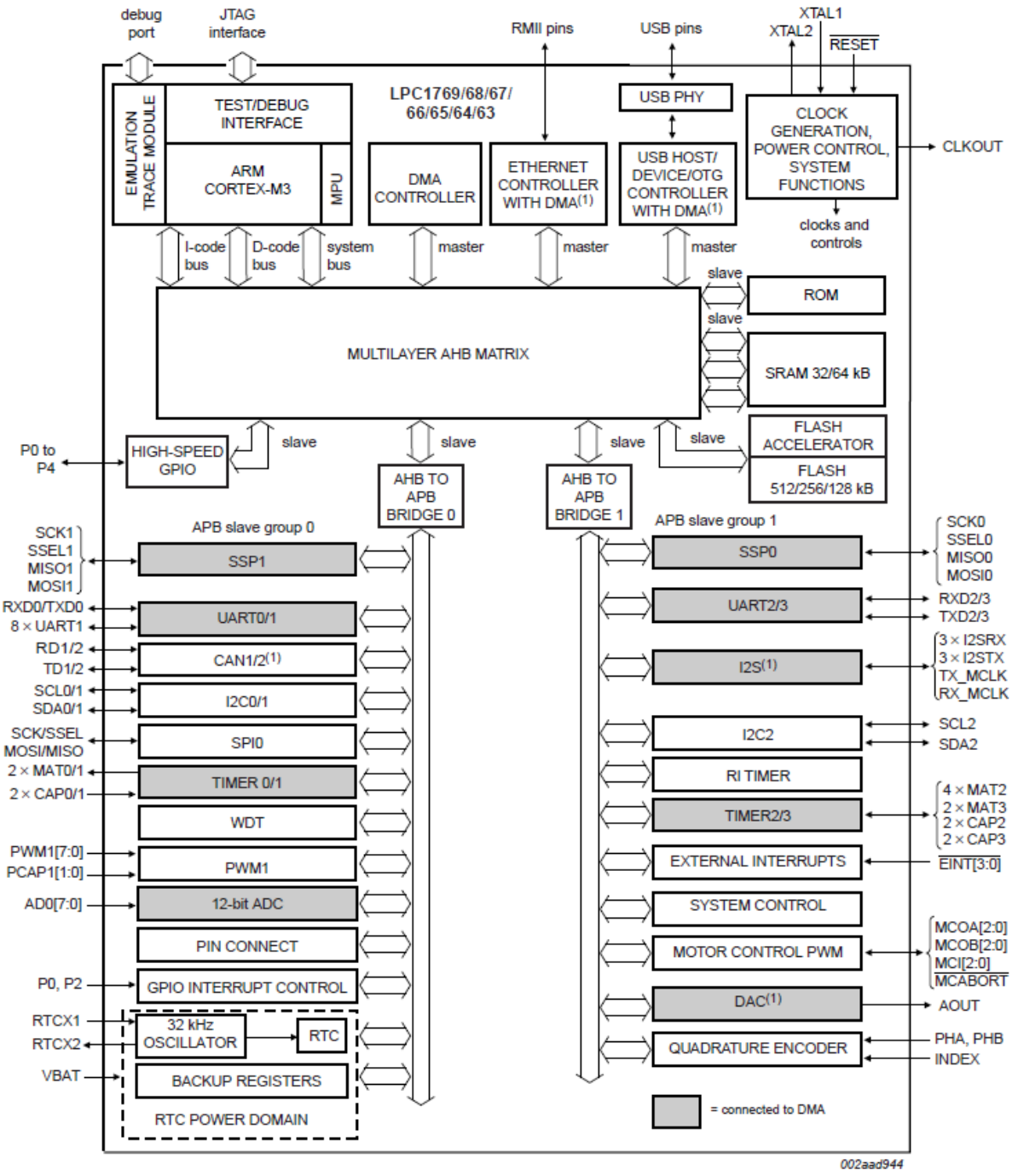
[12] UDP Routines. URL: *http://www.keil.com/support/man/docs/rlarm/rlarm_tn_udp_funcs.htm* [Link], Letzter Zugriff: 19.02.2015

[13] main_TcpNet. URL: *http://www.keil.com/support/man/-docs/rlarm/rlarm_main_tcpnet.htm* [Link], Letzter Zugriff: 19.02.2015

[14] Willert, Andreas: Richtiges richtig tun. Elektronisches Dokument, Willert Software Tools GmbH

8 Anhang

8.1 Block Diagramm NXP LPC1768



8.2 Quellcode Aktivitätsdiagramm

8.2.1 Quellcode der Operationen der Klasse LED

Init(aBitNr:int, aDelay:int)

```
/*## operation Init(int,int) */
void LED_Init(LED* const me, int aBitNr, int aDelay, WST_DUMMY_TASK * p_task) {
    /* Virtual tables Initialization */
    static const WST_FSM_Vtbl LED_reactiveVtbl = {
        rootState_dispatchEvent,
        rootState_entDef,
        NULL,
        (RiCObjectDestroyMethod) LED_Destroy,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
    };
    WST_FSM_init(&(me->ric_reactive), (void*)me, p_task, &LED_reactiveVtbl);
    /* Non-preemptive Framework used, no setTask operation needed. */
    initStatechart(me);
    {
        /*#[ operation Init(int,int) */
        me->BitNr = aBitNr;
        me->Delay = aDelay;
        LPC_GPIO1 -> FIODIR |= 0xB0000000;
        LPC_GPIO2 -> FIODIR |= 0x0000007C;

        LPC_PINCON->PINSEL1 &= ~(3<<18);
        LPC_PINCON->PINSEL1 |=  (1<<18);
        LPC_SC->PCONP       |=  (1<<12);
        LPC_ADC->ADCR        =  (1<< 2) |
                                (4<< 8) |
                                (1<<21);
        /*#]*/
    }
}
```

off():void

```
/*## operation off() */
void LED_off(LED* const me) {
    /*#[ operation off() */
    LPC_GPIO2 -> FIOPIN &= ~(1<<me->BitNr);
    /*#]*/
}
```

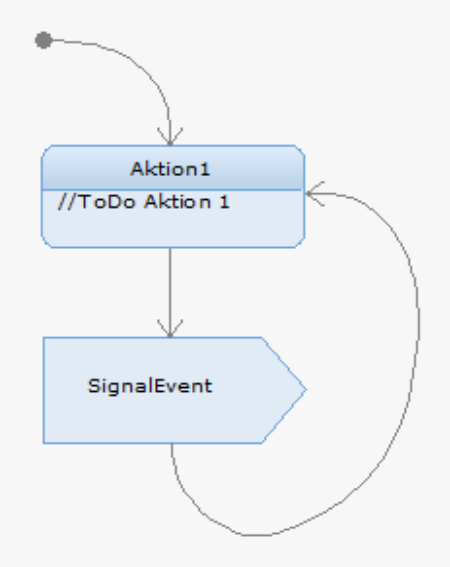
on():void

```
/*## operation on() */
void LED_on(LED* const me) {
    /*#[ operation on() */
    LPC_GPIO2 -> FIOPIN |= (1<<me->BitNr);
    /*#]*/
}
```

Sampling():void

```
/*## operation Sampling() */
void LED_Sampling(LED* const me) {
    /*#[ operation Sampling() */
    LPC_ADC->ADCR |= 0x01200000;
    do {

    } while ((LPC_ADC->ADGDR & 0x80000000) == 0);
    me->Value = LPC_ADC->ADDR2;
    LPC_ADC->ADCR &= ~0x01000000;
    me->Value = (me->Value >> 8) & 0x03FF;
    /*#]*/
}
```

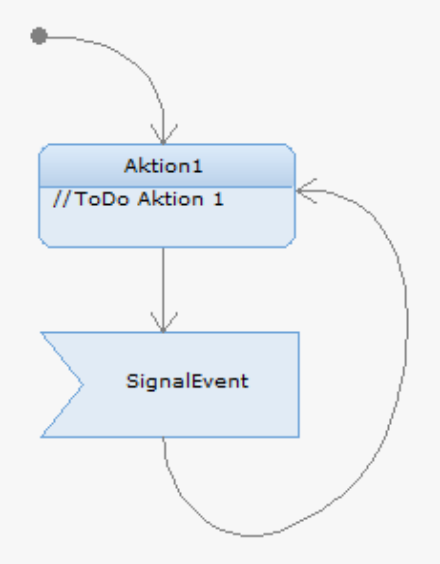
8.2.2 Signalsender (Send Signal Action)




```
static void rootState_entDef(void * const void_me) {

    LED * const me = (LED *)void_me;
    {
        WST_FSM_pushNullConfig(&(me->ric_reactive));
        me->rootState_subState = LED_Aktion1;
        me->rootState_active = LED_Aktion1;
        me->ric_reactive.currentState = me->rootState_active;
        {
            /*#[ state Aktion1.(Entry) */
            //ToDo Aktion 1;
            /*#]*/
        }
    }
}

static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {

    LED * const me = (LED *)void_me;
    RiCTakeEventStatus res = eventNotConsumed;
    switch (me->rootState_active) {


```

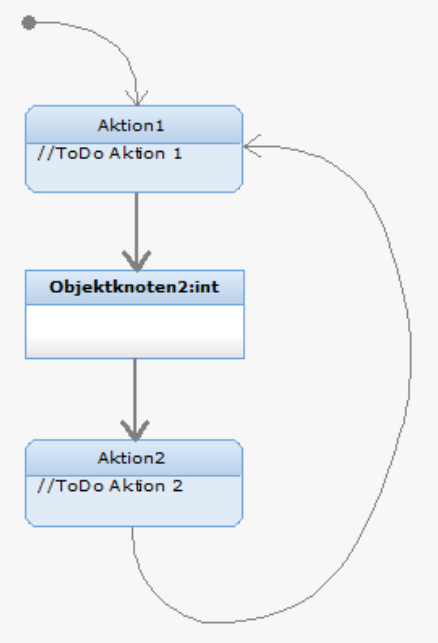
```
24  /* State Aktion1 */
25          case LED_Aktion1:
26          {
27              /*## transition 1 */
28              if(id == Null_id)
29                  {
30                      WST_FSM_popNullConfig(&(me->ric_reactive));
31                      WST_FSM_pushNullConfig(&(me->ric_reactive));
32                      me->rootState_subState = LED_Signalsender;
33                      me->rootState_active = LED_Signalsender;
34                      me->ric_reactive.currentState = me->rootState_active;
35                      {
36                          /*#[ state Signalsender.(Entry) */
37                          FIRE(me, SignalEvent());
38                          /*#]*/
39                      }
40                      res = eventConsumed;
41                  }
42          }
43          break;
44          /* State Signalsender */
45          case LED_Signalsender:
46          {
47              /*## transition 2 */
48              if(id == Null_id)
49                  {
50                      WST_FSM_popNullConfig(&(me->ric_reactive));
51                      WST_FSM_pushNullConfig(&(me->ric_reactive));
52                      me->rootState_subState = LED_Aktion1;
53                      me->rootState_active = LED_Aktion1;
54                      me->ric_reactive.currentState = me->rootState_active;
55                      {
56                          /*#[ state Aktion1.(Entry) */
57                          //ToDo Aktion 1;
58                          /*#]*/
59                      }
60                      res = eventConsumed;
61                  }
62          }
63          break;
64          default:
65              break;
66      }
67      return res;
68  }
```

8.2.3 Ereignisempfänger (Accept Event Action)



```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           me->rootState_subState = LED_Aktion1;
6           me->rootState_active = LED_Aktion1;
7           me->ric_reactive.currentState = me->rootState_active;
8           {
9               /*#[ state Aktion1.(Entry) */
10              //ToDo Aktion 1;
11              /*#]*/
12          }
13      }
14  }
15
16  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
17
18      LED * const me = (LED *)void_me;
19      RiCTakeEventStatus res = eventNotConsumed;
20      switch (me->rootState_active) {
21          /* State Aktion1 */
22          case LED_Aktion1:
23          {
24              /*## transition 1 */
25              if(id == SignalEvent_Default_id)
26                  {
27                      WST_FSM_pushNullConfig(&(me->ric_reactive));
28                      me->rootState_subState = LED_Ereignisemfpaenger;
29                      me->rootState_active = LED_Ereignisemfpaenger;
30                      me->ric_reactive.currentState = me->rootState_active;
31                      res = eventConsumed;
32                  }
33          }
34          break;
35          /* State Ereignisemfpaenger */
36          case LED_Ereignisemfpaenger:
37          {
38              /*## transition 2 */
39              if(id == Null_id)
40                  {
41                      WST_FSM_popNullConfig(&(me->ric_reactive));
42                      me->rootState_subState = LED_Aktion1;
43                      me->rootState_active = LED_Aktion1;
44                      me->ric_reactive.currentState = me->rootState_active;
45                      {
46                          /*#[ state Aktion1.(Entry) */
47                          //ToDo Aktion 1;
48                          /*#]*/
49                      }
50                      res = eventConsumed;
51                  }
52          }
53          break;
54          default:
55              break;
56      }
57      return res;
58  }
```

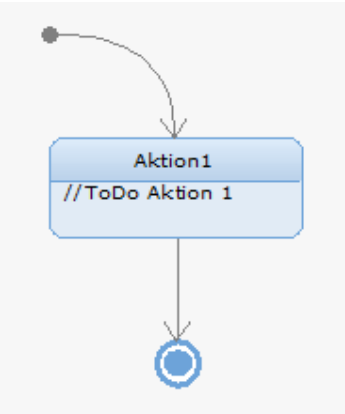
8.2.4 Objektknoten




```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           WST_FSM_pushNullConfig(&(me->ric_reactive));
6           me->rootState_subState = LED_Aktion1;
7           me->rootState_active = LED_Aktion1;
8           me->ric_reactive.currentState = me->rootState_active;
9           {
10              /*#[ state Aktion1.(Entry) */
11              //ToDo Aktion 1;
12              /*#]*/
13          }
14      }
15  }
16
17  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
18
19      LED * const me = (LED *)void_me;
20      RiCTakeEventStatus res = eventNotConsumed;
21      switch (me->rootState_active) {
22          /* State Aktion1 */
23          case LED_Aktion1:
24          {
25              /*## transition 0 */
26              if(id == Null_id)
27                  {
28                      WST_FSM_popNullConfig(&(me->ric_reactive));
29                      WST_FSM_pushNullConfig(&(me->ric_reactive));
30                      me->rootState_subState = LED_Objektknoten2;
31                      me->rootState_active = LED_Objektknoten2;
32                      me->ric_reactive.currentState = me->rootState_active;
33                      res = eventConsumed;
34                  }
35          }
36          break;
37
```

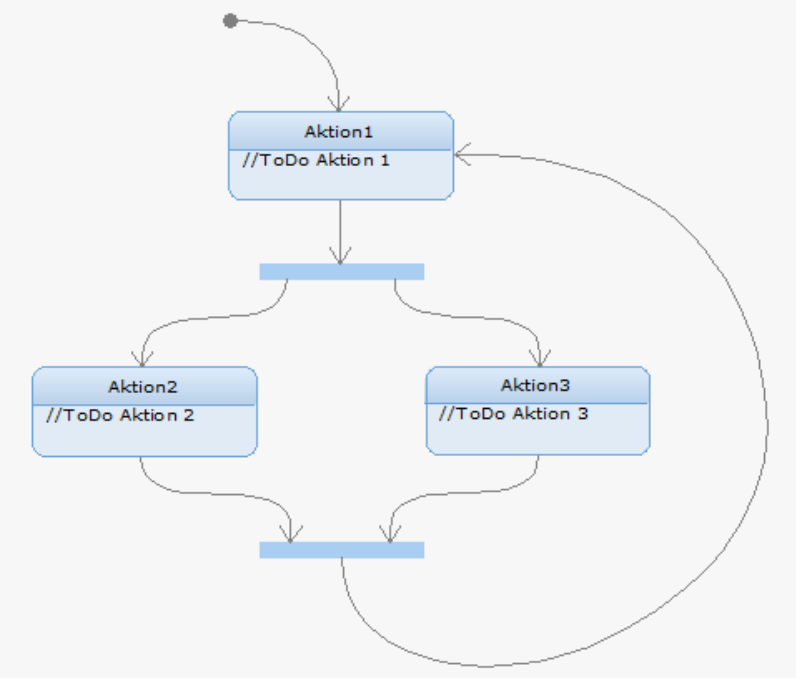
```
38          /* State Aktion2 */
39          case LED_Aktion2:
40          {
41              /*## transition 3 */
42              if(id == Null_id)
43                  {
44                      WST_FSM_popNullConfig(&(me->ric_reactive));
45                      WST_FSM_pushNullConfig(&(me->ric_reactive));
46                      me->rootState_subState = LED_Aktion1;
47                      me->rootState_active = LED_Aktion1;
48                      me->ric_reactive.currentState = me->rootState_active;
49                      {
50                          /*#[ state Aktion1.(Entry) */
51                          //ToDo Aktion 1;
52                          /*#]*/
53                      }
54                      res = eventConsumed;
55                  }
56          }
57          break;
58          /* State Objektknoten2 */
59          case LED_Objektknoten2:
60          {
61              /*## transition 2 */
62              if(id == Null_id)
63                  {
64                      WST_FSM_popNullConfig(&(me->ric_reactive));
65                      WST_FSM_pushNullConfig(&(me->ric_reactive));
66                      me->rootState_subState = LED_Aktion2;
67                      me->rootState_active = LED_Aktion2;
68                      me->ric_reactive.currentState = me->rootState_active;
69                      {
70                          /*#[ state Aktion2.(Entry) */
71                          //ToDo Aktion 2;
72                          /*#]*/
73                      }
74                      res = eventConsumed;
75                  }
76          }
77          break;
78          default:
79              break;
80      }
81      return res;
82  }
```

8.2.5 Start- und Endknoten



```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           WST_FSM_pushNullConfig(&(me->ric_reactive));
6           me->rootState_subState = LED_Aktion1;
7           me->rootState_active = LED_Aktion1;
8           me->ric_reactive.currentState = me->rootState_active;
9           {
10              /*#[ state Aktion1.(Entry) */
11              //ToDo Aktion 1;
12              /*#]*/
13          }
14      }
15  }
16
17  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
18
19      LED * const me = (LED *)void_me;
20      RiCTakeEventStatus res = eventNotConsumed;
21      /* State Aktion1 */
22      if(me->rootState_active == LED_Aktion1)
23          {
24              /*## transition 1 */
25              if(id == Null_id)
26                  {
27                      WST_FSM_popNullConfig(&(me->ric_reactive));
28                      me->rootState_subState = LED_activityfinal_110;
29                      me->rootState_active = LED_activityfinal_110;
30                      me->ric_reactive.currentState = me->rootState_active;
31                      res = eventConsumed;
32                  }
33          }
34      return res;
35  }
```

8.2.6 Parallelisierungs- und Synchronisationsknoten



```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           WST_FSM_pushNullConfig(&(me->ric_reactive));
6           me->rootState_subState = LED_Aktion1;
7           me->rootState_active = LED_Aktion1;
8           me->ric_reactive.currentState = me->rootState_active;
9           {
10              /*#[ state Aktion1.(Entry) */
11              //ToDo Aktion 1;
12              /*#]*/
13          }
14      }
15  }
16
17  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
18
19      LED * const me = (LED *)void_me;
20      RiCTakeEventStatus res = eventNotConsumed;
21      switch (me->rootState_active) {
22          /* State state 0 */
23          case LED_state_0:
24          {
25              res = state_0_dispatchEvent(me, id);
26          }
27          break;
28          /* State Aktion1 */
29          case LED_Aktion1:
30          {
31              /*## transition 1 */
32              if(id == Null_id)
33                  {
34                      WST_FSM_popNullConfig(&(me->ric_reactive));
35                      me->rootState_subState = LED_state_0;
36                      me->rootState_active = LED_state_0;
37                      me->ric_reactive.currentState = me->rootState_active;
38                      WST_FSM_pushNullConfig(&(me->ric_reactive));
39                      me->state_1_subState = LED_Aktion2;
40                      me->state_1_active = LED_Aktion2;
41                      me->ric_reactive.currentState = me->state_1_active;
42                      {
43                          /*#[ state Aktion2.(Entry) */
44                          //ToDo Aktion 2;
45                          /*#]*/
46                      }
47                      WST_FSM_pushNullConfig(&(me->ric_reactive));
48                      me->state_2_subState = LED_Aktion3;
49                      me->state_2_active = LED_Aktion3;
50                      me->ric_reactive.currentState = me->state_2_active;
51                      {
52                          /*#[ state Aktion3.(Entry) */
53                          //ToDo Aktion 3;
54                          /*#]*/
55                      }
56                      res = eventConsumed;
57                  }
58          }
59          break;
60          default:
61              break;
62      }
63      return res;
64  }
65
66  static void state_0_entDef(LED* const me) {
67      me->rootState_subState = LED_state_0;
68      me->rootState_active = LED_state_0;
69      me->ric_reactive.currentState = me->rootState_active;
70      state_1_entDef(me);
71      state_2_entDef(me);
72  }
73
```

```
74  static RiCTakeEventStatus state_0_dispatchEvent(LED* const me, short id) {
75      RiCTakeEventStatus res = eventNotConsumed;
76      /* State state 1 */
77      if(state_1_dispatchEvent(me, id) != eventNotConsumed)
78          {
79              res = eventConsumed;
80              if(!IS_IN(me, LED_state_0))
81                  {
82                      return res;
83                  }
84          }
85      /* State state_2 */
86      if(state_2_dispatchEvent(me, id) != eventNotConsumed)
87          {
88              res = eventConsumed;
89              if(!IS_IN(me, LED_state_0))
90                  {
91                      return res;
92                  }
93          }
94
95      return res;
96  }
97  static void state_2_entDef(LED* const me) {
98
99      WST_FSM_pushNullConfig(&(me->ric_reactive));
100     me->state_2_subState = LED_Aktion3;
101     me->state_2_active = LED_Aktion3;
102     me->ric_reactive.currentState = me->state_2_active;
103     {
104         /*#[ state Aktion3.(Entry) */
105         //ToDo Aktion 3;
106         /*#]*/
107     }
108 }
109 static RiCTakeEventStatus state_2_dispatchEvent(LED* const me, short id) {
110     RiCTakeEventStatus res = eventNotConsumed;
111     /* State Aktion3 */
112     if(me->state_2_active == LED_Aktion3)
113         {
114             if(id == Null_id)
115                 {
116                     /*## transition 6 */
117                     if(IS_IN(me, LED_Aktion2))
118                         {
119                             LED_state_0_exit(me);
120                             WST_FSM_pushNullConfig(&(me->ric_reactive));
121                             me->rootState_subState = LED_Aktion1;
122                             me->rootState_active = LED_Aktion1;
123                             me->ric_reactive.currentState = me->rootState_active;
124                             {
125                                 /*#[ state Aktion1.(Entry) */
126                                 //ToDo Aktion 1;
127                                 /*#]*/
128                             }
129                             res = eventConsumed;
130                         }
131                 }
132
133         }
134     return res;
135 }
136 static void state_1_entDef(LED* const me) {
137
138     WST_FSM_pushNullConfig(&(me->ric_reactive));
139     me->state_1_subState = LED_Aktion2;
140     me->state_1_active = LED_Aktion2;
141     me->ric_reactive.currentState = me->state_1_active;
142     {
143         /*#[ state Aktion2.(Entry) */
144         //ToDo Aktion 2;
145         /*#]*/
146     }
147 }
148
```

```
149  static RiCTakeEventStatus state_1_dispatchEvent(LED* const me, short id) {
150      RiCTakeEventStatus res = eventNotConsumed;
151      /* State Aktion2 */
152      if(me->state_1_active == LED_Aktion2)
153          {
154              if(id == Null_id)
155                  {
156                      /*## transition 6 */
157                      if(IS_IN(me, LED_Aktion3))
158                          {
159                              LED_state_0_exit(me);
160                              WST_FSM_pushNullConfig(&(me->ric_reactive));
161                              me->rootState_subState = LED_Aktion1;
162                              me->rootState_active = LED_Aktion1;
163                              me->ric_reactive.currentState = me->rootState_active;
164                              {
165                                  /*#[ state Aktion1.(Entry) */
166                                  //ToDo Aktion 1;
167                                  /*#]*/
168                              }
169                              res = eventConsumed;
170                          }
171                  }
172
173          }
174      return res;
175  }
```

8.2.7 Quellcode Aktivitätsdiagramm Komparator

```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           WST_FSM_pushNullConfig(&(me->ric_reactive));
6           me->rootState_subState = LED_ReadADC;
7           me->rootState_active = LED_ReadADC;
8           me->ric_reactive.currentState = me->rootState_active;
9           {
10              /*#[ state ReadADC.(Entry) */
11              LED_Sampling(me);
12              /*#]*/
13          }
14      }
15  }
16
17  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
18
19      LED * const me = (LED *)void_me;
20      RiCTakeEventStatus res = eventNotConsumed;
21      switch (me->rootState_active) {
22          /* State ReadADC */
23          case LED_ReadADC:
24          {
25              if(id == Null_id)
26                  {
27                      /*## transition 0 */
28                      if(me->Value > 128)
29                          {
30                              WST_FSM_popNullConfig(&(me->ric_reactive));
31                              WST_FSM_pushNullConfig(&(me->ric_reactive));
32                              me->rootState_subState = LED_Range2;
33                              me->rootState_active = LED_Range2;
34                              me->ric_reactive.currentState = me->rootState_active;
35                              {
36                                  /*#[ state Range2.(Entry) */
37                                  LED_off(me);
38                                  me->BitNr = 3;
39                                  LED_on(me);
40                                  /*#]*/
41                              }
42                              res = eventConsumed;
43                          }
44                      else
45                          {
46                              /*## transition 1 */
47                              if(me->Value <= 128)
48                                  {
49                                      WST_FSM_popNullConfig(&(me->ric_reactive));
50                                      WST_FSM_pushNullConfig(&(me->ric_reactive));
51                                      me->rootState_subState = LED_Range1;
52                                      me->rootState_active = LED_Range1;
53                                      me->ric_reactive.currentState = me->
54                                        rootState_active;
55                                      {
56                                          /*#[ state Range1.(Entry) */
57                                          LED_off(me);
58                                          me->BitNr = 2;
59                                          LED_on(me);
60                                          /*#]*/
61                                      }
62                                      res = eventConsumed;
63                                  }
64                          }
65                  }
66          }
67          break;
68
```

```
69          /* State Range2 */
70          case LED_Range2:
71          {
72              /*## transition 4 */
73              if(id == Null_id)
74                  {
75                      WST_FSM_popNullConfig(&(me->ric_reactive));
76                      WST_FSM_pushNullConfig(&(me->ric_reactive));
77                      me->rootState_subState = LED_ReadADC;
78                      me->rootState_active = LED_ReadADC;
79                      me->ric_reactive.currentState = me->rootState_active;
80                      {
81                          /*#[ state ReadADC.(Entry) */
82                          LED_Sampling(me);
83                          /*#]*/
84                      }
85                      res = eventConsumed;
86                  }
87          }
88          break;
89          /* State Range1 */
90          case LED_Range1:
91          {
92              /*## transition 5 */
93              if(id == Null_id)
94                  {
95                      WST_FSM_popNullConfig(&(me->ric_reactive));
96                      WST_FSM_pushNullConfig(&(me->ric_reactive));
97                      me->rootState_subState = LED_ReadADC;
98                      me->rootState_active = LED_ReadADC;
99                      me->ric_reactive.currentState = me->rootState_active;
100                     {
101                         /*#[ state ReadADC.(Entry) */
102                         LED_Sampling(me);
103                         /*#]*/
104                     }
105                     res = eventConsumed;
106                 }
107         }
108         break;
109         default:
110             break;
111     }
112     return res;
113 }
114         /* State Range2 */
115         case LED_Range2:
116         {
117             /*## transition 4 */
118             if(id == Null_id)
119                 {
120                     WST_FSM_popNullConfig(&(me->ric_reactive));
121                     WST_FSM_pushNullConfig(&(me->ric_reactive));
122                     me->rootState_subState = LED_ReadADC;
123                     me->rootState_active = LED_ReadADC;
124                     me->ric_reactive.currentState = me->rootState_active;
125                     {
126                         /*#[ state ReadADC.(Entry) */
127                         LED_Sampling(me);
128                         /*#]*/
129                     }
130                     res = eventConsumed;
131                 }
132         }
133         break;
134
```

```
135         /* State Range1 */
136         case LED_Range1:
137         {
138             /*## transition 5 */
139             if(id == Null_id)
140                 {
141                     WST_FSM_popNullConfig(&(me->ric_reactive));
142                     WST_FSM_pushNullConfig(&(me->ric_reactive));
143                     me->rootState_subState = LED_ReadADC;
144                     me->rootState_active = LED_ReadADC;
145                     me->ric_reactive.currentState = me->rootState_active;
146                     {
147                         /*#[ state ReadADC.(Entry) */
148                         LED_Sampling(me);
149                         /*#]*/
150                     }
151                     res = eventConsumed;
152                 }
153         }
154         break;
155         default:
156             break;
157     }
158     return res;
159 }
```

8.2.8 Quellcode Aktivitätsdiagramm Komparator Workaround

```
1   static void rootState_entDef(void * const void_me) {
2
3       LED * const me = (LED *)void_me;
4       {
5           WST_FSM_pushNullConfig(&(me->ric_reactive));
6           me->rootState_subState = LED_ReadADC;
7           me->rootState_active = LED_ReadADC;
8    me->ric_reactive.currentState = me->rootState_active;
9           {
10              /*#[ state ReadADC.(Entry) */
11              LED_Sampling(me);
12              /*#]*/
13          }
14      }
15  }
16
17  static RiCTakeEventStatus rootState_dispatchEvent(void * const void_me, short id) {
18
19      LED * const me = (LED *)void_me;
20      RiCTakeEventStatus res = eventNotConsumed;
21      switch (me->rootState_active) {
22          /* State ReadADC */
23          case LED_ReadADC:
24          {
25              if(id == Null_id)
26                  {
27                      /*## transition 0 */
28                      if(me->Value > 128)
29                          {
30                              WST_FSM_popNullConfig(&(me->ric_reactive));
31                              me->rootState_subState = LED_Range2;
32                              me->rootState_active = LED_Range2;
33                              me->ric_reactive.currentState = me->rootState_active;
34                              {
35                                  /*#[ state Range2.(Entry) */
36                                  LED_off(me);
37                                  me->BitNr = 3;
38                                  LED_on(me);
39                                  /*#]*/
40                              }
41                              WST_DUMMY_TASK_schedTm(me->ric_reactive.myTask, me->
42  Delay, LED_Timeout_Range2_id, &(me->ric_reactive), NULL);
43                              res = eventConsumed;
44                          }
45                      else
46                          {
47                              /*## transition 1 */
48                              if(me->Value <= 128)
49                                  {
50                                      WST_FSM_popNullConfig(&(me->ric_reactive));
51                                      me->rootState_subState = LED_Range1;
52                                      me->rootState_active = LED_Range1;
53                                      me->ric_reactive.currentState = me->
54                                        rootState_active;
55                                      {
56                                          /*#[ state Range1.(Entry) */
57                                          LED_off(me);
58                                          me->BitNr = 2;
59                                          LED_on(me);
60                                          /*#]*/
61                                      }
62
```

```
63                                      WST_DUMMY_TASK_schedTm(me->
64  ric_reactive.myTask,me->Delay, LED_Timeout_Range1_id, &(me->ric_reactive), NULL);
65                                      res = eventConsumed;
66                                  }
67                              }
68                  }
69          }
70          break;
71          /* State AcceptTimeEventRange2 */
72          case LED_AcceptTimeEventRange2:
73          {
74              /*## transition 6 */
75              if(id == Null_id)
76                  {
77                      WST_FSM_popNullConfig(&(me->ric_reactive));
78                      WST_FSM_pushNullConfig(&(me->ric_reactive));
79                      me->rootState_subState = LED_ReadADC;
80                      me->rootState_active = LED_ReadADC;
81                      me->ric_reactive.currentState = me->rootState_active;
82                      {
83                          /*#[ state ReadADC.(Entry) */
84                          LED_Sampling(me);
85                          /*#]*/
86                      }
87                      res = eventConsumed;
88                  }
89          }
90          break;
91          /* State Range2 */
92          case LED_Range2:
93          {
94              /*## transition 5 */
95              if(id == Timeout_id)
96                  {
97                      if(RiCTimeout_getTimeoutId((RiCTimeout*) me->
98  ric_reactive.current_event) == LED_Timeout_Range2_id)
99                          {
100                             WST_DUMMY_TASK_unschedTm(me->ric_reactive.myTask,
101 LED_Timeout_Range2_id, &(me->ric_reactive));
102                             WST_FSM_pushNullConfig(&(me->ric_reactive));
103                             me->rootState_subState = LED_AcceptTimeEventRange2;
104                             me->rootState_active = LED_AcceptTimeEventRange2;
105                             me->ric_reactive.currentState = me->rootState_active;
106                             res = eventConsumed;
107                         }
108                 }
109         }
110         break;
111         /* State Range1 */
112         case LED_Range1:
113         {
114             /*## transition 4 */
115             if(id == Timeout_id)
116                 {
117                     if(RiCTimeout_getTimeoutId((RiCTimeout*) me->
118 ric_reactive.current_event) == LED_Timeout_Range1_id)
119                         {
120                             WST_DUMMY_TASK_unschedTm(me->ric_reactive.myTask,
121                               LED_Timeout_Range1_id, &(me->ric_reactive));
122                             WST_FSM_pushNullConfig(&(me->ric_reactive));
123                             me->rootState_subState = LED_AcceptTimeEventRange1;
124                             me->rootState_active = LED_AcceptTimeEventRange1;
125                             me->ric_reactive.currentState = me->rootState_active;
126                             res = eventConsumed;
127                         }
128                 }
129         }
130         break;
131
```

```
132         /* State AcceptTimeEventRange1 */
133         case LED_AcceptTimeEventRange1:
134         {
135             /*## transition 7 */
136             if(id == Null_id)
137                 {
138                     WST_FSM_popNullConfig(&(me->ric_reactive));
139                     WST_FSM_pushNullConfig(&(me->ric_reactive));
140                     me->rootState_subState = LED_ReadADC;
141                     me->rootState_active = LED_ReadADC;
142                     me->ric_reactive.currentState = me->rootState_active;
143                     {
144                         /*#[ state ReadADC.(Entry) */
145                         LED_Sampling(me);
146                         /*#]*/
147                     }
148                     res = eventConsumed;
149                 }
150         }
151         break;
152         default:
153             break;
154     }
155     return res;
156 }
```

8.3 Quellcode Ethernet_Lib

```
#include "Lib_Ethernet.h"
#include <stdlib.h>
#include <string.h>
#include "Ethernet.h"

static U8 socket_udp;         // UDP Socket
#define PORT_NUM_empf 1024   // UDP Port empfangen
#define PORT_NUM_send 1024   // UDP Port senden
#define SENDLEN 1
static Ethernet * me = NULL;

/*-------------------------------------------------------------------------
 @author: Pollithy S., Steinmeyer T.
-------------------------------------------------------------------------
 procrec = Process Receive
 -------------------------------------------------------------------------
 procrec (
  U8* buf);         Pointer to buffer containing the received data
-------------------------------------------------------------------------*/
void procrec (U8 *buf) {

/*-------------------------------------------------------------------------
  Anhand der Empfangenen Daten (1 Byte), welche Informationen über die
  Position des Joysticks enthalten, wird ein Event

   evUDPDataReceived (
    int Data);  enthält Information über die Joystickposition

  an die Klasse Ethernet im UML-Modell gefeuert.

  Information der Daten:
      'L' = Links
      'R' = Rechts
      'U' = Oben
      'D' = Unten
      'S' = Select
-------------------------------------------------------------------------*/
 switch(buf[0]){
  case'L':
  {
   CGEN(me, evUDPDataReceived (JoyStick_LEFT));
   break;
  }
  case'U':
  {
   CGEN(me, evUDPDataReceived (JoyStick_UP));
   break;
  }
  case'R':
  {
   CGEN(me, evUDPDataReceived (JoyStick_RIGHT));
   break;
  }
  case'D':
  {
   CGEN(me, evUDPDataReceived (JoyStick_DOWN));
   break;
  }
  case'S':
  {
    CGEN(me, evUDPDataReceived (JoyStick_SELECT));
   break;
  }
  default:
   break;
 }

 /* done, exec some other work */
 os_tsk_pass();
}
```

```
/*---------------------------------------------------------------------------
 @author: Pollithy S., Steinmeyer T.
---------------------------------------------------------------------------
 UDP callback Funktion
 Wird beim Anlegen des Socket festgelegt und beim Empfang
 von Daten aufgerufen
 ---------------------------------------------------------------------------
 udp_callback (
  U8  socket,     Socket handle of the local machine.
  U8* rip,        Pointer to IP address of remote machine.
  U16 rport,      Port number of remote machine.
  U8* buf,        Pointer to buffer containing the received data.
  U16 len );      Number of bytes in the received data packet.
---------------------------------------------------------------------------*/
U16 udp_callback (U8 soc, U8 *rip, U16 rport, U8 *buf, U16 len) {
  rip  = rip;
  rport= rport;
  len  = len;

  if (soc != socket_udp) {
    // Check if this is the socket we are connected to
    return (0);
  }

  // Empfangene Daten verarbeiten
  procrec(buf);

 return (0);
}

/*---------------------------------------------------------------------------
 @author: Pollithy S., Steinmeyer T.
---------------------------------------------------------------------------
 Funktion zum Senden von Daten via UDP
 Diese Funktion kann aus dem UML-Modell zum Senden von Daten aufgerufen
 werden.
 ---------------------------------------------------------------------------
 UDP send data joystick (
   char Target_IP,    char Array mit der Ziel-IP Adresse
   int Data);         enthält Informationen über die Position des Joysticks
---------------------------------------------------------------------------*/
void UDP_send_data_joystick (char Target_IP[4], int Data){

  U8 *sendbuf;
  U8 IP_send[4];

  // Ziel IP Adresse
  IP_send[0] = (Target_IP[0]);
  IP_send[1] = (Target_IP[1]);
  IP_send[2] = (Target_IP[2]);
  IP_send[3] = (Target_IP[3]);

/*---------------------------------------------------------------------------
  Speicher (1 Byte) für das zu sendende Datenpacket allokieren
---------------------------------------------------------------------------
  U8* udp get buf (
    U16 size);    Number of bytes to be sent
  Return Value: Pointer auf allokierten Speicher
---------------------------------------------------------------------------*/
  sendbuf = udp_get_buf (1);

/*---------------------------------------------------------------------------
   Anhand von Data, welche die Position des Joysticks enthält,
   werden die zu sendenden Daten ermittelt.
   Dies entspricht einem Charakter (1 Byte) für die entsprechende Position:
      'L' = Links
      'R' = Rechts
      'U' = Oben
      'D' = Unten
      'S' = Select
---------------------------------------------------------------------------*/
```

```
  switch(Data){
   case JoyStick_LEFT:{
    sendbuf[0] = 'L';
   break;
   }
   case JoyStick_RIGHT:{
    sendbuf[0] = 'R';
   break;
   }
   case JoyStick_UP:{
    sendbuf[0] = 'U';
   break;
   }
   case JoyStick_DOWN:{
    sendbuf[0] = 'D';
   break;
   }
   case JoyStick_SELECT:{
    sendbuf[0] = 'S';
   break;
   }
   default:
    break;
  }

/*---------------------------------------------------------------------------
 Daten senden via UDP
---------------------------------------------------------------------------
 udp_send (
  U8  socket,     UDP socket to send the data packet from
  U8* remip,      Pointer to the IP address of the remote machine
  U16 remport,    Port number of remote machine to send the data to
  U8* buf,        Pointer to buffer containing the data to send
  U16 dlen );     Number of bytes of data to send
---------------------------------------------------------------------------*/
    udp_send (socket_udp, IP_send, PORT_NUM_send, sendbuf, SENDLEN);
}

/*---------------------------------------------------------------------------
 @author: Pollithy S., Steinmeyer T.
---------------------------------------------------------------------------
 Task  't_tcp_main': main Task für Ethernet
 Initialisierung des UDP Sockets
 öffnen des Sockets
 Zyklisches Aufrufen der main_TcpNet
 ---------------------------------------------------------------------------
 t tcp main (
   void me);  void Pointer auf Ethernetklasse aus UML-Modell
!!!!! Pointer wird benötigt, um Events aus Funktion procrec
      an Ethernetklasse zu senden !!!!!
---------------------------------------------------------------------------*/
__task void t_tcp_main (void* void_me) {

 /* cast void-Pointer void me zu Ethernet-Pointer */
 me = (Ethernet *)void_me;

 /* Initialize the TcpNet */
 init_TcpNet ();

/*---------------------------------------------------------------------------
 Initialize UDP Socket and start listening
---------------------------------------------------------------------------
 U8 udp_get_socket (
    U8   tos,           Type Of Service
    U8   opt,           Option to calculate or verify the checksum
    U16 (*listener));   Function to call when TCPnet receives a data packet
---------------------------------------------------------------------------*/
  socket_udp = udp_get_socket (0, UDP_OPT_SEND_CS | UDP_OPT_CHK_CS, udp_callback);

  if (socket_udp != 0) {
```

```
    /*---------------------------------------------------------------------
     Open UDP port PORT_NUM_empf for communication
    ---------------------------------------------------------------------
    udp open (
    U8  socket,      Socket handle to use for communication
    U16 locport);    Local port to use for communication
    ---------------------------------------------------------------------*/
    udp_open (socket_udp, PORT_NUM_empf);
  }

  while (1) {
    /* Run main TcpNet 'thread' */
    main_TcpNet();
 }
}
```