Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Björn Bettzüche

Entwicklung von Software Anwendungen mit MS Robotics Developer Studio für das RoboCup Rescue Szenario

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Björn Bettzüche

Entwicklung von Software Anwendungen mit MS Robotics Developer Studio für das RoboCup Rescue Szenario

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Ing. Andreas Meisel
Zweitgutachter : Prof. Dr. rer.nat. Wolfgang Fohl

Abgegeben am 19.08.2010

Björn Bettzüche

Thema der Bachelorarbeit

Entwicklung von Software Anwendungen mit MS Robotics Developer Studio für das RoboCup Rescue Szenario

Stichworte

Robotics Developer Studio, Search and Rescue Robotics

Kurzzusammenfassung

Diese Arbeit untersucht die Möglichkeiten und potentielle Eignung der Entwicklungsumgebung Microsoft Robotics Developer Studio für Search and Rescue Robots mit dem Hauptaugenmerk auf Design und Implementierung von Roboteranwendungen für das RoboCup Rescue Szenario.

Björn Bettzüche

Title of the paper

Development of software applications with MS Robotics Developer Studio for the RoboCup Rescue scenario

Keywords

Robotics Developer Studio, Search and Rescue Robotics

Abstract

This bachelor thesis examines the possibilities and potential suitability of the development environment Microsoft Robotics Developer Studio for Search and Rescue Robots with a focus on design and implementation of robotic applications for the RoboCup Rescue scenario.

Inhaltsverzeichnis

1 Projekt AMEE

Das rechtlich nicht geschützte Akronym AMEE stammt aus dem Kinofilm „Red Planet“ (Hoffman, 2000) und steht für „Autonomous Mapping, Exploration and Evasion“, was im Deutschen in etwa „Selbständiges Kartographieren, Erkunden und (Hindernis) Ausweichen“ bedeutet.

Abbildung 1.1: 3D CAD Modell von AMEE (erstellt mit C4D)

1.1 Motivation

Nach Katastrophen und Unfällen ist das dringendste Problem, Opfer zu finden und zu bergen, dabei aber auch die Unversehrtheit der Rettungskräfte zu gewährleisten. Jedoch ist der Weg zum Einsatzort oft durch Trümmer oder Feuer versperrt. Daher sind sogenannte Urban Search And Rescue (USAR, siehe unten) Roboter Gegenstand aktueller Forschung und Entwicklung, zu dem auch ein alljährlich stattfindender internationaler Wettbewerb veranstaltet wird (siehe RoboCup Rescue). An der HAW gibt es Bestrebungen, in dieser Hinsicht Projekte ins Leben zu rufen, so auch den Roboter AMEE von den Kommilitonen J. Ruhnke, D. Schnell und dem Autor dieser Arbeit.

Bei Roboteranwendungen hat man es im besonderen Maße mit nebenläufigen Prozessen und Asynchronität in einem verteilten System zu tun. Auch die in der Regel beschränkten Hardware-Ressourcen stellen im Entwicklungsprozess oftmals einen erheblichen (Zeit-) Kostenfaktor dar. Microsoft bietet mit dem Robotics Developer Studio (Microsoft Corporation, 2010f) eine bereits in der Industrie eingesetzte Gesamtlösung für die Entwicklung von Roboteranwendungen. Sie beinhaltet ein Architektur- und Programmiermodell, basierend auf .NET-Technologien, sowie eine Simulationsumgebung und Authoringtools.

Im Rahmen dieser Arbeit und als Vorbereitung auf das oben genannte studentische Projekt zur Entwicklung des Rescue Robots AMEE wird der Einsatz von Microsoft Robotics Developer Studio (RDS) einer Machbarkeitsprüfung unterzogen. Zunächst werden einige generelle Aspekte der Search and Rescue Robotic betrachtet und der RoboCup Rescue zusammenfassend erläutert (1). Das Kapitel Robotics Developer Studio (2) klärt die Hintergründe zum Programmiermodell CCR und dem Architekturmodell DSS sowie deren Benutzung. Beide Modelle werden einigen Performanztests und Beurteilungen unterzogen. Folgend wird am Beispiel einer Motorsteuerung eine Integration von Hardwarekomponenten sowie eine zugehörige einfache Robotics Studio Anwendung demonstriert. Basierend auf den Analyseergebnissen dieser drei Abschnitte zieht das abschließende Kapitel ein Resümee (3), das ein erstes und noch vereinfachtes Anwendungsdesign für AMEE beinhaltet und eine Beurteilung des Robotics Studio für den Einsatz im Projekt abgibt.

Diese Arbeit betrachtet vordergründig das Anwendungsdesign und der Implementierung in C#, der Fokus liegt daher auf dem CCR & DSS Toolkit. Weitere Tools des Robotics Developer Studio wie die Visual Programming Language oder das Simulationstool sind nicht Gegenstand dieser Arbeit.

1.2 Search and Rescue Robotics

Im Kapitel “Search and Rescue Robotics” aus (Murphy, et al., 2008b) beschreiben die Autoren kompakt, aber sehr fundiert die Anforderungen, den aktuellen Entwicklungsstand und die Lehren aus vergangenen Einsätzen für Rescue Robotics. Andere, hier nicht weiter verwendete Quellen, bestätigen deren Beurteilung. Folgende Beschreibung der Thematik „Search and Rescue Robotics“ basiert somit auf dem oben genannten Kapitel.

Alle Katastrophenszenarien, bei denen Rescue Robotics eingesetzt werden sollen, haben ein oberstes gemeinsames Ziel: die Rettung von Leben. Es gibt aber kein Roboter-Design, das für alle denkbaren Szenarien optimal ist. Vielmehr muss es konkretisierten Aufgaben angepasst werden. Die Autoren identifizieren zehn nicht zwangsläufig disjunkte Einsatzarten:

1. **Suchen**: Aufspüren von Opfern inner- oder unterhalb von Strukturen
2. **Erkundung und Kartographierung**
3. **Trümmerbeseitigung**: schneller als manuell; kleiner als schwere Kräne/Bagger
4. **Strukturelle Prüfung**
5. **Medizinische Untersuchung/Versorgung** vor Ort
6. **Bergung/Evakuierung**: unter medizinischen Aspekten möglichst schonend für das Opfer
7. **Mobiles Funkfeuer** oder Verstärker: Kommunikations-Reichweite/-Stabilität erhöhen
8. **Telepräsenz**: als Ersatz für ein Teammitglied vor Ort dienen
9. **Adaptive Abstützung**[1]: Sicherung instabiler Strukturen (z.B. Tunnel, Wände)
10. **Logistische Unterstützung**: Transport von Hilfsgütern und Material

Im Gegensatz zu vielen Industrierobotern können sich Search and Rescue Roboter fortbewegen. Sie gehören zur Kategorie der Field Robotics und werden gemeinhin nach ihrem Fortbewegungsmedium klassifiziert:

- **UGV**: Unmanned Ground Vehicle
- **UAV**: Unmanned Aereal Vehicle
- **UUV**: Unmanned Underwater Vehicle
- **USV**: Unmanned Surface Vehicle

Jede Fortbewegungsart hat ihre Vor- und Nachteile, abhängig vom Einsatzszenario. So ist das durch Naturkatastrophen betroffene Gebiet oft sehr großflächig. Daher werden hier bevorzugt UAVs eingesetzt. Von Menschenhand herbeigeführte Katastrophen (ungewollt

[11] Im Originaltext: „adaptive shoring“

oder beabsichtigt), wie Fabrikunfälle oder Grubeneinstürze, sind in der Regel an einem Ort konzentriert. Hier nutzt man eher UGVs und UUVs. Boote (USVs) dienen meist der Inspektion in Kanälen oder werden bei Überschwemmungen eingesetzt. Unterhalb dieser Modalitäten kategorisieren die Autoren noch nach der Größe des Roboters:

- **Man-packable**: klein/leicht genug, sie über Schutt oder Leitern mit sich zu tragen
- **Man-portable**: von zwei Personen über kurze Instanzen zu tragen
- **Maxi**: benötigen Transportmittel wie Anhänger o.ä.

Der geplante Roboter AMEE ist nach dieser Klassifizierung und ersten Schätzungen zu diesem sehr frühen Entwicklungsstadium ein man-portable UGV. Beim ersten Prototyp werden sich AMEEs Aufgaben auf das Suchen, Erkunden und Kartographieren beschränken.

Das Ausgangsszenario beim RescueCup ist ein Erdbeben und die Testumgebung ist für den UGV-Einsatz in zerstörten Siedlungen ausgelegt. Solche Missionen fallen in die Kategorie Urban Search and Rescue (USAR oder US&R)[2]. Obwohl ein Erdbeben eine typischerweise großflächige Naturkatastrophe ist, für die die Autoren eindeutig den Einsatz von UAVs am hilfreichsten einschätzen, ist das Szenario des RescueCups nicht zwangsläufig unrealistisch. Denn auch die Autoren berichten, dass mehrere Helferteams selbständig und unabhängig voneinander gezielt erst dort suchen, wo am ehesten und am meisten Opfer zu erwarten sind, also in Wohnsiedlungen, Einkaufszentren, Verkehrsknotenpunkten, etc. Somit ist ein Einsatz von UGVs für die partielle Untersuchung von Gebäuden und Gebäudekomplexen gerechtfertigt.

Bei Such- und Rettungs-Missionen ist ein entscheidender Faktor die Zeit. Erfahrungen zeigen, dass verschüttete Opfer innerhalb von 48 Stunden gefunden werden müssen. Danach tendiert die Wahrscheinlichkeit, Überlebende zu finden, gegen Null. Bei Menschen, die lediglich von ihrer Umwelt abgeschnitten sind, erweitert sich das Zeitfenster auf 72 Stunden. Für USAR Robotersysteme gilt folglich, dass sie die Rettungskräfte unterstützen und vor allem den Einsatz beschleunigen sollen. Ein USAR Roboter muss desweiteren außergewöhnlichen physischen Belastungen standhalten und extremen Mobilitätsanforderungen gerecht werden. Ein großes und bisher nicht zufriedenstellend gelöstes Problem stellt auch die drahtlose Kommunikation zum Roboter dar. Schon die normal übliche Kommunikation der menschlichen Hilfskräfte ist durch Interferenzen im Funkverkehr gekennzeichnet. Eine Forderung bzw. Lehre aus dem Einsatz von Robotern nach dem Attentat auf das World Trade Center (2001, New York) war daher, bevorzugt kabelgebundene Kommunikation zu benutzen, am besten integriert in eine Sicherungsleine, mit der man den Roboter notfalls herausziehen kann. Aber auch diese Lösung ist unter Aspekten der Mobilität und Autonomität des Roboters nicht befriedigend (damals wurden nur

[2] Daneben gibt es unter anderem noch Such- und Rettungsmissionen in der Wildnis, welches in Deutschland nach Aussage eines Rettungshundeführers (glücklicherweise) der häufigste Einsatz für Rettungshunde darstellt.

ferngesteuerte, aber nicht autonome Roboter eingesetzt). Was die Erfahrungen aus New York ebenfalls erbracht haben ist, dass Wärmebildkameras prinzipiell zwar zum Aufspüren von Menschen geeignet sind, jedoch an Orten mit vielen, Feuer bedingten Hitzequellen keinen Mehrwert haben[3].

Als Fortbewegungsart für UGVs haben sich aktuell sowohl in Forschung, als auch im realen Einsatz für Militär-/Regierungs-Organisationen polymorphe Kettenfahrzeuge etabliert. Diese haben meist zusätzlich schwenkbare Ausleger, sodass kleinere Unebenheit oder Steilhänge (Treppen, Absätze) besser überwunden werden können. Trotzdem erfüllen sie noch nicht die hohe Anforderung an USAR Roboter. So haben die 2005 in La Conchita (Kalifornien) nach einer Schlammlawine eingesetzten Roboter schon nach wenigen Minuten versagt, weil sich immer wieder Wurzeln und Äste im Kettengetriebe verhakt haben. Als sehr vielversprechend beurteilen die Autoren stattdessen biomimetische Antriebsarten wie bei dem schlangenähnlich kriechenden „Soyu“ vom International Rescue System Institute (IRS) oder Bein-Antriebe wie beim Hexapod vom RHex Projekt, beziehungsweise wie es auch bei AMEE der Fall sein wird.

1.2.1 Grundlegende Probleme und offene Fragen

Rescue Robotics ist ein Forschungsfeld mit vielen Herausforderungen und ungelösten Problemen. Über den Ansprüchen an die einzelnen Subsysteme, stehen drei übergreifende Anforderungen: Die Missionszeit verkürzen; Lokalisierungs- und Kartographierungsdaten nutzbringend in übergeordnete Informationssystem integrieren; Die allgemeine Zuverlässigkeit von Rescue Robots verbessern. In der bisherigen Praxis hat sich gezeigt, dass UGVs nur 55% ihrer Einsatzzeit in Bewegung sind. Entweder, weil die Mobilität eingeschränkt oder weil ein Ort zu lange überprüft werden muss, da die gelieferten Informationen ungenügend sind. Hier gibt es viel Potential, die Mission zu beschleunigen. Auch ist die Ausfallsicherheit bezüglich physikalischer Ursachen noch ungenügend. Die Mean Time Between Failure betrug 2004 zwanzig Stunden, während das US-Militär, zum Vergleich, 96 Stunden fordert. Folgend sind die Anforderungen an einige Subsysteme, die auch eine hervorstechende Betrachtung für den geplanten ersten Prototyp von AMEE haben, vorgestellt.

[3] Während einer Mission des erwähnten Rettungshundeführers führte ein mit Wärmekamera ausgestattetes UAV die Suchkräfte zu einem Trafohaus.

1.2.1.1 Mobilität

Ein UGV muss mit verschiedensten Untergründen und Hindernissen zurechtkommen, auch um in möglichst vielen Situationen eingesetzt werden zu können. Sowohl die Mechanik als auch die Algorithmen müssen sich den wandelnden Bedingungen dynamisch anpassen können. Die Autoren teilen die Einschätzung des AMEE-Projektteams, dass beinbasierte Systeme vielversprechend sind.

1.2.1.2 Steuerung

Diese Thematik betrifft sowohl die Theorie der Steuerungstechnik (Plattform), als auch die künstliche Intelligenz (Verhalten). Bisher werden meist ferngesteuerte Roboter mit einem sehr geringen Grad an Autonomität eingesetzt. Die Autoren sehen die Zukunft aber in einer Verlagerung hin zur künstlichen Intelligenz (AI)[4]. Viel Potential hat das „Shared Roles Model“, wie es in (Murphy, et al., 2008a) beschrieben ist[5]. Lapidar zusammengefasst bilden dort Mensch und Maschine eine mentale Einheit, während der Roboter grundlegende Funktionen autonom umsetzt (wie ein zweites Unterbewusstsein). Doch dies fällt in das eigenständige Thema Human-Robot Interaction (HRI).

1.2.1.3 Sensoren

Abhängig von der Roboterausführungsart und dem Missionsziel können die benötigten Sensoren variieren. Generell für SAR-Robots wären folgende Sensoren von unverzichtbaren Nutzen: Abstands-/Bereichs-Sensoren zum Lokalisieren und Kartographieren; welche, die durch Schutt verdeckte Opfer detektieren; Sensoren, die bewusstlose aber noch lebende Opfer erkennen. Zusätzlich ist noch Forschungsbedarf bei Auswertungs- und Interpretations-Algorithmen.

Weitere Herausforderungen gibt es den Bereichen Kommunikation, Energiekapazität/-effizienz, HRI und Evaluation. Zusammenfassend kann man behaupten, dass die Thematik Search and Rescue Robotic sehr herausfordernd ist und multiple Forschungszweige tangiert. Bezogen auf das Projekt AMEE wäre es utopisch anzunehmen, dass die drei Initiatoren, dies alleine bewältigen könnten. Daher ist es aber auch von vornherein als fakultätsübergreifendes, mehrjähriges Projekt geplant, aus dem vielfältige Forschungsarbeiten herausgehen können.

[4] Die Roboter des RescueCup, so auch AMEE, sollen fast ausschließlich autonom agieren.
[5] Persönliche Einschätzung des Autors dieser Arbeit

1.3 RoboCup Rescue

Nach der Selbstdarstellung des RoboCup Rescue (Tadokoro, et al., 2009) wurde das Projekt anlässlich des Erdbebens im japanischen Kobe (17. Jan 1995) ins Leben gerufen. Die Lehre aus diesem Erdbeben sei unter anderem, dass Informationssysteme mit folgenden Anforderungen entworfen werden sollten:

- Erfassen, sammeln, weiterleiten, auswählen, zusammenfassen und verteilen von notwendigen Informationen.
- Umgehende Hilfestellung zur Planung von Katastrophenentschärfung, Such- und Rettungsaktionen.
- Zuverlässigkeit und Robustheit des Systems in Übungen und Ernstfällen.

Basierend auf obige Anforderungen, sei das Ziel des RoboCup Rescue Projekts, die Entwicklung und Forschung in diesem Bereich auf verschieden Ebenen zu fördern. Dies beinhalte die Arbeitskoordination von Multi-Agenten-Systemen, physische Such- und Rettungsroboter, Informationsinfrastrukturen, personengebundene digitale Assistenten, einen Standardsimulator sowie Systeme zur Entscheidungshilfe und Evaluation für Rettungsstrategien und Robotersysteme. Letztlich solle alles in einem künftigen umfassenden System integriert werden. Die RoboCup Rescue Initiative ist in zwei Hauptstränge geteilt:

- Das RoboCup Rescue Simulation Project, wiederum in zwei Teilgebiete gegliedert: den Virtual Robots und den Agent Simulation Projekten.
- Der Wettbewerb für Roboter im **Real Robotics** Projekt reicht von Mechatronik für anspruchsvolle Fortbewegung über Wahrnehmung und Interpretation bis zur Erbringung vollständiger Autonomie. Die Roboter werden in einer speziellen Testumgebung geprüft, den NIST rescue arenas (National Institute of Standards and Technology).

Die Wettkampfarenen sind nach Schwierigkeitsgrad und Anforderungen an Autonomie, Mobilität und Erkundung unterteilt. Es wird versucht eine Umgebung mit möglichst reellen Herausforderungen für die Roboter und damit ein einheitliches Testszenario mit objektiven Bewertungsmöglichkeiten zu schaffen. Innerhalb der Arena werden Puppen mit multiplen simulierten Lebensmerkmalen versteckt, die der Roboter aufzuspüren hat.

Abbildung 1.2: Arenen 2008 (National Institute of Standards and Technology)

2 Robotics Developer Studio

In diesem Kapitel werden Grundkenntnisse und ein Überblick zum CCR & DSS Toolkit des Robotics Developer Studio vermittelt. Betrachtet mit dem Hintergrund aus Kapitel 1, Projekt AMEE, schließen die ersten beiden Abschnitte mit einem Performanztest und eine Zwischenbilanz ab. Der Abschnitt 2.3 wendet das Toolkit schließlich an, um eine selbst entwickelte MicroController Einheit zur Steuerung eines Schrittmotors in Robotics Studio zu integrieren.

2.1 Concurrency and Coordination Runtime

Schon halbwegs komplexe Roboter besitzen in der Regel eine ganze Reihe von Sensoren und Aktoren, die von diversen Softwarekomponenten überwacht und gesteuert beziehungsweise geregelt werden. Wieder andere Softwarekomponenten bedienen sich der vorherigen, um das gewünschte Verhalten des Systems zu realisieren. Der Roboter interagiert mit einer realen Umwelt. Das heißt insbesondere, dass oft auf äußere Ereignisse reagiert werden muss, die zu unvorhersagbaren Zeitpunkten auftreten. Angesichts der meist stark eingeschränkten Ressourcen in Robotersystemen wäre es ineffizient, mit Polling zu arbeiten. Also beispielsweise periodisch die Daten des Kontaktsensors auszuwerten oder zu überprüfen, ob ein Befehl per Fernsteuerung angekommen ist. Dieses Beispiel veranschaulicht, dass es vielmehr wünschenswert wäre, parallel auf verschiedene asynchron auftretende Ereignisse zu reagieren.

Die Concurrency and Coordination Runtime (CCR[6]) ist ein Programmiermodell, dass sich im Wesentlichen der parallelen Verarbeitung und Koordinierung asynchroner Nachrichten widmet. Sie basiert auf Microsofts Common Language Runtime (CLR) und unterstützt die .NET Sprachen C# und Visual Basic.

[6] Nicht zu verwechseln mit der US-amerikanischen Rockband aus den 60ern/70ern

2.1.1 Was CCR bietet

Die Vorteile paralleler Verarbeitung in einem autonomen Roboter sind offensichtlich. Es können mehrere Aufgaben nebenläufig bearbeitet werden und das System bleibt auf weitere Ereignisse ansprechbar. Latenzzeiten, wie das Warten auf neue Sensordaten oder die Auswertung dessen können mit anderen Aufgaben überbrückt werden. Das System ist skalierbar. So macht beispielsweise die Verarbeitung von drei oder dreitausend Nachrichten, zumindest programmatisch, keinen Unterschied. Eine nebenläufige, verteilte Anwendung erfordert allerdings auch ein erhebliches Maß an Koordination und meist komplexe Mechanismen zur Fehlerbehandlung, Wahrung der Datenkonsistenz und vieles mehr. CCR stellt dafür einfache Methoden und Konstrukte bereit, die die Verwendung von sogenannten Primitves (Mutex, Lock, Semaphore,…) und Erzeugung von Threads verbirgt sowie die Handhabung von Fehlern in asynchronem nebenläufigen Code vereinfacht. Außerdem stellt die Runtime verschiedene Scheduling-Mechanismen zur Verfügung, die unter anderem zeitliche Aspekte oder Datenabhängigkeiten berücksichtigen.

Die Funktionalitäten der CCR-Library kann man in drei Hauptkategorien unterteilen.

Ports bzw. **PortSets** sind FIFO-Queues, die Daten aufnehmen. Dies können primitive Datentypen oder selbstdefinierte CLR-Typen sein (CCR unterstützt keine direkte COM-Interoperabilität). Ports sind bei Services die Schnittstelle nach außen.

Die neuen koordinativen Primitive wie Receive, Join oder Choice werden allgemein **Arbiter** genannt. Sie lesen die Ports aus, filtern die empfangenen Daten nach definierten Bedingungen (constraints) und sorgen dafür, dass die Daten von geeigneten, benutzerdefinierten Codes (Handler-Methoden/Delegaten) verarbeitet werden. Die Primitve können verschachtelt und vom Benutzer erweitert werden.

Dispatcher, **DispatcherQueue** und **Task** sind die dritte Kategorie, in der insbesondere die Threadverwaltung und nebenläufige Ausführung stattfindet. Durch diese Klassen kann der Benutzer festlegen, wie und mit welchen Ressourcen die Aufgaben auszuführen sind. Der Dispatcher enthält einen Threadpool mit fixer Größe zur Verarbeitung der Aufgaben (Tasks). Die DispatcherQueues eines Dispatchers sind die einzige Möglichkeit mit ihm zu interagieren.

Im Folgenden wird der Ablauf vom Datenempfang hin zur Verarbeitung und somit der Zusammenhang zwischen den Klassen vereinfacht erläutert.

Ein Port stellt die Schnittstelle nach außen dar. Eine beliebige Softwarekomponente kann jederzeit mittels `Post()` Daten einfügen. Der Arbiter entnimmt die Daten und überprüft die Conditions, also ob das Datum verarbeitet werden kann. Wenn nicht, wird das Datum verworfen. Die empfangenen Daten werden mit dem zugehörigen Handler (Methodenverweis) zu einer Task zusammengefügt und in eine DispatcherQueue eingereiht. Damit ist der Arbiter wieder bereit, neue Daten zu überprüfen. Wenn ein Thread aus dem Threadpool des Dispatchers bereit ist, wird nach dem Round-Robin-Verfahren eine DispatcherQueue ausgewählt und die nächste Task dem Thread übergeben, der die Handlermethode mit dem Datum als Argument ausführt.

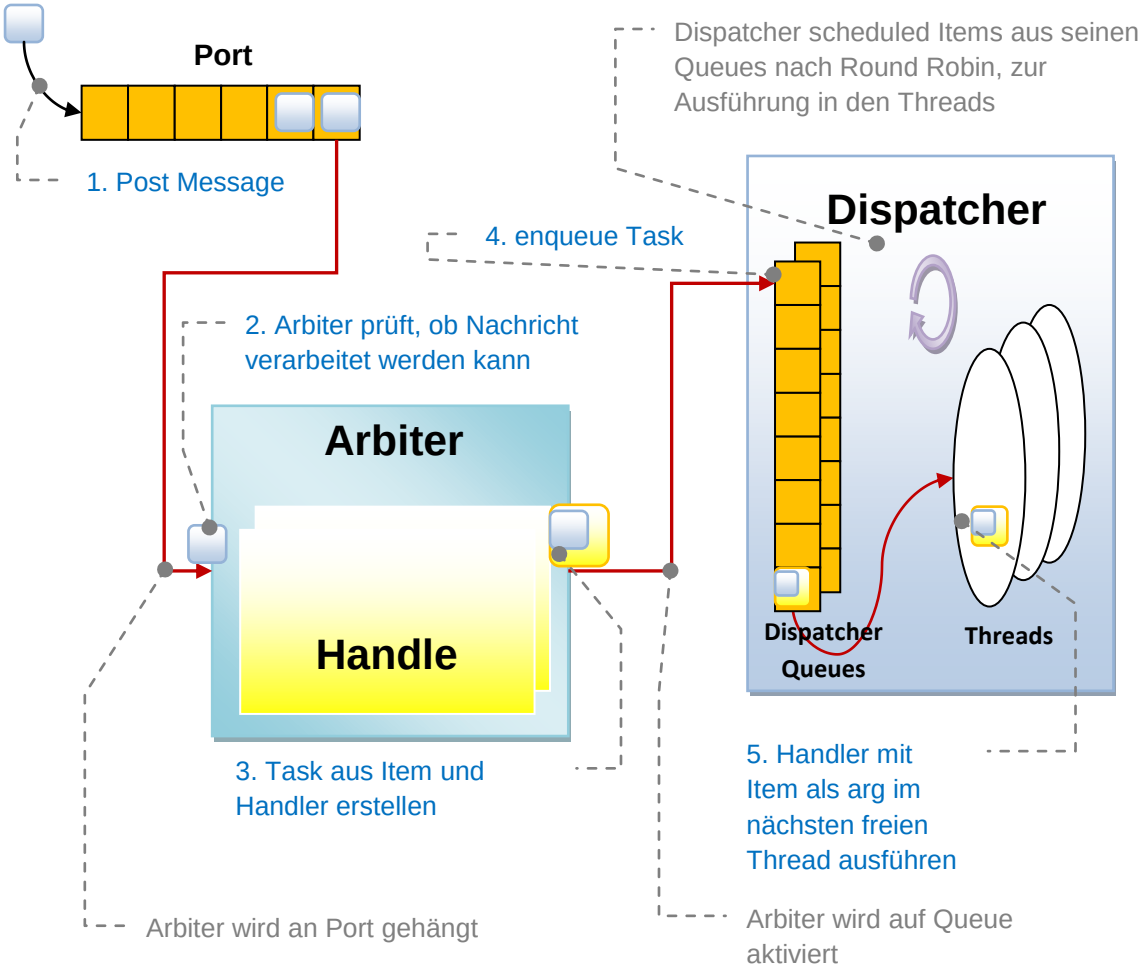


Abbildung 2.1: Übersicht CCR-Modell, frei nach (Chrysanthakopoulos, 2008)

Im Gegensatz zu CLR ist die Größe des Threadpools in CCR fix. Die Anzahl zu erzeugender Threads ist standardmäßig gleich der Anzahl vorhandener CPUs, mindestens aber zwei. Der dadurch geringere Overhead für das Erzeugen und Vernichten von Threads bringt einen potentiellen Performanz-Gewinn und das System ist besonders gut skalierbar. Wenn vier Kerne vorhanden sind, werden vier Threads bereitgestellt, wenn vier Millionen Kerne vorhanden sind, werden auch entsprechend viele Threads erzeugt. Im Grunde handelt es sich bei dem realisierten Konzept um Message-Passing. Gegenüber Remote Methode Invocation hat es gerade in einem verteilten System den Vorteil, dass lediglich Nachrichten übermittelt werden. Auftretende Fehler müssen nicht über Thread- oder gar Maschinengrenzen hinweg durchgereicht werden. Stattdessen wird ein unbrauchbares Datum verworfen und Verarbeitungsfehler werden dort behandelt, wo sie auftreten. Dies isoliert den Aufrufer vom Aufrufenden und die Daten vom Ausführungskontext.

2.1.2 Ports und PortSets

2.1.2.1 Port

Ein **Port** ist eine streng typisierte Queue, die auch das Einfügen von Null-Referenzen erlaubt und an der ein Receiver (ein Arbiter-Typ) angeknüpft wird. Einfügeoperationen sind asynchron, der Kontrollfluss geht unverzüglich an den Aufrufer zurück. Es gibt nur einen generischen Default-Konstruktor. Die wichtigsten Methoden von **Port** sind folgend aufgelistet:

Methode	Beschreibung
Post(T item) : void	Fügt eine Nachrichten Instanz ein
Test(out T item) :bool	Entfernt atomar ein Element, falls vorhanden. Gibt false, wenn keines vorhanden.
Implicit cast (Port<T> port) : T	Führt Test-Methode aus.

Tabelle 2.1: Auswahl einiger Port Instanzmethoden

Im folgenden Code-Beispiel wird der String-Port *portStrg* als passive Queue benutzt. Dies ist im CCR-Kontext zwar eher untypisch, aber trotzdem in Szenarien einsetzbar, wo man einfach eine relativ effiziente, thread-sichere Queue benötigt. Mit der Methode `Post` werden drei Instanzen eingefügt und anschließend wieder ausgelesen. Zunächst das erste Element mit der parametrisierten Methode `Test`. Diese gibt true zurück, wenn *portStrg* ein Element enthält, entfernt es und übergibt es an den Referenzparamer. Die restlichen Elemente werden mit dem *implicit* Operator in einer Schleife ausgelesen.

```
void usingPort() {
    Port<string> portStrg = new Port<string>();// Port für String-Instanzen

    for (int i = 10; i <= 30; i+=10)
        portStrg.Post(""+i); // Einfügen
    // Anzahl der Elemente im Port
    Console.WriteLine("#Items: "+portStrg.ItemCount);

    string item;
    if ( portStrg.Test(out item) ) //Eintrag mit Test abrufen
        Console.WriteLine(item);


    while (item != null) {
        item = (string)portStrg; //mit implizit cast abgerufen
        Console.WriteLine(item);
    }
}
```

Listing 2.1: Port, einfügen und entnehmen

2.1.2.2 PortSet

Häufig kann ein Service mehrere verschiedene Datentypen empfangen. Die Klasse **PortSet** bündelt hierfür mehrere unabhängige Ports in einer Instanz. Das generische **PortSet<T0,…,TN>** kann für bis zu zwanzig Typen angelegt werden (im Compact Framework bis zu acht). Über die Eigenschaftsfelder P0 bis PN oder einem impliziten Cast sind die einzelnen generischen Ports erreichbar. Die nicht generische Klasse PortSet dient hauptsächlich für die Fälle, bei denen die Anzahl der Ports zur Laufzeit unbekannt ist. Dem Konstruktor wird eine Liste von Typen übergeben, der dann entsprechende Ports dynamisch initialisiert. Da hier intern vermehrt Typüberprüfungen notwendig sind, kann die Performance etwas schlechter als bei der generischen Variante[7] sein. Man beachte, dass die deklarative Anordnug der verwendeten Typen nicht unbedingt mit der Reihenfolge der erzeugten Ports

[7] Tests ergaben, dass durch Optimierungen zur Laufzeit unter Umständen auch der nicht-generische PortSet schneller sein kann

übereinstimmen muss. So werden beispielsweise bei PortSet<int,int> zwei Port<int> erzeugt und es ist keine garantierte Aussage darüber zu treffen, auf welchen Port Einfüge- und Entferne-Operationen zugreifen. In solchen Fällen sollte man sich mit selbstdefinierten Klassen/Strukturen behelfen.

Die wichtigsten Methoden von **PortSet<T0,…,TN>** sind folgend aufgelistet:

Methode	Beschreibung
Post(T_i item) : void	In den/einen Port<T_i> einfügen
Pi {get;}	i steht für 0 bis N; gibt den i'ten Port
Implicit cast (PortSet<T0,…>) : Port<T>	Liefert den/einen enthaltenen Port<T>
Implicit cast (PortSet<T0,…>) : T	Ruft Test bei dem/einem Port<T> auf.
Test<T>() : T	Entfernt atomar aus dem entsprechenden Port<T>.
TryPostUnkownType(Object o) : bool	Fügt in einen kompatiblen Port ein. Liefert false, wenn keiner vorhanden.

Tabelle 2.2: PortSet Methoden

Das folgende Code Beispiel zeigt die Instanziierung eines generischen und eines untypisierten PortSets. In beide werden jeweils ein int und ein string eingefügt. Bei der nichtgenerischen PortSet *runtimePS* kann sowohl bei der Einreihenoperation `PostUnkownType` als auch bei der generischen Ausreihenmethode `Test<T>` eine `PortNotFoundException` auftreten.

```
void usingPortSet() {
    // generisches PortSet
    var genericPS = new PortSet<int, string>();
    genericPS.Post(42);
    genericPS.Post("Marvin");

    // runtime PortSet Konstruktor mit array of types
    PortSet runtimePS = new PortSet(typeof(int),
                                    typeof(string));
    runtimePS.PostUnknownType(42);
    runtimePS.PostUnknownType("Marvin");

    string who = genericPS; //implicite cast
    int answer = runtimePS.Test<int>();
    Console.WriteLine(who + " says " + answer);
}
```

Listing 2.2: PortSet, einfügen und entnehmen

Klassendiagramme Port, PortSet:

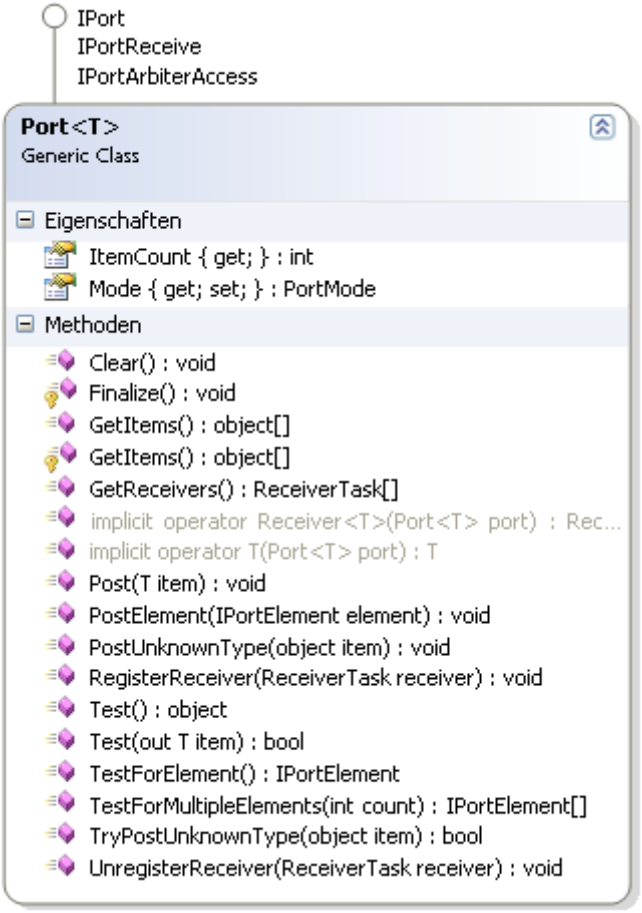


Abbildung 2.2: Port Klassendiagramm

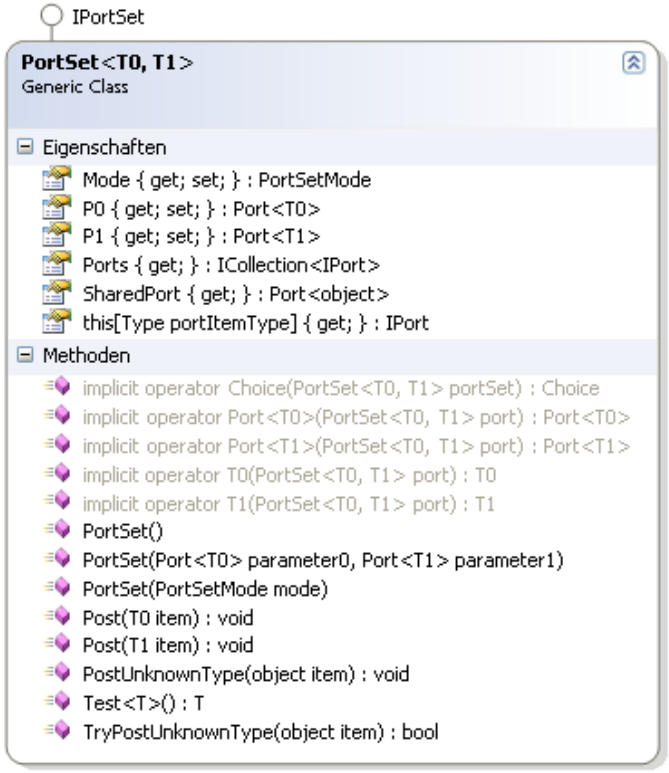


Abbildung 2.3: PortSet Klassendiagramm

2.1.3 Coordination Primitives (Arbiter)

Die statische Klasse Arbiter stellt Factory-Methoden bereit, um CCR Coordination-Primitives zu erstellen. Sie verbinden Ports und Handler und koordinieren die nebenläufige Verarbeitung. Eine der wichtigsten Methoden ist **Arbiter.Activate**. Sie registriert eine Arbiter Instanz an einem Port und assoziiert eine Dispatcher Queue zur Ausführung des Benutzer Codes (Handlers). Die untere Abbildung gibt einen Überblick über die Arbiter, gefolgt von kurzen Beschreibungen. Detaillierte Informationen sind in der MSDN Library Documentation (Microsoft Corporation, 2010b) zu finden.

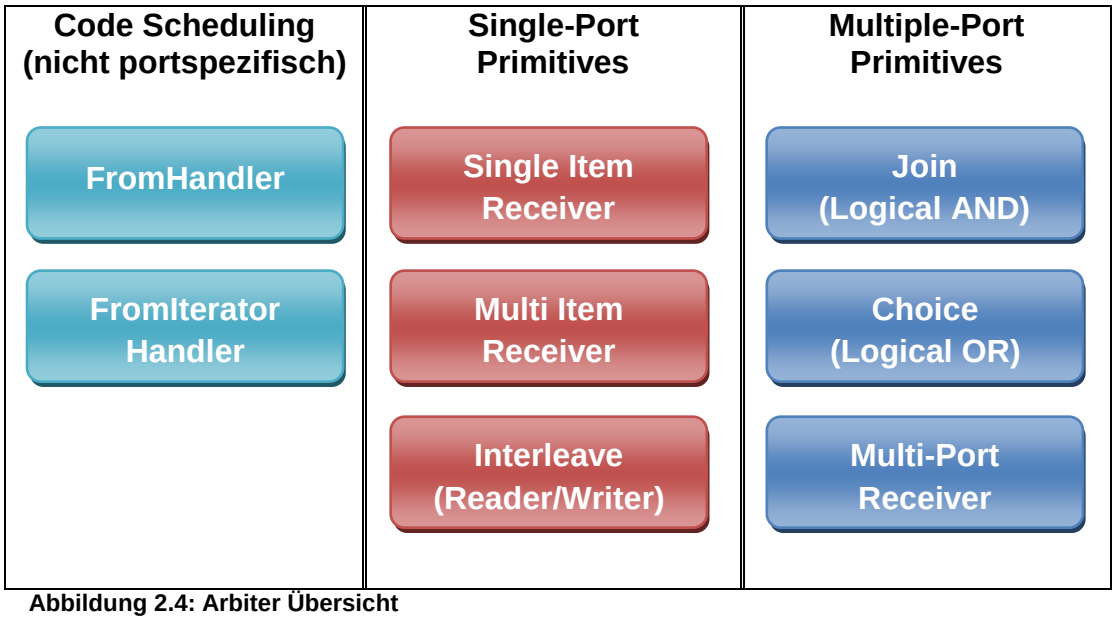


Abbildung 2.4: Arbiter Übersicht

FromHandler:

Führt einen Handler (Benutzer Code) mit dem erstbesten Thread aus.

```
Dispatcher dispatcher = new Dispatcher();
var taskQueue = new DispatcherQueue();

Arbiter.Activate(
    taskQueue,
    Arbiter.FromHandler(
        () => Console.WriteLine("single async MethodCall")
    )
);
```

Listing 2.3: FromHandler Arbiter

FromIteratorHandler:

Verschachtelte und/oder wiederholte asynchrone Ausführung

```
double[] measurements = new double[]{...};
void arbiters() {
    Arbiter.Activate(
        taskQueue,
        Arbiter.FromIteratorHandler(
            new IteratorHandler(myIteratorHandler))
    );
}
IEnumerator<ITask> myIteratorHandler() { // Delegate
    foreach (var value in measurements)
        yield return new Task<double>(value, convertHandler);
}
void convertHandler(double d) { ... }
```

Listing 2.4: FromIteratorHandler Arbiter

Receive (Single Item Receiver):

Assoziiert einen Delegaten/Handler mit einem Port. Der Delegat wird ausgeführt, wenn ein Datum vom Port empfangen wurde. Alle empfangenen Nachrichten werden nebenläufig verarbeitet.

```
Arbiter.Activate(
    taskQueue,
    Arbiter.Receive(
        true,    //persistent
        intPort, //Port
        num => Console.WriteLine("Received " + num)) //Handler
);
```

Listing 2.5: Receive Arbiter

MultipleItemReceive:

Liest N Nachrichten von einem Port aus. Der assoziierte Delegat wird erst nach dem Empfang der spezifizierten Anzahl von N Nachrichten mit diesen ausgeführt.

```
Arbiter.Activate(
    taskQueue,
    Arbiter.MultipleItemReceive(
        false,   //persistent
        intPort, //Port, to rcv from
        42,      // itemCount to rcv
        values => Console.WriteLine("Received " + values.Length)
    )
);
```

Listing 2.6: MultipleItemReceive Arbiter

Interleave:

Konzeptionell dem Reader/Writer-Lock ähnlich werden hier Code-Abschnitte koordiniert. Mit dem Interleave-Receiver lassen sich nicht-präemptive Prozesse verwalten, insbesondere welche Codeabschnitte parallel (vgl. Concurrent Read) bzw. nur exklusiv (vgl. Exclusive Write) ausgeführt werden dürfen.

```
Arbiter.Activate(
  taskQueue,
  Arbiter.Interleave(
    new TeardownReceiverGroup(
      intPort.Receive(i => Console.WriteLine("ShutDown with " + i))
    ),
    new ExclusiveReceiverGroup(
      dblPort.Receive(d => Console.WriteLine("Exclusive computation"))
    ),
    new ConcurrentReceiverGroup(
      intPort.Receive(i => Console.WriteLine("parallel computation")),
      portSet.Receive((int j) => Console.WriteLine("parallel"))
    )
  )
);
```

Listing 2.7: Interleave Arbiter

Join:

Join empfängt von zwei Ports. Unabhängig von der Reihenfolge der eingegangenen Nachrichten, wird der Handler ausgeführt, sobald von jedem Port eine Nachricht vorhanden ist.

```
Arbiter.Activate(
    taskQueue,
    Arbiter.JoinedReceive(
        false,   //persistent
        intPort, //Port 1
        dblPort, //Port 2
        (i, d) => Console.WriteLine("sum is "+ (i+d))
    )
);
// oder auch:
Arbiter.Activate(
    taskQueue,
    intPort.Join( //non-persistent Joined Receiver
        dblPort,
        (i, d) => Console.WriteLine("sum is " + (i+d))
    )
);
```

Listing 2.8: Join Arbiter

Choice:

Sobald an einem beliebigen Port eine Nachricht eingegangen ist, wird der damit assoziierte Handler ausgeführt.

```
Arbiter.Activate(
    taskQueue,
    Arbiter.Choice(
        portSet, //PortSet<int,Exception>
        i => Console.WriteLine("Got a " + i), //intHandler
        e => Console.WriteLine(e.Message)     //ExcHandler
    )
);
```

Listing 2.9: Choice Arbiter

MultiplePortReceive:

Join Receiver für N Ports gleichen Typs.

```
Arbiter.Activate(
    taskQueue,
    Arbiter.MultiplePortReceive(
        false, //persistent
        new Port<int>[] { intPort, intPort2 },
        ints => Console.WriteLine("sum is " + ints[0] + ints[1])
    )
);
```

Listing 2.10: MultiplePortReceive Arbiter

Hinweis:

In den obigen Beispielen wurde intensiv von Lambda-Funktionen (Microsoft Corporation, 2010e) Gebrauch gemacht. Durch diese anonymen Methoden, kann die Logik lokal innerhalb einer Methode gehalten und somit die Lesbarkeit erhöht werden. Links von ‚=>' stehen die Eingangsparameter, rechts die Anweisung(en). Der Rückgabetyp ergibt sich implizit durch das return Statement, beziehungsweise das Fehlen dessen (void).

Arbiter-Klasse:

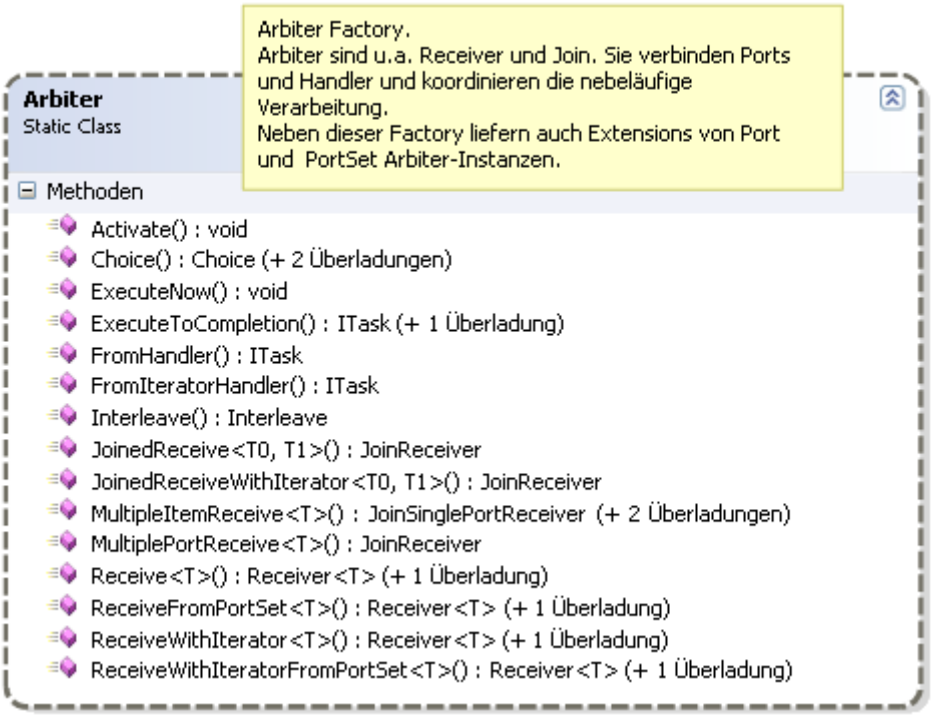


Abbildung 2.5: Arbiter Klassendiagramm.
Übersichtlichkeitshalber eingeschränkte Darstellung ohne Parameter

Port Extensions:

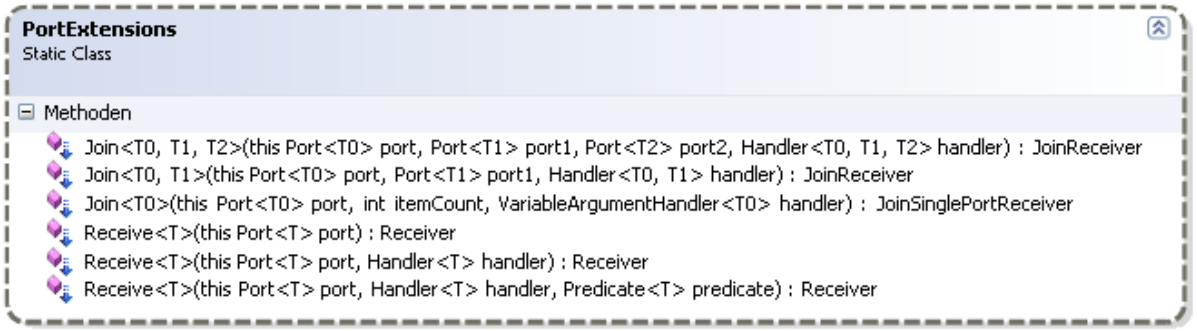


Abbildung 2.6: Port Extension Methods

PortSet Extensions:

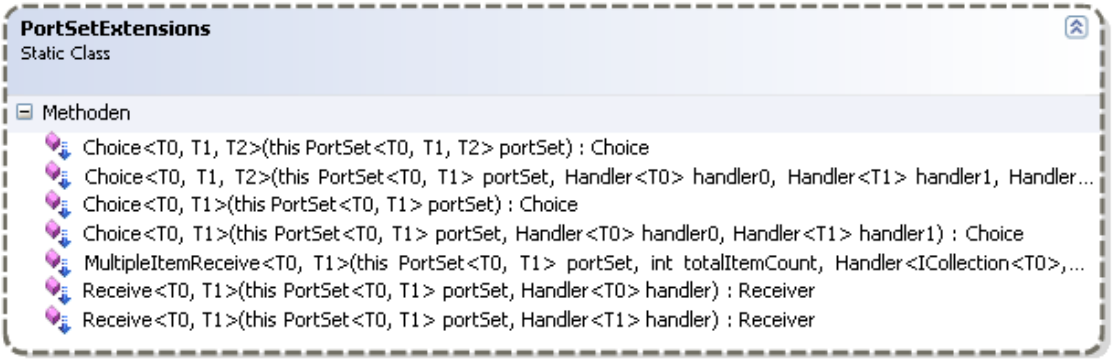


Abbildung 2.7: PortSet Extension Methods

Beispiel mit Single Item Receiver:

Folgendes Beispiel stellt einen Port für string-Nachrichten bereit. An den Port wird ein Receiver gebunden, der die erste Nachricht ausliest und mit einem Handler assoziiert, der den string in Großbuchstaben auf der Konsole ausgibt. Der Handler wird in einem Worker-thread des Dispatchers ausgeführt.

```
void singleItemReceive() {
    //Threadpool mit fixer Größe
    var dispatcher = new Dispatcher(0, "ExecutorPool");

   //                              (friendly name, dispatcher instance)
    var taskQueue = new DispatcherQueue("Queue1", dispatcher);
    var port = new Port<string>();

    //Receiver für port aktivieren und mit taskQueue verbinden
    Arbiter.Activate(
        taskQueue,
        port.Receive(message => Console.WriteLine(message.ToUpper()),//Handler<T>
                     message => message != null)                     //Predicate<T>
        );

    // Nachrichten Posten. Receiver ist bereits aktiv!
    var msgs = new string[] { "Hallo", "schöne", "neue", "Welt" };
    foreach (var str in msgs) {
        Console.WriteLine("Posting \"" + str + "\"");
        port.Post(str);   //Post
        Thread.Sleep(15); //15 ms warten (nur demonstrativ)
    }
}
```

Listing 2.11: vollständiges Beispiel zu Port, Arbiter, Dispatcher

Jede CCR-Anwendung benötigt einen Pool von Workerthreads, den **Dispatcher**. Dabei sollte aus Performanzgründen je Anwendung und Rechner nur ein Threadpool vorhanden sein. In diesem Beispiel wird der Dispatcher mit dem Default-Parameter 0 initialisiert. Das

heißt, es werden so viele Threads wie Cores, mindestens aber zwei, angelegt. **DispatcherQueue**s, welche über deren Konstruktor an den Dispatcher gebunden werden, führen dem Dispatcher neue Task zu. Es können theoretisch unbegrenzt viele Queues erzeugt werden, wobei jede Queue gleichberechtigt ist; der Dispatcher wählt nach dem Round-Robin Verfahren. Die Erweiterungsmethode **Port.Receive** liefert einen an den Port gebundenen Single Item Receiver. Als Parameter werden ein **Handler**-Delegat für die neben-läufige Verarbeitung der Nachrichten und (optional) ein **Predicate**-Delegat übergeben. Letzteres stellt eine Bedingung (condition) dar, die eine Nachricht erfüllen muss, um in einer Task übernommen zu werden (hier: not null). In diesem Beispiel wurden keine konkreten Delegatinstanzen (Methoden), sondern Lambda-Funktionen (Microsoft Corporation, 2010e) benutzt. Dieses Feature wird seit .Net 3.0 unterstützt. Aus einer empfangenen Nachricht und dem Handler erzeugt der Receiver eine Task, die über die *taskQueue* zur Verarbeitung an den Dispatcher weitergereicht wird. Wie eingangs beschrieben, wird durch **Arbiter.Activate** der Receiver aktiviert und mit einer DispatcherQueue assoziiert.

Man beachte, dass der Receiver in diesem Beispiel nur für ein Element aktiv ist. Es wird nur „Hallo“ verarbeitet, die übrigen geposteten Elemente verbleiben in der Queue. Ein persistenter Receiver ist neben weiteren Beispielen im Unterabschnitt „Code Beispiele zu Coordination Primitives“ aufgeführt.

2.1.4 Task Scheduling

Das folgende Sequenzdiagramm veranschaulicht etwas detaillierter die Vorgänge in der Port-Implementierung, wenn eine Nachricht versendet wurde. Das Diagramm ist nach einer eigenen Interpretation der MSDN-Dokumentation zu „CCR Task Scheduling“ (Microsoft Corporation, 2010b) entstanden. Die tatsächliche Implementierung kann davon abweichen.

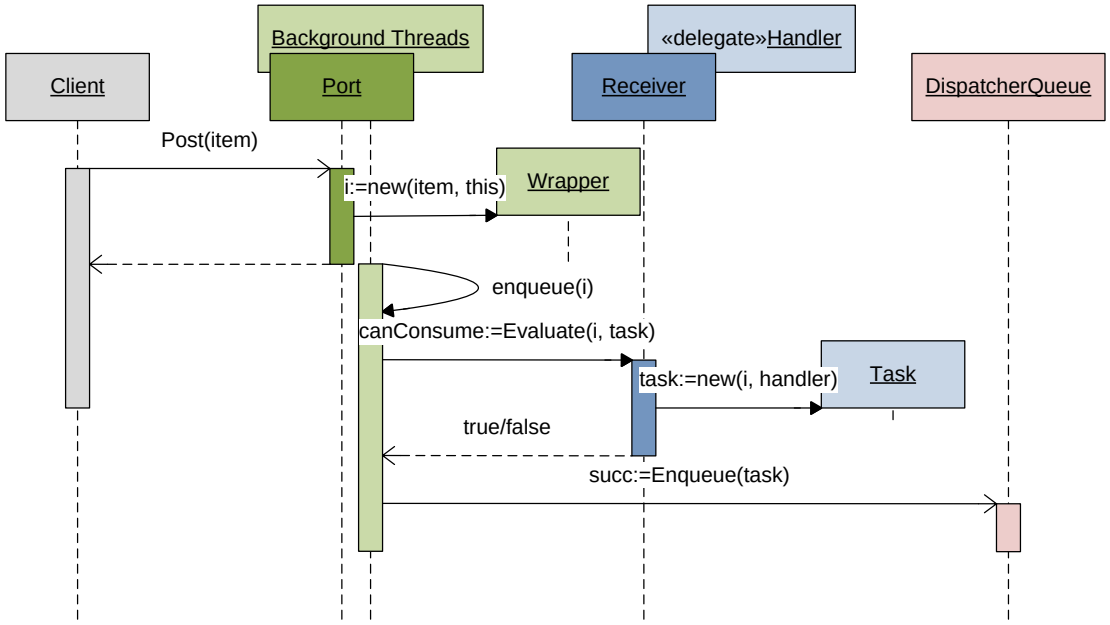


Abbildung 2.8: Task Scheduling

Ein `Post` ist eine asynchrone Operation, bei der der Kontrollfluss so bald wie möglich an den Aufrufer zurückgeht. Die Nachricht wird in der Wrapper-Klasse PortElement zusammen mit einem Verweis auf den Port selbst verpackt und intern eingereiht. Die registrierten Receiver werden „gefragt“, ob sie das Element verarbeiten können. Dies ist dann der Fall, wenn der Nachrichtentyp passend, die condition erfüllt und ein geeigneter Handler registriert ist. Bei Erfolg liefert `Receiver.Evaluate` true sowie eine neue Task, bestehend aus dem PortElement und Handler, zurück. Können mehrere Receiver die Nachricht verarbeiten, gewinnt der Schnellste. Anschließend wird die Task in diejenige DispatcherQueue eingereiht, welche bei der Aktivierung des Receivers assoziiert wurde.

2.1.4.1 TaskExecutionPolicy

Das System eines autonomen Rescue Robots besteht typischerweise aus einer Vielzahl von Regelungs- und Verhaltenskomponenten. Im CCR-Kontext bedeutet dies, dass Tasks aus diversen, mehr oder weniger gleichpriorisierten, Quellen erzeugt werden, angestoßen durch das Versenden von Nachrichten. Es ist offensichtlich sinnvoll, für verschiedene Komponenten auch unterschiedliche DispatcherQueues zu benutzen. So kann verhindert werden, dass seltene oder unregelmäßige Tasks, wie beispielsweise die Reaktion auf einen Kontaktsensor, verhungern, weil diverse Tasks der Abstandssensoren oder inversen Kinematik noch weiter vorne in der Queue eingereiht sind.

Dieses beispielhafte Szenario veranschaulicht aber auch, dass auf Nachrichtenfluten und kritische Zeitintervalle adäquat reagiert werden sollte. Ein Mechanismus in CCR, solchen Anforderungen gerecht zu werden, ist die **TaskExecutionPolicy**, die auf DispatcherQueues angewandt werden kann. Mittels einiger vordefinierter Strategien lässt sich u.a. die Größe der Queue, eine mittlere Scheduling-Rate oder der Umgang mit Nachrichtenfluten festlegen.

Folgende Tabelle listet die Strategien des Aufzählungstyps **TaskExecutionPolicy** auf, welche dem Konstruktor einer Dispatcherqueue übergeben werden können.

Name	Beschreibung
Unconstrained	Default behavior, all tasks are queued with no constraints
ConstrainQueueDepth DiscardTasks	Queue enforces maximum depth (specified at queue creation) and discards tasks enqueued after the limit is reached
ConstrainQueueDepth ThrottleExecution	Queue enforces maximum depth (specified at queue creation) but does not discard anny tasks. It forces the thread posting any tasks after the limit is reached, to sleep until the queue depth falls below the limit
ConstrainSchedulingRate DiscardTasks	Queue enforces the rate of task scheduling specified at queue creation and discards tasks enqueued after the current scheduling rate is above the specified rate
ConstrainSchedulingRate ThrottleExecution	Queue enforces the rate of task scheduling specified at queue creation and forces the thread posting tasks to sleep until the current rate of task scheduling falls below the specified average rate

Tabelle 2.3: TaskSchedulingRate, (Microsoft Corporation, 2010g)

2.1.5 Exception Handling

Die bekannte, strukturierte Fehlerbehandlung mittels try/catch/finally ist nur im threadlokalen Kontext anwendbar. Bei nebenläufigen und insbesondere verteilten Prozessen ist dieser Mechanismus ungenügend. Da CCR auf Interprozesskommunikation via Nachrichtenübermittlung basiert, ist eine weitere exklusive Methode der Ausnahmebehandlung naheliegend: Fehler sind Nachrichten. So kann beispielsweise bei Zwei-Wege-Kommunikationen nach dem Request/Response Prinzip ein Antwortport für Fehlermeldungen eingerichtet werden, in welchen Workerthreads auftretende Fehler weiterreichen können.

Darauf aufbauend kommt noch eine Besonderheit von CCR hinzu, nämlich eine implizite Fehlerbehandlung durch Kausalitäten. Versendete Nachrichten oder ausgeführte Delegaten werden hier mit einem Kausailtäts-Kontext verbunden, der einen Ausnahmeport beinhaltet. Tritt ein Fehler in einem beliebigen beteiligten Handler auf, wird dieser vom CCR-Scheduler implizit an den Exception-Port weitergereicht. Dies stellt eine Multi-Threaded-Version der strukturierten Ausnahmebehandlung dar, die beliebig weit verschachtel und verteilt sein kann.

2.1.5.1 Explizite Fehlerbehandlung

In Objekt Orientierten Sprachen wie C# oder Java werden Ausnahmen meist solange im Stack Trace weitergereicht, bis sie in einem try/catch-Block behandelt werden. Dies zwingt den Programmierer, sich mit Fehlern auseinander zu setzen und ermöglicht ihm, geeignete Maßnahmen zur Sicherung der Programmstabilität zu ergreifen. Diese Methode ist über Thread- oder gar Maschinengrenzen nicht möglich.

Dieselbe Methodik lässt sich aber im Rahmen des Message Passing auch in CCR umsetzen. Innerhalb einer Task, beziehungsweise dessen Handler, ist der Einsatz von try/catch oder die Erzeugung selbstdefinierter Fehler weiterhin möglich. Sie müssen aber nicht zwangsweise behandelt, sondern können über Exception-Ports weitergereicht und dann von einem separaten Exception-Handler bearbeitet werden, auch über Thread- und Maschinengrenzen hinaus. Bei DSS-Services (siehe nächstes Kapitel), wo PortSets für Request/Response-Nachrichten dienen, wird häufig davon Gebrauch gemacht. Typischerweise antwortet eine Service-Methode mit Success oder Failure. Die CCR-Arbiter wie Choice oder MultipleItemReceive bieten dazu bequeme Methoden, darauf zu reagieren.

Beispiel:

```
internal void ExclusiveHandlingExample() {
    Port<int> request = new Port<int>();
    PortSet<string, Exception> response = new PortSet<string,Exception>();

    request.Post(1);
    request.Post(0);

    Arbiter.Activate(
        _taskQueue,
        //Handle Request
        Arbiter.Receive<int>( true, request,
            i => {
                if (i != 0)
                    response.Post(i.ToString());
                else
                    response.Post(new ArgumentException("Zero not allowed"));
            }
        ),
        //Handle either Success or Failure Response
        Arbiter.Choice<string, Exception>(
            response,
            result => Console.WriteLine("Success Result:" + result),
            ex => Console.WriteLine("Failure: " + ex.Message)
        )
    );
}
```

Listing 2.12: Exklusive Fehlerbehandlung mit Exception-Port und Choice

2.1.5.2 Implizite Fehlerbehandlung (Causality)

Kausalität bezeichnet die Beziehung zwischen Ursachen und Wirkung. Sie spielt unter anderem bei verteilten Transaktionen eine große Rolle, wo Ereignisse beispielsweise durch logische Uhren und einer „happen before“ Relation in eine kausale Ordnung gebracht werden. Bezogen auf die Fehlerbehandlung verursacht zum Beispiel ein Methodenaufruf eine Ausnahme in einer Subroutine (Wirkung). Die CCR-Kausalität dient allein diesem Aspekt und hat mit Transaktionen gemeinsam, dass eine Folge von Operationen als eine Einheit betrachtet wird, da die Operationen in einem logischen Zusammenhang stehen, dem Kausalitätskontext.

Ein **Causality** Objekt beinhaltet einen Exception-Port. Dieser wird vom CCR-Scheduler benutzt, um unbehandelte Ausnahmen im Kausalitätskontext implizit weiterzureichen. Ein auf dem Port aktivierter Arbiter behandelt die Ausnahmen. Der Kausalitätskontext umfasst alle Ausführungen, prozedural, asynchron oder verteilt, die ab dem Zeitpunkt des Hinzufügens der Kausalität im logischen Zusammenhang zum Prozess stehen, der die Kausalität erstellt

hat. Es werden also alle Subroutinen, asynchrone Aufrufe oder per `Post()` übermittelte Items mit dieser Kausalität verknüpft. Sie wird bei jeder Verschachtelung oder Verzweigung weitergereicht, sodass der Kontext beliebig weit verschachtelt oder verzweigt sein kann. Die Kausalität „fließt“ mit und an einem beliebigen Zeitpunkt und Ort kann der CCR-Scheduler oder ein Handler auf den Exception-Port zugreifen.

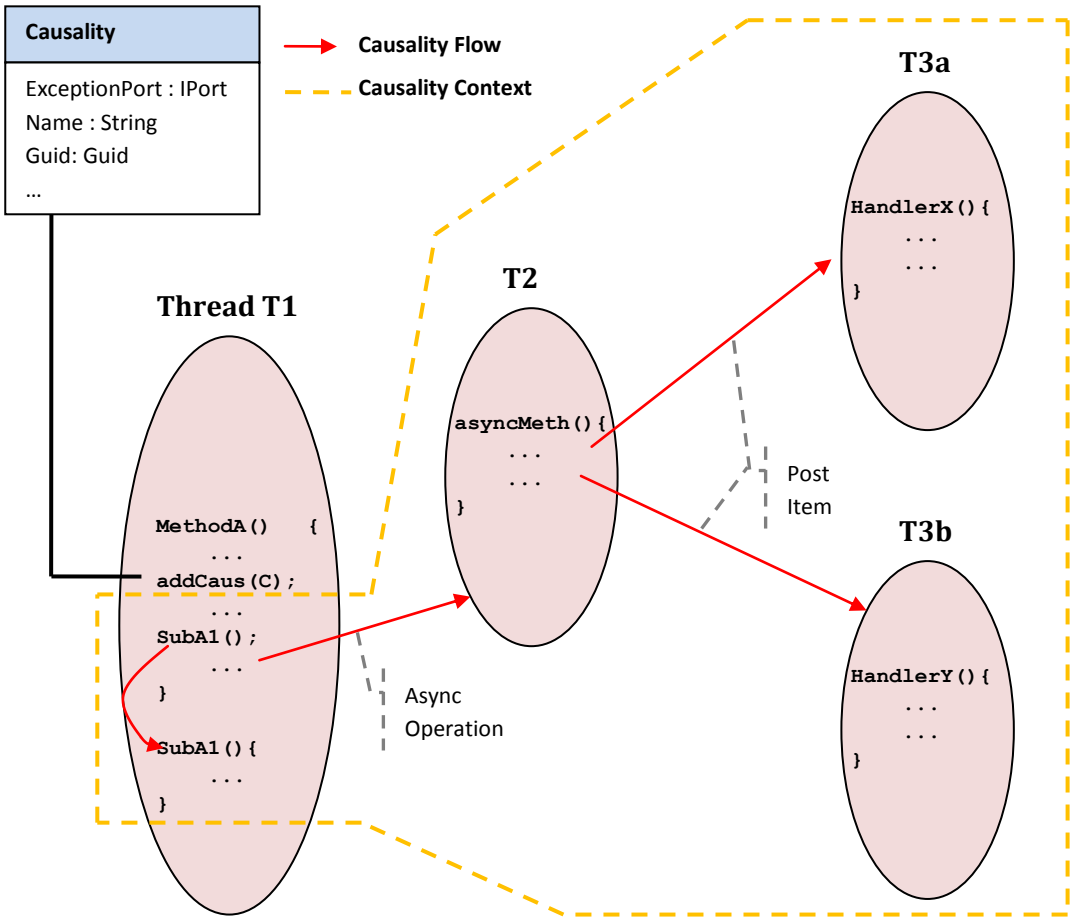


Abbildung 2.9: Kausalitäts-Fluss und -Kontext

Kausalitäten können verschachtelt werden und auch das Zusammenfügen von Kausalitäten wird unterstützt. Treffen beispielsweise über einen Join-Arbiter zwei Items mit ihrer jeweils eigenen Kausalität ein, wird eine Ausnahme auf beiden Exception-Ports veröffentlicht.

Auf nähere Details soll an dieser Stelle aber nicht eingegangen werden. Denn bei Roboteranwendungen mit DSS-Services ist meist die exklusive Ausnahmebehandlung eher angebracht, da sie unter anderem den Entwickler zu einer expliziten Beschäftigung mit möglichen Fehlern ermutigen. Diese Einschätzung wird dadurch bestätigt, dass in den

diversen Beispielen und Tutorials zu DSS die Causalities praktisch nicht vorzufinden sind. Das folgende Beispiel zeigt eine einfache Verwendung von Causality.

```
internal void SimpleCausalityExample() {

    var intPort = new Port<int>();
    var excPort = new Port<Exception>();
    var causality = new Causality("friendlyName", excPort);

    // activate failure handler for the causality
    Arbiter.Activate(_taskQueue,
        Arbiter.Receive(false, excPort, exc => Console.WriteLine(exc.Message)
    ));

    Dispatcher.AddCausality(causality); // add causality to the *current thread*

    // any unhandled exception from this point on, in this method or
    // any delegate that executes due to messages from this method,
    // will be posted on excPort.

    Arbiter.Activate(_taskQueue,
        Arbiter.Receive(false, portInt, i => 10/i)
    );

    intPort.Post(0); //provoke Exception in Handler above
}
```

Listing 2.13: Causality, Codebeispiel

2.1.6 Code Beispiele zu Coordination Primitives

Folgend ist die Verwendung von zwei der in DSS gängigsten Arbiter beispielhaft dargestellt. Der IteratorHandler ist die Standardvariante für Port-Handler in DSS und ein Receiver wird häufig für Notifications oder interne Posts benutzt. Die Beispiele sind auch ohne Kenntnisse aus dem DSS-Kapitel nachzuvollziehen.

2.1.6.1 Persistenter Receiver

Die statische Methode Arbiter.Receive liefert einen Single Item Receiver. Der erste bool'sche Parameter gibt an, ob der Receiver persistent ist, also ob er aktiv bleibt und für jede Message im Port eine Task erzeugt. Port und Handler müssen angegeben werden, der letzte Parameter vom Typ Predecate<T> ist optional. Die Ausgabe der WorkerThreads ist „HALLO", SCHÖNE", „NEUE", „WELT", allerdings nicht zwangsläufig in dieser Reihenfolge. Zur Erinnerung, es handelt sich um asynchrone, nebenläufige Methoden.

```
void ReceiveWithPersistentHandler() {

    var dispatcher = new Dispatcher(0, "ExecutorPool");
    var taskQueue = new DispatcherQueue("Queue1", dispatcher);
    var port = new Port<string>();
    var msgs = new string[] { "Hallo", "schöne", "neue", "Welt" };

    Arbiter.Activate(
        taskQueue,
        Arbiter.Receive<string>(
            true, //persistent?
            port, //Port to receive from
            msg => Console.WriteLine(msg.ToUpper()), //Handler
            msg => msg != null                       //condition
            )
        );

    foreach (var str in msgs) {
        Console.WriteLine("Posting \"" + str + "\"");
        port.Post(str);   //Post
        Thread.Sleep(15); //15 ms warten
    }

    Thread.Sleep(1000);
    dispatcher.Dispose();
}
```

Listing 2.14: persistenter Receive-Arbiter

2.1.6.2 Iterators

Iterators in CCR sind Kollektionen von Tasks, die nach benutzerdefinierter Reihenfolge und Art generiert werden. Die Generierung und Verarbeitung der Task ist weiterhin asynchron.

Seit C# 2.0 sind Iterators ein Feature der Sprache, die eine foreach-Iteration durch eine Klasse oder Struktur ermöglicht. Ihnen ist gemein, dass sie ein **IEnumarator<T>** zurückgeben. Anhand dieser Signatur erkennt der Compiler, dass es sich um einen Iterator handelt und im Methodenrumpf ein yield statement zu erwarten ist. Ein **yield return** liefert das nächste Element, **yield break** beendet die Iteration vorzeitig. Der Compiler generiert daraus einen Zustandsautomaten mit den von IEnumerator<T> definierten Methoden wie *Next* und *Current*. Der Vorteil für den Programmierer ist, dass er das Verhalten des Iterators in sequenzieller Weise beschreiben kann, ohne dass der Iterator blockierend ist. Es müssen nicht mehr umständlich die diversen Methoden eines IEnumerator implementiert werden.

Die Klasse **IterativeTask** ist ein Arbiter, der mit einem **IterativeHandler**-Delegat registriert wird. Es werden von ihm eine Reihe von Tasks erzeugt, die nebenläufig bearbeitet werden. Statt nach dem herkömmlichen asynchronen Programmiermodell (APM) eine Reihe von verschachtelten Callbacks zu implementierten, kann hier alles in einer Methode in sequenzieller Schreibweise codiert werden. Auf diese Weise lassen sich Listen durchlaufen, die Antworten mehrerer zusammenhängender Requests asynchron bearbeiten (bspw. bei verteilten Algorithmen) oder verschachtelte asynchrone Prozesse realisieren.

Im folgenden Beispiel sollen sukzessive Strings aus dem Port ausgelesen und zusammengesetzt werden. Dazu wird ein generischer IterativHandler mit dem Parametertyp Port<String> implementiert. Dieser bekommt den Port übergeben und liest ihn aus. Dafür wird die Extension Methode `Port.Receive` benutzt, die einen Receiver liefert. Alle Arbiter implementieren ITask, daher kann der Receiver mit `yield return` zurückgegeben werden. Hier blockiert der Handler logisch. Tatsächlich wird der WorkerThread für neue Aufgaben wieder freigegeben. Wenn der Receiver ein Element empfangen hat, wird der Code in der nächsten Zeile fortgesetzt (womöglich in einem anderen WorkerThread). Erst jetzt wird per implizitem cast die Nachricht aus dem Port konsumiert (dem erzeugten Receiver ist kein Handler zugewiesen, daher verbleibt die Nachricht im Port). Anschließend wird in einer Schleife noch zweimal per `yield return` eine Task erzeugt und zwei weitere Elemente ausgelesen, um sie aneinander zu hängen. Man beachte diese Besonderheit: Es können Schleifen um asynchrone Aufrufe herum implementiert werden. In der Hauptmethode `iterativeTaskScheduling` wird eine neue IterativeTask aktiviert, bei der der IterativeHandler registriert wird. Zwei Coding-Varianten sind darunter auskommentiert zu finden. Bei der ersten wir der Handler durch ein Lambda-Funktion aufgerufen, bei der zweiten wird die IterativeTask durch `Arbiter.FromIterativeHandler` erzeugt.

Die Tasks erzeugen folgende Ausgabe auf der Konsole:

```
Hallo
Hallo schöne
Hallo schöne neue
```

```
static void iterativeTaskScheduling() {

    var dispatcher = new Dispatcher(0, "ExecutorPool");
    var taskQueue = new DispatcherQueue("Queue1", dispatcher);
    var port = new Port<string>();
    var msgs = new string[] { "Hallo", "schöne", "neue", "Welt" };

    //IEnumerator<ITask> IteratorHandler()
    Arbiter.Activate(
        taskQueue,
        new IterativeTask<Port<string>>(port, MyIteratorHandler)
        //new IterativeTask( () => MyIteratorHandler(port) )
        //Arbiter.FromIteratorHandler( () => MyIteratorHandler(port) )
        );

    foreach (var str in msgs) {
        Console.WriteLine("Posting \"" + str + "\"");
        port.Post(str);   //Post
        Thread.Sleep(10); //10 ms warten
    }

    Thread.Sleep(1000);
    dispatcher.Dispose();
}

//Der Compiler erzeugt hieraus eine State machine:
public static IEnumerator<ITask> MyIteratorHandler(Port<string> port) {
    var msg = "";
    yield return port.Receive(); //blockiert "logisch"
    //nach erfolgtem Receive: hier weiter mit neuer Task
    msg += (string)port; //consume PortItem, append to msg
    Console.WriteLine(msg);
    for (int i = 0; i < 2; i++) {    //asynchrone Schleife!!
        yield return port.Receive();
        msg += " "+(string)port;
        Console.WriteLine(msg);
    }
}
```

Listing 2.15: IterativeTask und IteratorHandler

2.1.7 Performance

Vorrangig versprechen die Entwickler der CCR-Library eine einfachere Handhabung asynchroner und nebenläufiger Prozesse. Aber auch eine gute Skalierbarkeit und Performanz gilt als ein Aushängeschild ihres Modells. Um dies zu untersuchen, wurden einige simple Tests durchgeführt, die einen direkten Vergleich verschiedener CCR-Arbiter zum Asynchronen Programmier Modell (APM) von .NET (Microsoft Corporation, 2010a) ermöglichen sollen.

2.1.7.1 Testaufgabe

Es sollen direkt aufeinanderfolgend N asynchrone Berechnungen angestoßen und die ganzzahligen Ergebnisse aufaddiert werden. Die aufgerufene Methode bekommt eine Fließkommazahl (double) übermittelt, welche zu einer Ganzzahl umgewandelt wird. Zusätzliche 10.000 Additionen von +/- 1 simulieren eine aufwändige Berechnung. Bei der APM-Variante summiert eine Callback-Funktion die N Resultate, bei CCR sind das entsprechende Handler oder Code-Abschnitte, abhängig von den benutzten Arbitern[8].

2.1.7.2 Testausführung

Die Tests wurden mehrfach und auf drei verschiedenen Systemen ausgeführt: ein WinXP-Rechner mit single-core bei 1,3 GHz; ein Win7-Rechner mit dual-core bei 2,5 GHz; ein Win7-Rechner mit quad-core bei 3,2 GHz. Der Testprozess beanspruchte maximal 30 MB Arbeitsspeicher, der CCR-Dispatcher lief stets mit der Standardkonfiguration (ein Worker-Thread pro CPU). Das Testprojekt wurde mit .NET 3.5 ohne CPU-spezifische Optimierung kompiliert und lief jeweils mit der CLR 2.0.

[8] Das Testprojekt (Code) ist auf der beigefügten CD

2.1.7.3 Testergebnis

Für alle Varianten stieg die Ausführungszeit wie erwartet linear mit der Anzahl N von asynchronen Aufrufen. Das nebenstehende Diagramm (Abbildung 2.10) für 1.000 Aufrufe ist daher repräsentativ. Es zeigte sich, dass alle Varianten sehr gut mit der Anzahl von Prozessoren skalierten. Außer bei der ersten Version des IterativeHandler waren alle Kerne stets unter Volllast.

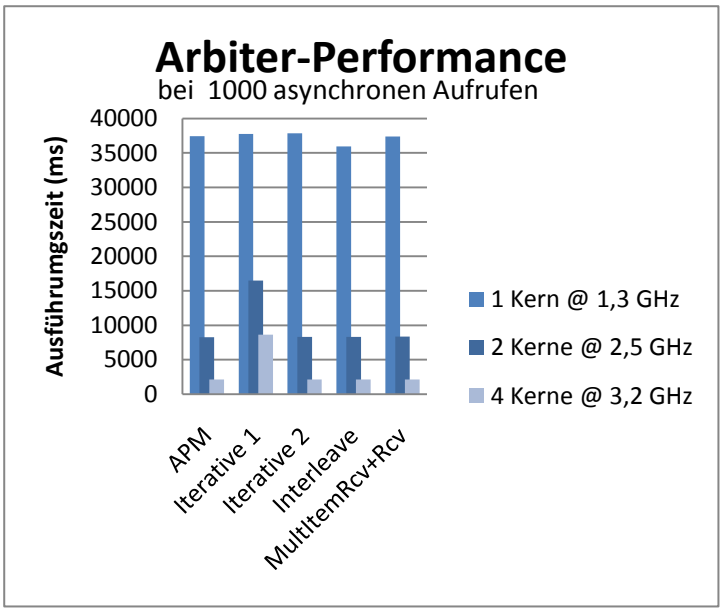


Abbildung 2.10: Arbiter Performance Diagramm (Testergebnisse)

Tatsächlich schien die Version „Iterative 1“ nur single-threaded zu laufen, was sich sowohl bei der Ausführungszeit, als auch bei der CPU-Last zeigte. In der Analyse stellte sich heraus, dass dies an der Implementierung lag, die intern mit Receive-Arbitern arbeitet: In einer Schleife werden nacheinander Tasks erzeugt, die einen Double-Wert empfangen, die asynchrone Berechnung anstoßen, das Zwischenergebnis empfangen und schließlich aufsummieren. Dies wirkte sich negativ auf die Nebenläufigkeit aus, da immer erst dann neue Tasks erzeugt wurden, wenn ein Zwischenergebnis vorlag, es wurden vorher keine weiteren Berechnungen angestoßen. Die Variante „Iterative 2“ benutzt intern stattdessen Choice-Arbiter.

2.1.7.4 Testbeurteilung

In der Ausführungszeit sind, abgesehen von „Iterative 1“, keine nennenswerten Unterschiede festzustellen. Die Behauptung, CCR sei meist auch schneller, kann der Test so nicht bestätigen. Aber zumindest ist das CCR-Modell auch nicht langsamer. Da der Test relativ simpel gehalten ist, ist auch der Programm-Code relativ überschaubar, gut zu lesen und zu warten, für alle Varianten. Allerdings wurde hier nur der Vergleich zu APM aufgestellt. Unter speziell diesem Aspekt, kann man auch nur bedingt zustimmen, dass der Code mit CCR „einfacher“ wird. Die Variante „Iterative 1“ hat sich sogar als eher negativ für CCR herausgestellt, da man hier durch die Verwendung eines gängigen CCR-Patterns deutliche Einbuße bezüglich der Parallelität hat.

Meine persönliche Gesamtbeurteilung ist daher, dass Microsoft mit dem CCR-Modell zwar keine „Wunderwaffe“ geschaffen hat, es aber sehr wohl das Potential hat, die Handhabung komplexerer verteilter Anwendungen zu vereinfachen. Das nächste Kapitel, Decentralized Software Services, wird dies zeigen.

2.2 Decentralized Software Services (DSS)

Im vorangegangen Kapitel wurde das Programmiermodell CCR vorgestellt und gezeigt, wie diese Bibliothek genutzt werden kann, um asynchrone Operationen zu verwalten, nebenläufige Verarbeitung zu koordinieren und mit partiell auftretenden Fehlern umzugehen. Das auf Message-Passing basierende Konzept ermöglicht eine Software-Architektur von lose gekoppelten, fein granulierten Komponenten und ist die Basis für das in Robotics Studio verwendete Architekturmodell, Decentralized Software Services (DSS).

Gemäß diverser Beschreibungen orientiert DSS sich an der Service Oriented Architecture (SOA), wobei es allerdings zu bedenken gilt, dass SOA eher ein Marketingbegriff ist, als ein gemeingültiges, eindeutig definiertes Architekturmuster. OASIS definierte SOA folgend: “Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. … However, service, as the term is generally understood, also combines the following related ideas:

- The capability to perform work for another
- The specification of the work offered for another
- The offer to perform work for another”

(OASIS;, 2006)

Im DSS-Kontext ist ein Service eine autonome Software-Komponente in einer verteilten Anwendung mit einem einheitlichen Aufbau. Sie besitzt eine Identität, einen Zustand und bietet Aktivitäten (Operationen) an. Ports bilden die Kommunikationsschnittstelle, über die Nachrichten zwischen den Services ausgetauscht werden. Es gibt drei Kategorien von Nachrichten: Die Anforderung, eine Aktivität auszuführen, die Antwort darauf und Benachrichtigungen über Zustandsänderungen. DSS bedient sich dabei etablierter Netzwerktechnologien. So können Services einer Anwendung lokal oder über ein Netzwerk (LAN, Internet,…) verteilt sein. Das Kommunikationsprotokoll DSSP erweitert HTTP um Operationen wie beispielsweise UPDATE oder REPLACE zur Manipulation des Zustands.

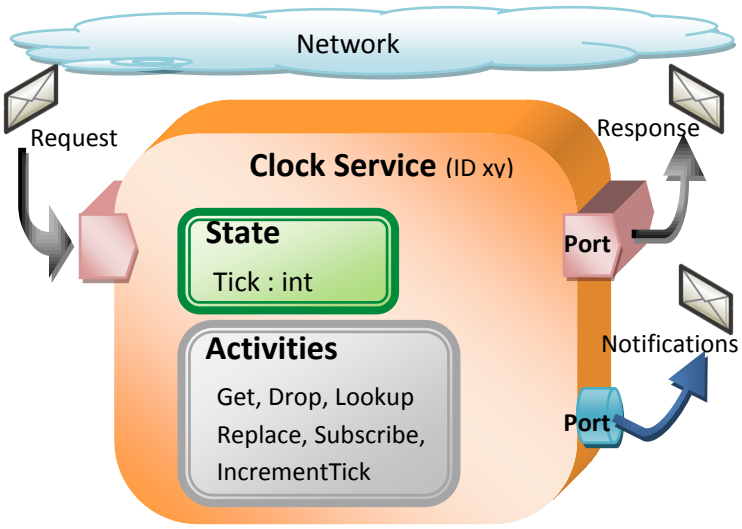


Abbildung 2.11: logischer Aufbau eines DSS Service‘

2.2.1 DSS für Roboteranwendungen

Ziel des Designprozesses mit SOA im Allgemeinen und DSS im Speziellen ist es, das Verhalten und die Funktionalitäten in einer konkreten Anwendungsdomäne soweit zu entkoppeln, dass weitgehend isolierte Komponenten in Dienste gekapselt werden können, die dann wiederum zu höheren Diensten mit komplexerem Verhalten kombiniert werden (Orchestration).

Dass eine solche Architektur auch für Roboteranwendungen vorteilhaft ist, ist naheliegend. Denn hier hat man es typischerweise nicht nur konzeptionell sondern per se mit physisch getrennten und eigenständigen Komponenten zu tun. Sensoren liefern Daten über die Umwelt,

eine Software interpretiert diese, um dann mittels Aktoren wie einem Servomotor das gewünschte Verhalten zu realisieren.

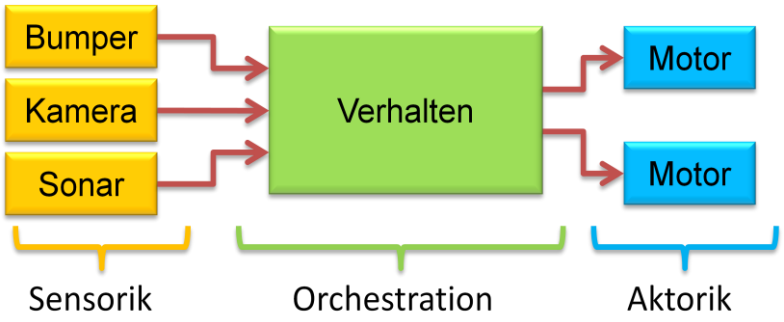


Abbildung 2.12: Roboteranwendungen, Orchestration

Ein DSS-Service repräsentiert somit beispielsweise einen einzelnen Kontaktsensor oder den Motor des linken Rades. Der Zustand des Motorservice kann dann die aktuelle Drehzahl und die Information, ob der Motor ein- oder ausgeschaltet ist, sein. Durch Aktivitäten, die der Motorservice als Dienst anbietet, kann dessen Zustand abgefragt oder manipuliert werden (GibZustand, MotorEin, StelleDrehzahl,…). Seine Identität unterscheidet ihn vom gleichen Service für das rechte Rad.

Nach dem in Abbildung 2.12 dargestellten Beispiel, gäbe es je einen Service für die drei Sensoren, zwei Motorservices und einen Service, der das logische Verhalten realisiert. Per Notification erhält der Verhaltensdienst asynchron neue Sensordaten (Polling per Request des Sensorzustands wäre auch möglich), die von ihm ausgewertet werden, woraufhin die Motoren mittels Request aktuelle Stellwerte übermittelt bekommen. Der Motor antwortet beispielsweise mit einer Success/Failure-Response.

2.2.2 DSS Service Aufbau

Während eingangs dieses Kapitels der prinzipielle, logische Aufbau eines DSS-Services umrissen wurde, werden folgend einige Implementierungsdetails erläutert, die für die Entwicklung eigener Services in C# erforderlich sind. Mit der Installation von Robotics Studio erhält die IDE Visual Studio ein neues Project-Template für DSS-Services. Legt man ein neues Projekt dieses Typs an, werden automatisch alle grundlegenden Klassen und Implementierungen eines einfachen Service erstellt. Ein solches (Default-)Projekt wird hier ebenfalls zur Erläuterung herangezogen.

Die Abbildung 2.13 zeigt eine Übersicht der Komponenten, die jeden Services gemein sind. Das untere Klassendiagramm zeigt die Implementierung direkt nach dem Anlegen eines neuen DSS-Projekts. ExampleService hat die Dienste ServiceTutorial4 und Subscription-Manager als Partner. Näheres zu Partnern und dem Publisher-Subscription Modell folgt weiter unten. Ports und Handler sind diejenigen aus der CCR-Library.

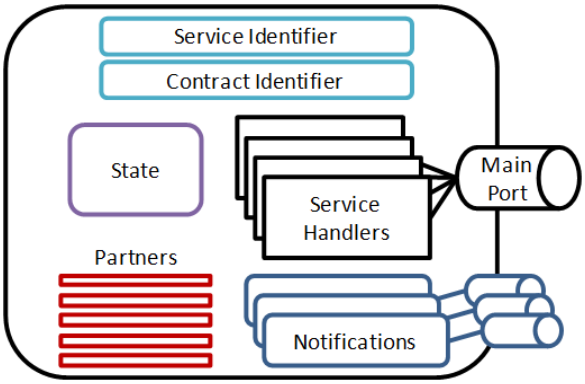


Abbildung 2.13: DDS-Modell (Microsoft Corporation, 2010c)

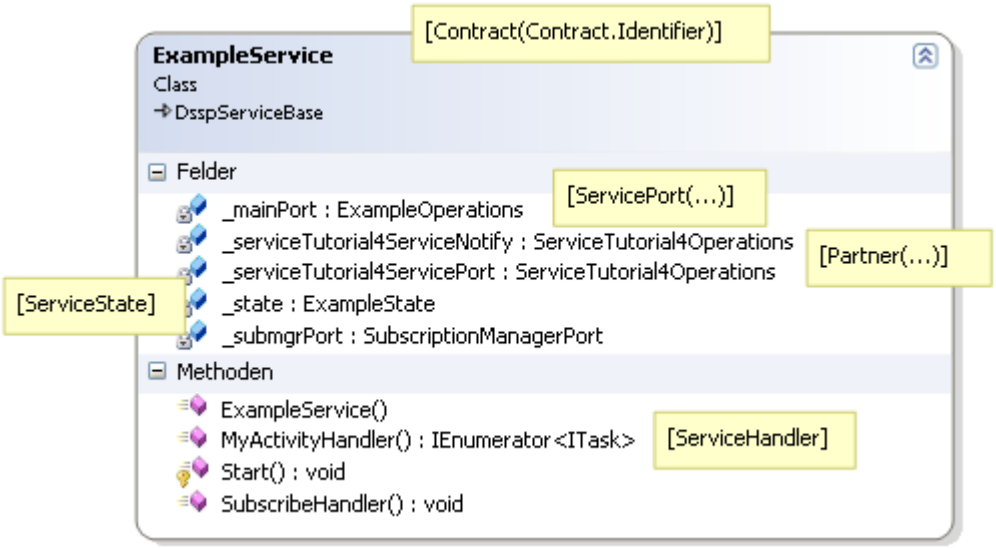


Abbildung 2.14: Klassendiagramm eines DSS-Service

2.2.2.1 Service Identifier

Die Laufzeitumgebung für Services ist der sogenannte DSS-Node, der auch als Webserver fungiert. Der Service Identifier ist ein URI mit dem eine konkrete Instanz eines Service auf einem DSS-Node adressiert werden kann. Der URI wird dynamisch beim Start eines Services zugewiesen und hat allgemein den Aufbau:

`http://machine:port/ServiceName`

Der Identifier kann auf verschiedene Weise festgelegt werden:

1. **ServicePort-Attribut**
   Der Main-Port des Service wird mit diesem Attribut und einen Namen als Parameter versehen.

```
[ServicePort("/Example", AllowMultipleInstances = true)]
ExampleOperations _mainPort = new ExampleOperations();
```

   Der URI lautet dann bspw. `http://localhost:50000/Example`.
   Wenn, wie in diesem Beispiel, mehrere Instanzen eines Service existieren können, erweitert sich die Adresse um einen dynamisch erzeugten Globally Unique Identifier (GUID):
   `http://localhost:50000/Example/20d941cf-3312-20c6-b445-0d1dfd805001`

2. **Manifest**
   Wird ein Service mit einem Manifest gestartet und enthält es einen Instanz-Namen, so wird dieser genommen.

```
        ...
        <dssp:Service>http://localhost/Example</dssp:Service>
    </ServiceRecordType>
</CreateServiceList>
```

3. **CREATE Request**
   Ein DSS-Node wird stets mit einer Reihe von System Services wie dem Constructor gestartet. Enthält eine CREATE-Request an den Constructor einen Namen, so erhält der erzeugte Service diesen als Identifier.

2.2.2.2 Contract Identifier

Als Contract wird eine Zusammenfassung des Service bezeichnet. Es beinhaltet die von ihm definierten Typen und sein Verhalten. Der Contract Identifier ist ein URI, der den Vertrag identifiziert und es anderen Teilnehmern ermöglicht, den Service zu benutzen. Ein typischer Contract-URI hat die Form:

`http://schemas.company.com/<year>/<month>/<serviceName>.html.`

Im Defaultprojekt wird in der Datei <ServiceName>Types.cs die Klasse **Contract** erzeugt, die den Identifier hält. Zusätzlich erhält die Serviceklasse das **ContractAttribut** mit einem Verweis auf den Contract Identifier.

```
[Contract(Contract.Identifier)]
[DisplayName("Example")]
[Description("This is an example service")]
class ExampleService : DsspServiceBase {
...
}
```

Listing 2.16: Contract Attribut

Abbildung 2.15: Contract Klasse

Der Contract wird vor allem auch dazu benutzt, um mit ihm eine Proxy-DLL des Service erzeugen zu lassen. Andere Services, die diesen benutzen, verlinken dann auf den Proxy statt direkt auf den Service (vergleichbar mit Stubs in CORBA).

```
using ST4Proxy = Bettzueche.Robotics.ServiceTutorial4.Proxy;

namespace ExampleService {
    class ExampleService: DsspServiceBase {
    ...
        ST4Proxy.ServiceTutorial4Operations _st4Port =
                                    new ST4Proxy.ServiceTutorial4Operations();
```

Listing 2.17: Verlinkung auf Proxy eines Services

Alle für die Kommunikation mit dem Service notwendigen Typen, dazu gehört insbesondere die Statusklasse, müssen mit dem **ContractDataAttribut** versehen werden (siehe Service State).

2.2.2.3 Service State

Der Service State repräsentiert den Service zu jedem beliebigen Zeitpunkt. Er ist als Klasse `<ServiceName>State` implementiert, die mit SOAP serialisierbar und somit auch im Webbrowser jederzeit zu betrachten ist. Daher wird der Service State gerne auch als dynamisches Dokument bezeichnet, das den Zustand des Service‘ beschreibt (Motor an, Drehzahl = 1000).

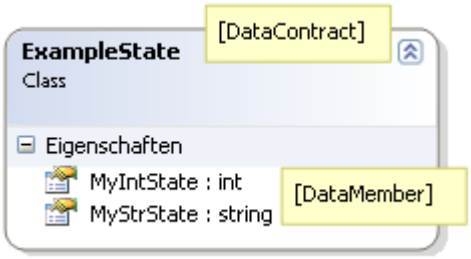


Abbildung 2.16: Service State Klasse

Die State-Klasse muss mit dem **ContractData**-Attribut markiert werden, damit der Compiler sie XML serialisierbar erstellt und vor allem in den Contract und somit in die Proxy-DLL aufnimmt. Die beinhalteten Daten werden über Eigenschafts-Felder zugänglich gemacht und müssen explizit mit dem **ContractMember**-Attribut zur Veröffentlichung markiert werden.

```
[DataContract]
public class ExampleState {
    [DataMember]
    public int MyIntState { get; set; }
    [DataMember]
    public string MyStrState { get; set; }
}
```

Listing 2.18: Service State, DataContract, DataMember

2.2.2.4 Main Port

Der Main Port zum Empfang von Requests ist ein CCR-Port, bzw. PortSet. Er ist ein Feld der Serviceklasse und muss durch das **ServicePortAttribut** als Main Port deklariert werden.

```
[ServicePort("/Example", AllowMultipleInstances = true)]
ExampleOperations _mainPort = new ExampleOperations();
```

Listing 2.19: ServicePort Attribut

Es ist üblich, einen separaten Typ zu definieren, der vom konkretisierten generischen PortSet erbt und die Klasse `<ServiceName>Operations` zu nennen. Diese Benennung ist sinnvoll, denn der Port stellt bekanntlich die Schnittstelle nach außen dar und die unterstützten Typen des Ports repräsentieren die

Abbildung 2.17: Operations-Klasse

Aktivitäten bzw. Operationen, die ein Dienst anbietet. Seien es DSSP-Operationen wie GET oder selbstdefinierte wie beispielsweise SetDrivePower.

```
[ServicePort]
public class ExampleOperations : PortSet<MyActivity, Get, Subscribe,
                                         DsspDefaultLookup, DsspDefaultDrop> { }
```

Listing 2.20: Typdefinition des (Main-)PortSets

Im DSS-Projekt wird eine Operations-Klasse mit einigen zwingend erforderlichen Standard-operationen automatisch in der Datei ~Types.cs generiert.

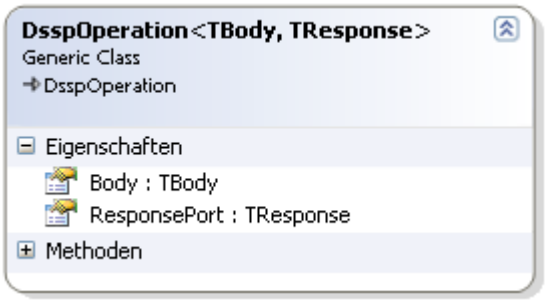


Abbildung 2.18: Basisklasse aller DSSP Operationen

Eine DSSP-Operation beinhaltet im Allgemeinen einen Requesttype, welcher ggf. Daten für die auszuführende Aktivität enthält, und einen ResponsePort. Selbstdefinierte Operationen sollten von einem der DsspOperation-Derivaten abgeleitet werden (siehe Tabelle 2.4). Eigene Requesttypes müssen mit dem ContractData-Attribut gekennzeichnet werden.

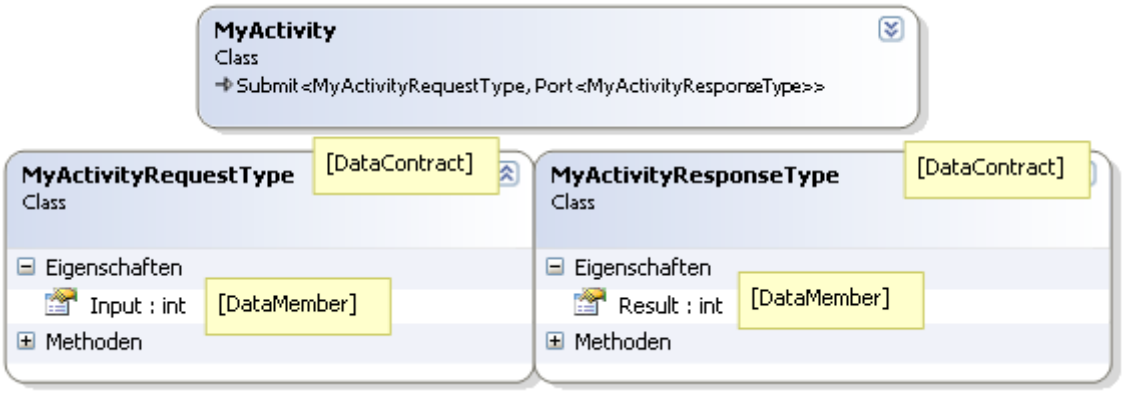


Abbildung 2.19: Klassendiagramme selbstdefinierter DSSP-Operation, Request-/ResponseType

Operation	Beschreibung
Create	Erzeugt neuen Service.
Delete	Die in der Delete-Request identifizierten Daten des Servicezustands löschen.
Drop	Beendet einen Service.
Get	Liefert den gesamten Zustand eines Service'.
Insert	Den Servicezustand um die enthaltenen Statusdaten ergänzen.
Lookup	Liefert den Servicekontext (Svc Identifier, Contract Identifier, Svc Context)
Query	Liefert Zustandsdaten die der spezifizierten strukturierten Anfragen entsprechen (SQL/LinQ)
Replace	Tauscht den gesamten Zustand mit dem enthaltenden aus.
Submit	Eine nicht Zustands ändernde Aktivität ausführen.
Subscribe	Für Benachrichtigungen über Zustandsänderungen anmelden.
Update	Modifiziert nur bestimmte Daten des Zustands (äquivalent zu Delete, Insert).
Upsert	Falls der Servicezustand bereits initialisiert ist, wird ein Update ausgeführt, andernfalls ein Insert.

Tabelle 2.4: DSSP Operationen (Nielsen, et al., 2007)

2.2.2.5 Service Handlers

Die Bearbeitung asynchroner Nachrichten in CCR wird von Handlern vorgenommen, so auch die Requests eines Service. Für jede Operation, also für jeden Typ des MainPorts, muss ein CCR-Handler vom Typ `IterativeHandler<OpType>` implementiert sein. Die DSS-Library stellt für fast alle Standard-DSSP-Operationen einen Default-Handler bereit, der automatisch übernommen wird, falls kein eigener implementiert ist. Selbstdefinierte Service Handler sollten mit dem **ServiceHandlerAttribut** markiert und über den zusätzlichen Attribut-Parameter **ServiceHandlerBehavior** (Concurrrent, Exclusive,…) das Nebenläufigkeits-verhalten bestimmt werden.

Das hat den Hintergrund, dass die Handler für die MainPort-Operationen von einem Interleave-Arbiter koordiniert werden (siehe Kap. CCR), welcher zur Basisklasse eines Service gehört und beim Start automatisch auf dem MainPort aktiviert wird. Das ServiceHandler-Attribut dient der deklarativen Registrierung des Handlers. Über das Eigen-

schaftsfeld `MainPortInterleave` der Service-Basisklasse ist der InterleaveArbiter auch direkt zugreifbar[9].

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> MyActivityHandler(MyActivity activity) {
    // Handle the MyActivity-Request
    yield break;
}
```

Listing 2.21: ServiceHandler mit deklarativer Registrierung

2.2.2.6 Service Partners

Partner Services sind diejenigen, mit denen der aktuelle Service interagiert, also deren Dienste er in Anspruch nimmt und eventuell sogar abhängig von ihrer Existenz ist, um selbst zu funktionieren. Im eingehenden Beispiel (siehe Abbildung 2.12: Roboteranwendungen) ist der MotorService ein Partner des Verhalten-Services.

Bekanntlich beschreibt der Contract das Verhalten eines Service und der Zugriff auf andere Services erfolgt über Proxies. Die Dienste (Operationen) werden angefragt, indem ein Service dem anderen eine Nachricht schickt. Vom Partner Service wird somit eine Referenz auf dessen Main Ports benötigt. Hier kommt das Partner-Konzept ins Spiel. Die Referenz auf den Partner ist nichts anderes, als dessen Main Port, der von seinem Proxy bereitgestellt wird. Dieser Port muss mit dem **PartnerAttribut** markiert werden, welches mit zusätzlichen Informationen ausgestattet werden kann, vor allem den Contract Identifier des Partners, aber auch wo der Service laufen soll, ob er extra gestartet werden soll oder wie zwingend er erforderlich ist.

```
using ST4Proxy = Bettzueche.Robotics.ServiceTutorial4.Proxy;
namespace ExampleService {
  //[Service Attributes]
  class ExampleService : DsspServiceBase {
    [Partner("MyPartner", Contract = ST4Proxy.Contract.Identifier,
             CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
    ST4Proxy.ServiceTutorial4Operations _st4Port =
                                  new ST4Proxy.ServiceTutorial4Operations();
```

Listing 2.22: Partner Service Deklaration

[9] Programmatisches Hinzufügen von Handlern sollte in der Start-Methode nach dem Aufruf von base.Start() geschehen, um sicher zu gehen, dass der MainPort-Arbiter korrekt initialisiert wird.

Abbildung 2.20 veranschaulicht, wie die lose Kopplung zweier Services und eine Request/Response-Kommunikation konzeptionell in DSS realisiert sind. Der Proxy Port des Partners, auch Forwarder genannt, serialisiert die Anfrage. System Services der Laufzeitumgebung (der DSS-Node) leitet die Nachricht an den Zielknoten weiter, wo die Nachricht deserialisiert dem eigentlichen Partner Port hinzugefügt wird. Eine Nachricht beinhaltet (meistens) den Response Port des Auftraggebers, für den die Laufzeitumgebung des Partner Knotens wiederum einen Forwarder generiert, so dass eine Antwort zurückgeschickt werden kann.

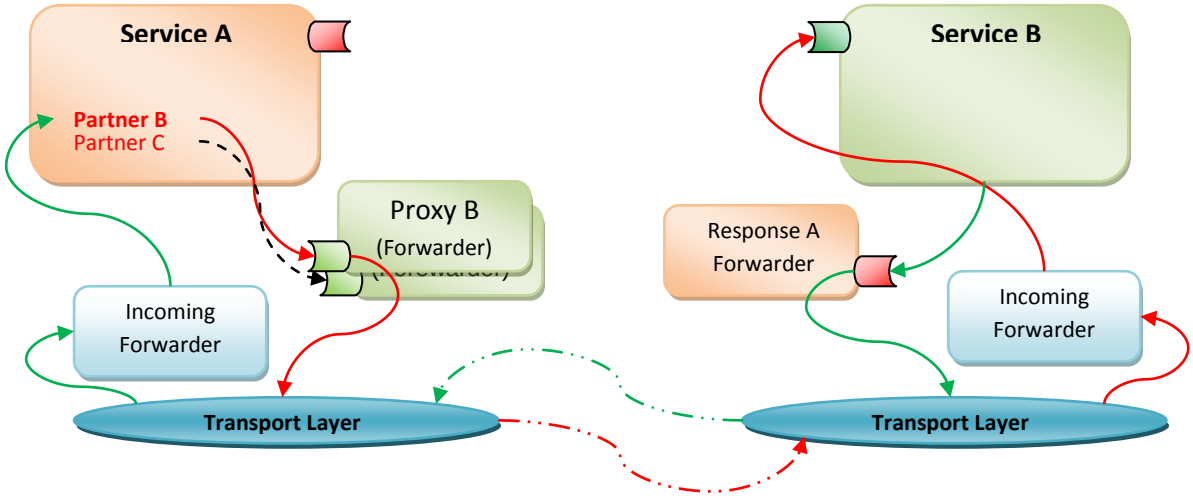


Abbildung 2.20: Service Partners, konzeptioneller Kommunikationsverlauf

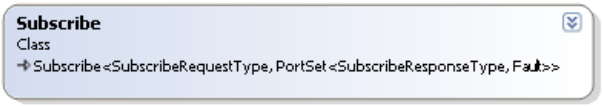
Dieses Prinzip erinnert wieder an CORBA, wo entfernte Objekte über einen Name- bzw. DirectoryService referenziert werden (Directory Service ist auch ein DSS System-Service), ein RequestBroker Nachrichten entgegennimmt und an einen geeigneten Skeleton zwecks unmarchalling weiterreicht und so weiter. Gemeinsamkeiten mit altbekannten Technologien kann man immer wieder entdecken. Ein Aspekt, der durchaus für das DSS Framework spricht, weil auch etablierte Techniken verwendet werden.

2.2.2.7 Notifications

Für DSS-Services ist ein Event-System realisiert, das dem Observer-Pattern entspricht. Hier spricht Microsoft von dem publish-subscribe Modell, das der asynchronen Benachrichtigung über Zustandsänderungen dient.

Ein Service, der Subscriber, kann sich bei einem anderen Service, dem Publisher, anmelden, um über Zustandsänderungen des Publishers benachrichtigt zu werden. Solche

Notifications lassen sich nach Typ oder Schlüsselattributen filtern, Näheres dazu ist in der Online Dokumentation nachzulesen (Microsoft Corporation, 2010d). Da nur DSSP-Operationen wie Insert, Update oder davon abgeleitete Operationen Zustandsänderungen herbeiführen, entsprechen die Notification-Nachrichten genau den Operationen des Main Ports. Das bedeutet in logischer Konsequenz, dass Notifications in erster Linie nur mitteilen, dass eine Änderung im Zuge einer bestimmten Operation stattgefunden hat, aber nicht zwangsläufig die geänderten Daten übermitteln. Eine Notification kann nur die Daten aufnehmen, die auch die entsprechende Operation beinhalten kann.



Damit sich andere Services zur Benachrichtigung anmelden können, muss dieser Dienst explizit angeboten werden, d.h. es muss die DSSP-Operation **Subscribe** implementiert und in den Main Port aufgenommen werden. Der zugehörigen SubsribeHandler reicht eine Anmeldung an den System Service **SubscriptionManager** weiter, welcher sich fortan auch um die Verteilung der Benachrichtigungen kümmert.

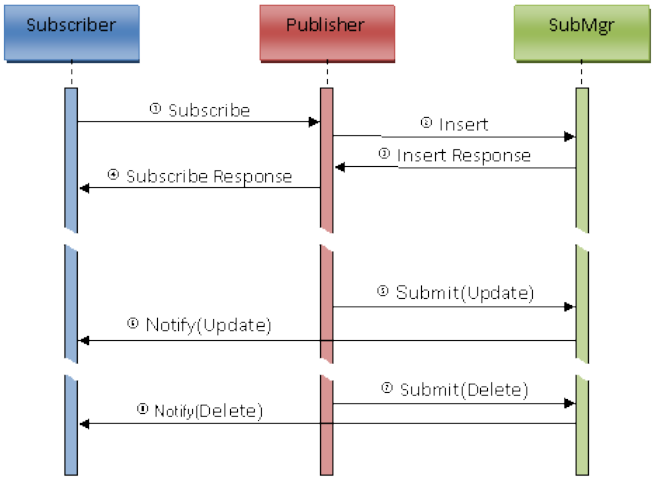


Abbildung 2.21: publish-subscribe Modell (Microsoft Corporation, 2010d)

Notifications senden

Die folgenden Code-Ausschnitte zeigen, wie ein Service die Subscribe-Operation und Notifications unterstützt. Der Operationstyp Subscribe ist in der ~Types.cs Datei definiert.

```
using submgr = Microsoft.Dss.Services.SubscriptionManager;
namespace ExampleService {
  //[Service-Attribute]
  class ExampleService : DsspServiceBase {
    // ...
    //Referenz auf SubscriptionManager
    [SubscriptionManagerPartner]
    submgr.SubscriptionManagerPort _submgrPort =
                                          new submgr.SubscriptionManagerPort();
    // ...
    // Subscription an den Manager weiterreichen
    [ServiceHandler]
    public void SubscribeHandler(Subscribe subscribe) {
      base.SubscribeHelper(_submgrPort, subscribe.Body, subscribe.ResponsePort);
    }
    // ...
    // Aktivität die Zustand ändert → Notification
    [ServiceHandler(ServiceHandlerBehavior.Concurrent)]
    public IEnumerator<ITask> MyActivityHandler(MyActivity activity) {
      // Handle the MyActivity-Request...
      base.SendNotification<MyActivity>(_submgrPort, activity);
      yield break;
    }
  }
}
```

Listing 2.23: SubscriptionManager und Notifications senden

Notifications empfangen

Es gibt für jeden Service, von dem Benachrichtigungen eingehen, einen separaten Port von dessen MainPort-Typ. So lässt sich die Verarbeitung der Nachrichten verschiedener Publisher beliebig kombinieren, beispielsweise mit einem Choice- oder Join-Arbiter. Die Handler zur Verarbeitung der Mitteilungen sind normalerweise vom Typ Handler<T>. Insbesondere sind sie aber keine Handler des eigenen MainPorts. Somit können sie nicht mit dem ServiceHandler-Attribut registriert werden. Der untere Code-Ausschnitt erweitert stattdessen den vorhandenen Interleave-Arbiter um die Notification-Handler. Dies geschieht in der Initialisierungsmethode `Start()` nach dem Aufruf von `base.Start()`, sodass sicher-

gestellt ist, dass der Interleave-Arbiter zuvor korrekt aktiviert wurde. Für jede Operation eines Partner Services stellt der Proxy Hilfsmethoden zur Verfügung, so auch für eine Subscribe-Request, hier am Ende der Start-Methode aufgerufen. Ein möglicher Fehler bei der Anmeldung (Fault-Response) wird hier ignoriert.

```
using ST4Proxy = Bettzueche.Robotics.ServiceTutorial4.Proxy;
namespace ExampleService {
  //[Service-Attribute]
  class ExampleService : DsspServiceBase {
    //Eingangsport für ST4-Notifications:
    ST4Proxy.ServiceTutorial4Operations _st4Notify =
                                  new ST4Proxy.ServiceTutorial4Operations();
      //...
    protected override void Start() {
      base.Start();
     // Interleave-Arbiter des MainPorts hier um die NotifyHandler erweitern:
      base.MainPortInterleave.CombineWith(Arbiter.Interleave(
        new TeardownReceiverGroup(),
        new ExclusiveReceiverGroup(),
        new ConcurrentReceiverGroup(
         Arbiter.Receive<ST4Proxy.Replace>(true,_st4Notify,notifyReplaceHandler),
         Arbiter.Receive<ST4Proxy.IncrementTick>(
            true,
            _st4Notify,
            tick => { //Lambda-Handler for IncTick-Notify
              LogInfo("Got Tick");
              _mainPort.Post(new MyActivity(42));
            }
         )
       ) //ConcurrentReceiverGroup
      )); //CombineWith()

      //Beim Partner anmelden:
      _st4Port.Subscribe(_st4Notify);
    }
      // ...
    void notifyReplaceHandler (ST4Proxy.Replace notify) { //Handle Notification }
```

Listing 2.24: Handle and subscribe for Notification

2.2.3 Nachrichten Performanz

In vielen eingebetteten Systemen und in besonderem Maße bei Roboteranwendungen wie einem Rescue Robot spielen Echtzeitbedingungen eine große Rolle. So müssen diverse Sensordaten innerhalb eines kritischen Zeitfensters gelesen und ausgewertet werden. Neben Software-Metriken, wie der Effizienz eines Algorithmus, ist in einer Service Orientierten Architektur wie DSS die Verarbeitungsgeschwindigkeit von Nachrichten ein wesentliches Merkmal.

Da DSS auf verteilte Anwendung ausgelegt ist, ist die Kommunikation für verschiedene Konfigurationen optimiert. Es werden drei Fälle unterschieden: Services laufen in einem Prozess (DSS-Knoten); Services kommunizieren zwischen zwei Knoten, aber auf einer Maschine; Knoten und Services sind auf separate Maschinen verteilt. Microsoftnahe Quellen beziffern die Zahl zu verarbeitender Nachrichten mit 100.000 Nachrichten pro Sekunde bei prozesslokalen Kommunikationen und mit 3.000 bei echt verteilten Services (siehe auch Abbildung 2.22), jeweils auf einem Intel Dual Core getestet. Allerdings sind in keiner der einschlägigen Quellen nähere Details zum Benchmark zu finden, sodass ein eigener einfacher Test entwickelt wurde.

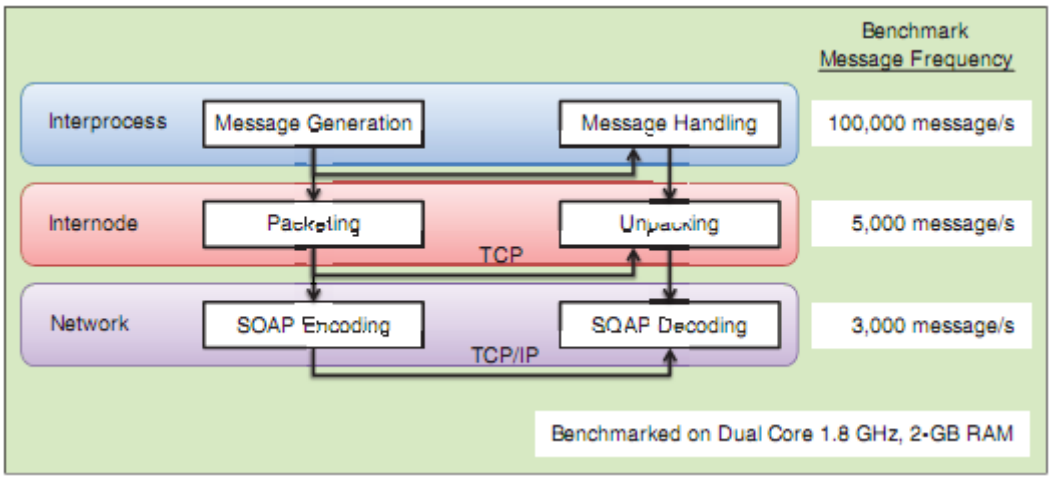


Abbildung 2.22: DSS Nachrichten Performanz (Jackson, 2007)

Benchmark-Services

Es wird eine simple Request/Response Kommunikation getestet, die, abgesehen von dem Hochzählen eines Zählers, keine Berechnungen beinhaltet. Ein Service, der „Driver“, sendet eine Request an einen „CounterService“. Sobald die Response angekommen ist, wird sofort eine neue Request gesendet. Der CounterService inkrementiert mit jeder Request seine

Statusvariable Count. Desweiteren bedient sich der CounterService eines PerformanceCounters des Windows-Frameworks, der periodisch jede Sekunde mit dem aktuellen Count-Wert beschrieben wird, angestoßen durch eine intern erzeugte, timer-basierte Request. Mit Hilfe des Windows eigenen Tools „Performance Monitor“ (deutsch: Leistungsüberwachung) können die verarbeiteten „Messages per Second“ visuell oder textuell überwacht werden. Eigentlich müsste der Zähler „Requests&Responses per Second“ heißen, da es sich genau genommen um zwei Nachrichten handelt, die gezählt werden. Alternativ kann man den angezeigten Wert auch einfach verdoppeln. Der Singlenode- und Internode-Test wurde jeweils auf einem Dual Core mit 2,5 GHz ausgeführt. Beim maschinen-übergreifenden Test per WLAN lief der „CounterService“ auf dem zuvor genannten Dual Core und der „Driver“ auf einem Single Core Notebook mit 1,3 GHz.

Single Node

Wie unterer Abbildung zu entnehmen ist, ergab die prozesslokale Variante auf dem Dual Core eine Request/Response-Frequenz von etwas mehr als 15.000 Nachrichtenpaaren pro Sekunde. Das entspricht gerade einmal 30% des erwarteten Wertes.

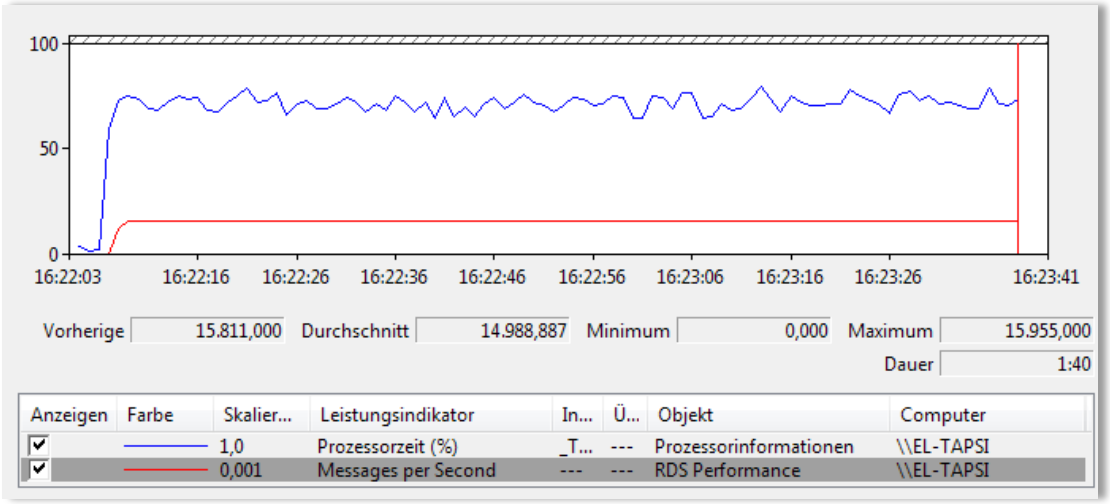


Abbildung 2.23: Nachrichtenperformanz Single Node (Performance Monitor)

Inter Node – Single Machine

War schon der erste Test nicht sonderlich zufriedenstellend, zeigt sich bei dem Inter-Node-Test ein nahezu dramatisches Ergebnis. Startet der DriverService, so werden anfänglich ca. 1.000 Nachrichtenpaare bei einer Netzwerklast von 10 MBit/s verarbeitet, was 40% des Erwartungswerts entspricht. In einer steil regressiven Kurve sinkt die Frequenz sogleich auf wenige hundert Nachrichten, nach etwa fünf Minuten auf sogar nur noch 50. Dieser Effekt tritt wiederholt nur bei der Internode-Konfiguration auf. Eine Netzwerkauslastung konnte

ausgeschlossen, aber die genaue Ursache nicht ermittelt werden. In den Foren und Newsgroups der Community sind keine Hinweise zu finden, dass ähnliche Effekte bekannt sind. Daher ist dieser Test vorerst unter Vorbehalt zu betrachten und bedarf näherer Untersuchung.

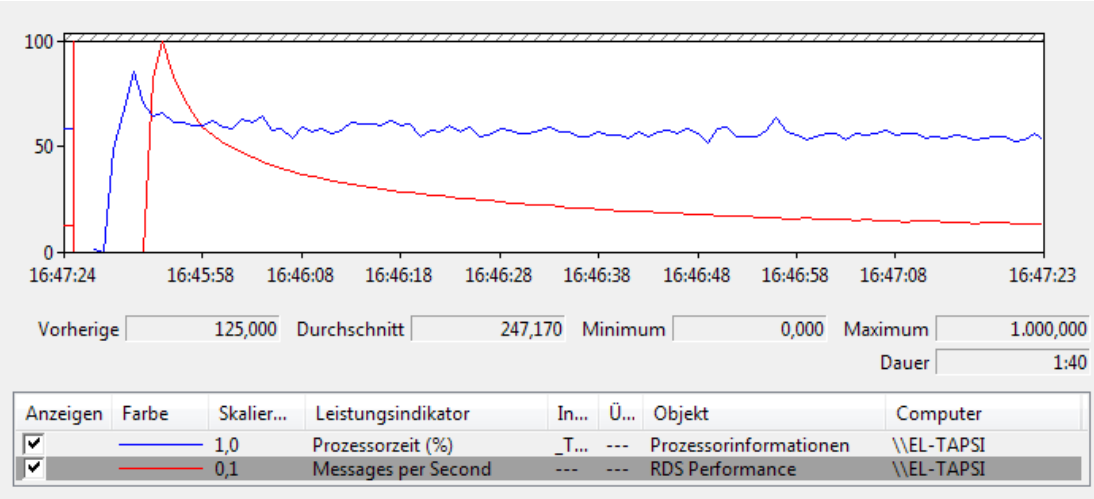


Abbildung 2.24: Nachrichtenperformanz Inter Node (Performance Monitor)

Cross Machine

Die über Maschinengrenzen verteilte Konfiguration zeigt eine einigermaßen stabile Nachrichtenfrequenz von etwa 140 Request/Response-Nachrichten pro Sekunde bei einer Netzwerklast von ca. 600Kbit/s. Allerdings sind dies gerade einmal 4% der versprochenen Performanz.

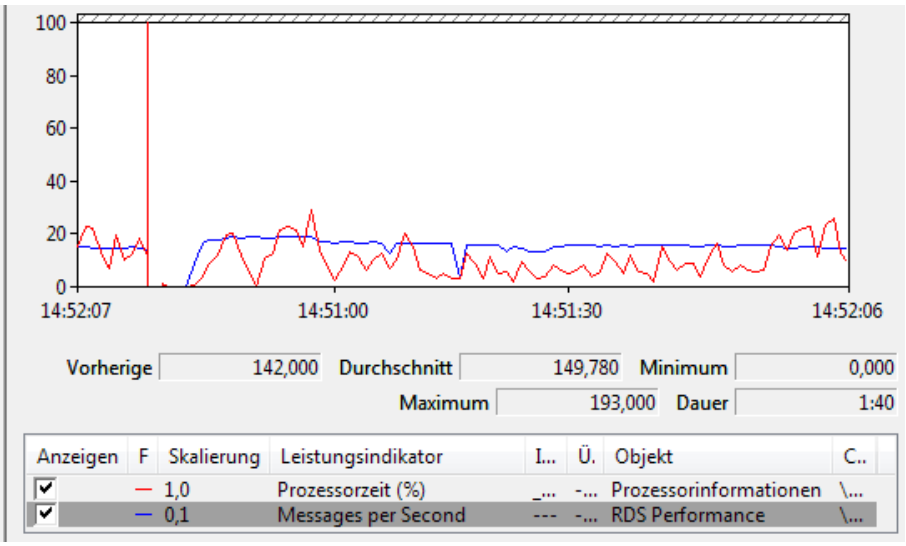


Abbildung 2.25: Nachrichtenperformanz Cross Machine (Performance Monitor)

Testbeurteilung

Obwohl nur Standard-Templates und –Konfigurationen für das Testprojekt benutzt und die Testservices extra simpel gehalten wurden, stellte sich ein deutlich schlechteres Bild dar, als von Microsoftquellen versprochen. Insbesondere der Internode-Test, mit seiner regressiv fallenden Nachrichtenfrequenz, lässt allerdings Zweifel an der Korrektheit des Tests aufkommen. Trotzdem spricht es nicht gerade für das DSS-Framework, wenn eine einfache Anwendung wie dieser Test, sich derart dramatisch auf die Performanz auswirkt. Sollte sich das AMEE-Projektteam für die Benutzung von Robotics Studio entscheiden, sind auf jeden Fall weitere Untersuchungen zur Performanz ratsam.

2.2.4 Der Nutzen von DSS

DSS verspricht, die wichtigsten und gängigsten Anforderungen für Roboteranwendung zu berücksichtigen und eine einheitliche, allgemein verwendbare Architekturplattform zu bieten.

- **Robustheit**
  Stabilität und Robustheit wird durch Entkopplung der Komponenten in isolierte Dienste erzielt. Aber auch serviceintern werden Daten und Ausführung durch CCR streng voneinander getrennt.
  Die Fehlerbehandlung erfolgt entweder servicelokal und/oder als Benachrichtigung und somit wieder isoliert; eine DSS-Anwendung ist stets datengetrieben.
  Ein einheitliches, übersichtliches Programmiermodell zur Nebenläufigkeit (CCR) vereinfacht die Entwicklung und schützt potentiell vor Design- und Implementierungsfehlern, die die Threadsicherheit beeinträchtigen.
  Services lassen sich dynamisch erzeugen und beenden, auch über Rechnergrenzen hinaus (Skalierungs- und Leistungstransparenz, Fehlertransparenz).

- **Konfigurierbarkeit**
  Die Laufzeitumgebung und DSSP unterstützen zusätzlich zur Erzeugung auch das Verwalten, Auffinden, Komponieren und Verteilen von Services.
  Desweiteren erlaubt DSSP die Manipulation strukturierter Daten und somit u.a. eine Konfiguration des Systems auch zur Laufzeit.
  Generische Services ermöglichen den einfachen Austausch von Algorithmen oder Hardwarekomponenten. Durch Manifeste lässt sich eine Anwendung neu konfigurieren, ohne neu Kompilieren zu müssen.
  Das DSS-Modell ermöglicht einen erhöhten Grad paralleler Entwicklung und der Wiederverwendbarkeit von Softwarekomponenten.

- **Monitoring**
  Zustand und Verhalten eines Services sind getrennt. Der Zustand eines jeden Service ist quasi ein dynamisches Dokument, welches jederzeit abgerufen werden kann. Entweder als XML-Struktur, welches auch im Webbrowser dargestellt werden kann, oder als serialisiertes Datenobjekt zur Verwendung in anderen Services.
  DSS beinhaltet ein Eventsystem, welches mittels Notifications Änderungen des Status bekanntgibt. Benachrichtigungen gehen dabei nur an registrierte Services und können zusätzlich gefiltert werden. Dem Programmierer steht auch der Debugger von Visual Studio zur Verfügung.

- **Soft Skills**
  Die Implementierung eines Service ist denkbar einfach, durch das in VS integrierte Projekt-Template steht mit wenigen Maus-Klicks ein funktionsfähiger Service zur Verfügung. Der Entwickler kann sich weitgehend auf die Semantik konzentrieren. Seit der Version 2008 R3 ist die DSS/CCR-Library auch für den professionellen Gebrauch kostenlos. Es gibt ausführliche, leicht verständliche Dokumentationen und Tutorials, die einen Einstieg in wenigen Tagen ermöglichen. Eine etablierte Community und viele Open-Source-Projekte bieten einen guten Support. Da das Framework von Microsoft entwickelt und vertrieben wird, kann man eine gute Performance auf und Kompatibilität zu aktuellen und künftigen MS Betriebssystemen erwarten und darauf hoffen, dass die Weiterentwicklung der Library, wie auch die Community in mittelbarer Zukunft noch lebhaft Bestand haben wird.

Doch wie fast überall hat auch DSS seine zwei Seiten, und es gilt einige Nachteile und potentielle Herausforderungen zu berücksichtigen. So ist man (trotz Mono[10]) im Grunde auf ein MS Betriebssystem beschränkt. Diese Plattformabhängigkeit kann unter wirtschaftlichen, politischen oder ideologischen Gesichtspunkten ein bedeutsames Manko darstellen. Auch die Notwendigkeit eines Betriebssystems an sich kann problematisch sein. Viele Roboter sind in ihren Rechenressourcen und Energiekapazitäten soweit eingeschränkt, dass eine Plattform mit MS Betriebssystem nicht in Frage kommt. Insbesondere, da seit der Version 2008 R3 das .NET Compact Framework nicht mehr unter-stützt wird und somit auch die Verwendung von Microsoft CE[11] entfällt. Zwar ist es mit DSS immer noch möglich, die höheren und rechenaufwändigeren Komponenten auf einen leistungsstarken PC auszulagern und den Roboter fernzusteuern, dies ist für autonome Feldroboter aber nicht immer akzeptabel. Für das Design und die Implementierung von Roboteranwendungen mit DSS ist ein weiterer Aspekt des SOA-Konzepts zu berücksichtigen. Services sind prinzipiell gleichberechtigt, ihre Dienste lassen sich zwar hierarchisch anordnen, aber nicht direkt priorisieren. Das mehrschichtige Verhaltensmodell Subsumption beispielsweise, wie es von Brooks in einem IEEE Journal (Brooks, 1986) beschrieben und von Andreas Basener in seiner Bachelorarbeit (Basener, 2008) benutzt wurde, lässt sich nur schwer adaptieren[12]. Stattdessen müssen im Designprozess mit DSS sehr sorgfältig der Grad der Granularität, zu erwartende Nachrichtenhäufigkeiten und die Verteilung des Systems bestimmt werden, sodass sich der Roboter auch in extremen Situationen angemessen verhält. Gegebenenfalls ist es sicherlich ratsam, hardwarenahe und vor allem zeitkritische Anwendungen auszulagern. Wie dies realisiert werden kann, zeigt das nächste Kapitel.

[10] Plattformunabhängige, .NET kompatible Entwicklungs- und Laufzeitumgebung der Firma Novell
[11] Microsofts Echtzeitbetriebssystem für Embedded Systems
[12] Insbesondere können keine Nachrichten unterdrückt oder überschrieben werden. Ein Versuch, dies in DSS programmatisch zu realisieren, würde massiv gegen die zugrunde liegenden Paradigmen und Modelle arbeiten.

2.3 Hardware Integration nach RDS

MCU-Komponente einer RDS-Anwendung

Ein Roboter besitzt typischerweise eine ganze Reihe von Aktoren, Sensoren, Hilfsbauteile wie Integrierte Schaltkreise (IC), ein oder mehrere Microcontroller (MCU), verschiedenste Treiber und Bussysteme. Dieses Kapitel widmet sich der Frage, welche Entwicklungsschritte notwendig sind, um Hardwarekomponenten in eine RDS-Anwendung zu integrieren. Dazu wird beispielhaft eine Microcontroller-Anwendung geschrieben, die über einen seriellen Bus (RS-232) Befehle zur Steuerung eines Schrittmotors empfängt. Anschließend wird die Funktionalität in ein DSS-Service gekapselt, um sie im RDS-Kontext zur Verfügung zu stellen. Der Motor soll ein- und ausgeschaltet werden und sich in einer angegebenen Geschwindigkeit und Richtung drehen können.

Material

- PC mit serieller Schnittstelle
- Atmel Entwicklerboard STK500 mit serieller Schnittstelle
- ATmega 8515L:
  - 8 Bit MCU in RISC-Architektur
  - 8 KByte Flash-Speicher, 512 Byte interner SRAM, 512 Byte EEPROM
  - 4 x 8 programmierbare I/O -Pins
  - Bis 8 MHz System-Takt
  - 2 Timer(8 und 16 Bit), Watchdog, USART, Analog-Comparator
- Bipolarer Schrittmotor (aus altem Drucker), 42 V, 250 Schritte, 10 Ohm.

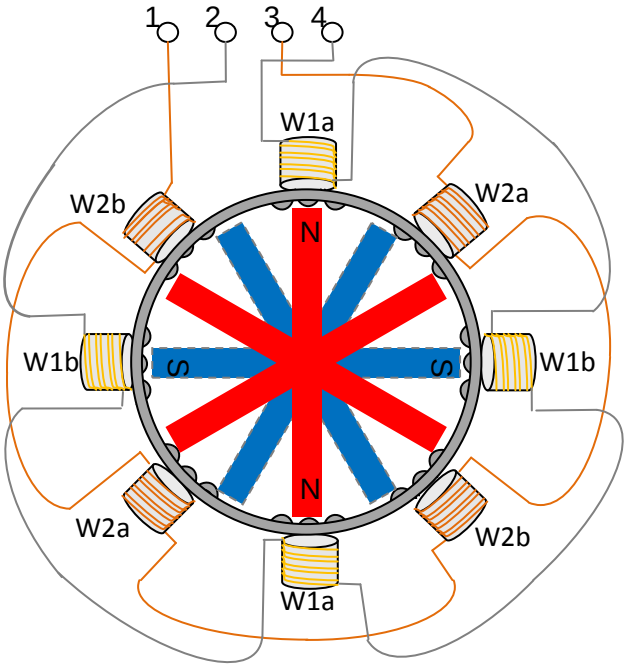
Um eine konkrete Drehgeschwindigkeit zu realisieren, muss der Motor periodisch angesteuert werden (siehe Schrittmotor). Ziel des ersten Entwurfsschritts ist es daher, den auf der MCU integrierten Timer für diesen Zweck zu programmieren. Der Microcontroller kann und soll über eine serielle Schnittstelle angesprochen werden. Der zweite Entwurfsschritt erweitert deswegen den ersten um eine Implementierung dieser Kommunikationsart. Nachdem die benötigten Funktionalitäten generell beleuchtet sind, folgt letztlich die Realisierung der MCU-Anwendung zur Steuerung des Motors und die Implementierung des den Motor repräsentierenden Service.

Anhand dieser exemplarischen Entwicklung werden folgende generelle Fragen verdeutlicht:

1. Was soll das Bauteil können?
2. Wie und womit wird seine Funktionalität realisiert?
3. Welcher Art ist die Kommunikation zum Gerät und wie wird sie umgesetzt?
4. Wie wird das Bauteil von einem DSS-Service benutzt und repräsentiert?

2.3.1 Schrittmotor

„Ein **Schrittmotor** ist ein Synchronmotor, bei dem der Rotor (drehbares Motorteil mit Welle) durch ein gesteuertes schrittweise rotierendes elektromagnetisches Feld der Statorspulen (nicht drehbarer Motorteil) um einen minimalen Winkel (Schritt) oder sein Vielfaches gedreht werden kann.“ (Wikipedia)



Der hier schematisch dargestellte Schrittmotor ist dem in dieser Arbeit verwendeten nachempfunden.
Über jeweils zwei Anschlusspins werden zwei Vierergruppen Statorspulen angesteuert.
Die sich gegenüberliegenden Spulen gehören einer Gruppe an und sind immer gleichherum gewickelt, sodass sich ihr Magnetfeld identisch ausrichtet. Die anderen beiden Spulen der Gruppe sind um 90° versetzt und entgegengesetzt zu den ersteren gewickelt.

Abbildung 2.26: Schrittmotor. Schematische Darstellung eines bipolaren Hybridmotors

Der hier verwendete Motor ist ein Hybridmotor. Der Rotor besteht aus zwei übereinander liegenden gezahnten Scheiben, die gegensätzlich magnetisiert und um eine halbe Zahnung gegeneinander versetzt sind. Bei der Ansteuerung der Spulen erzeugen diese ein Magnetfeld, auf das sich der Rotor ausrichtet. Durch geschickte wechselnde Polarisierung der Spulen wird der Rotor in Drehung versetzt. Man unterscheidet verschiedene Ansteuerarten. Werden immer beide Spulengruppen geschaltet, so spricht man von Vollschritt-Betrieb, der Rotor dreht um den motorspezifischen Schrittwinkel. Bei dem verwendeten Motor ergeben beispielsweise 250 Schritte eine volle Umdrehung, somit entspricht ein Vollschritt 1,44°. Der Schrittwinkel lässt sich durch eine Halbschrittansteuerung halbieren. Ein Halbschritt entsteht, indem zwischen den Vollschritten eine Spulengruppe

nicht mehr mit Strom versorgt (ausgeschaltet) wird. Dadurch reduziert sich allerdings auch das mittlere Drehmoment, da das Magnetfeld zwischenzeitig schwächer ist. Außerdem kann die angelegte Spannung noch schrittweise erhöht und gesenkt werden (Mikroschritt) oder man betreibt den Motor mit Wechselspannung. Die Abbildung 2.29 schematisiert einen kompletten Steuerzyklus, der bei Vollschritt vier und bei Halbschritt acht Phasen beinhaltet.

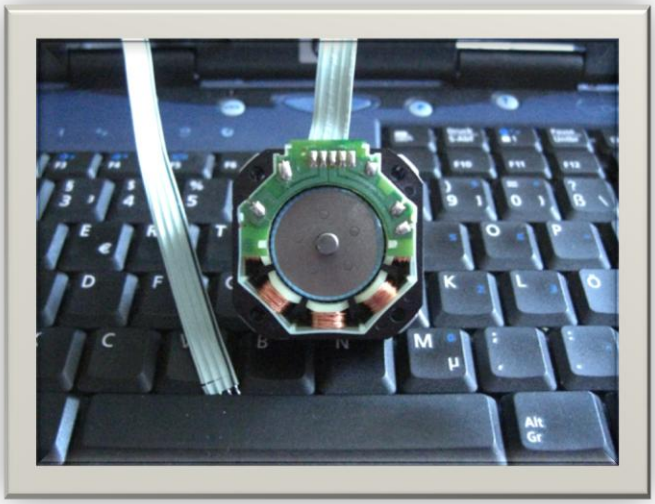


Abbildung 2.27: Schrittmotor aus einem Drucker

Abbildung 2.28: Atmel Development Board STK500

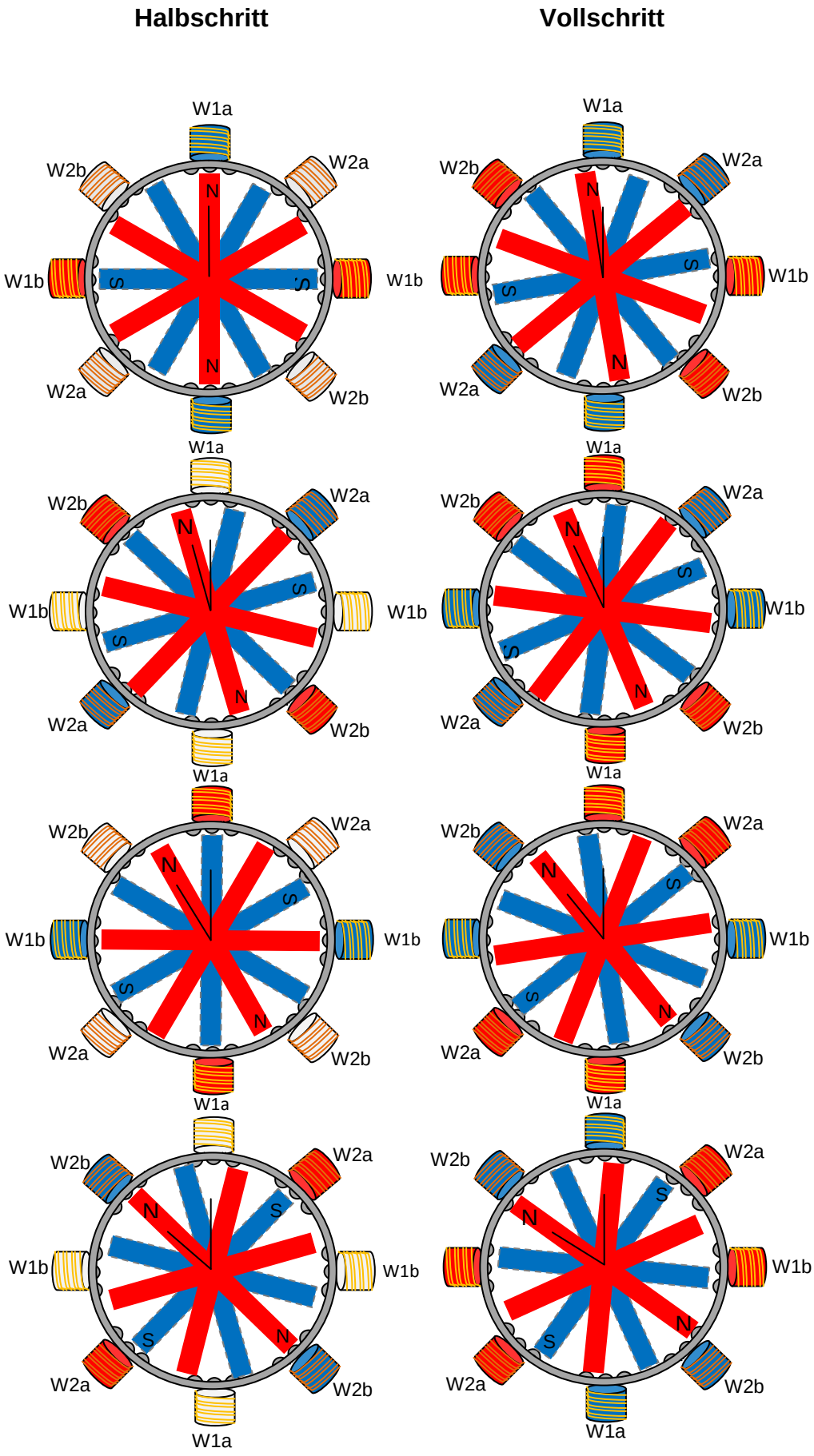


Abbildung 2.29: Schrittmotorphasen

2.3.2 Design Entwurf

Der Schrittmotor hat eine Nennspannung von 42 Volt bei einem Spulenwiderstand von 10 Ohm. An den Ausgangspins des Microcontrollers liegen aber nur 5 Volt an (Betriebsspannung) und sie sollten nicht mit mehr als 20 mA belastet werden. Das bedeutet, dass der Motor nicht direkt von der MCU betrieben werden kann, sondern eine elektronisches Schaltnetz (z.B. H-Brücke oder gekaufter IC) und eine separate Spannungsversorgung zwischengeschaltet sein muss. Falls so etwas nicht zur Verfügung steht, kann man die Funktionalität auch dadurch testen und veranschaulichen, indem die Steuersignale der MCU auf die LEDs des Entwicklungsboards gelegt werden.

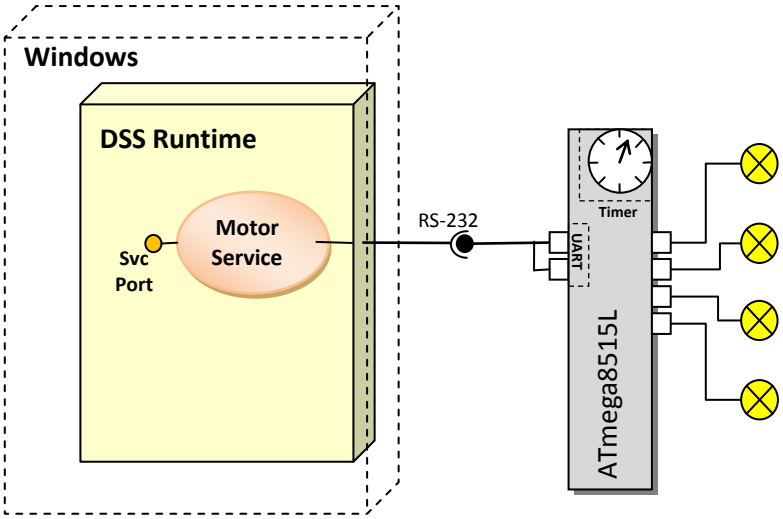


Abbildung 2.30: Systemaufbau der Motorsteuerung mit RDS

Ein DSS-Service repräsentiert den Schrittmotor durch seinen Status und sein Verhalten. Zur Beschreibung des Motorzustands gibt der Service an, wie schnell und in welche Richtung sich der Motor dreht, sowie ob er aus- oder eingeschaltet ist. Eingeschaltet bedeutet in diesem Fall, dass der Motor mit Strom versorgt wird und somit ein Haltemoment (Drehmoment im Stand) wirkt. Zustandsverändernde Operationen (das Verhalten) sind Ein-/Ausschalten, Drehen und Stopp. Andere Services können den Motor-Service beauftragen, eine solche Operation auszuführen, woraufhin der Befehl intern über die serielle Schnittstelle an den Microcontroller weitergereicht und real umgesetzt wird. Die MCU-Anwendung erzeugt dann die notwendigen Steuersignale für den Schrittmotor.

2.3.3 Entwurf 1 – Blinklicht

Für die Umsetzung der Drehgeschwindigkeit ist ein periodisches Timersignal vorgesehen, um den nächsten Motorschritt auszuführen. Der erste Entwurfsschritt widmet sich somit der Aufgabe, einen der MCU-internen Timer so zu programmieren, dass er jede halbe Sekunde ein Timer-Interrupt auslöst, der eine ISR anstößt. Um zu signalisieren, dass die ISR ausgeführt wird, soll sie eine der Board-LEDs abwechselnd ein- und ausschalten, also im Sekundentakt blinken lassen.

2.3.3.1 Timer Setup

Der in der MCU integrierte Timer soll mit der System-Clock (f_{Sys}=8 MHz) getrieben werden. Beim ATmega8515 lässt sich die Frequenz noch um einen Prescaler von 1, 8, 64, 256 oder 1024 teilen, um die letztliche Zählfrequenz f_{TC} einzustellen, mit der von 0 bis zum Maximalwert MAX gezählt wird (beim 8Bit-Timer0 ist MAX=255, beim 16Bit-Timer1 ist MAX=65.535).

$$f_{TC} = {f_{Sys}}/{Prsc} \qquad \text{(1) Counter Frequency}$$

Im sogenannten Clear Timer on Compare (CTC) Mode zählt der Timer periodisch immer von 0 bis zu einem benutzerdefinierten Maximalwert TOP hoch, welches im Output Compare Register OCR0 (Timer0) bzw. OCR1A (Timer1) gespeichert wird. Bei Erreichen des Wertes TOP wird der Zähler zurückgesetzt und das Output Compare Flag OCF gesetzt, das als Trigger für den Interrupt dient. Es gilt nun Prescaler und TOP so zu wählen, dass jede halbe Sekunde ein Interrupt ausgelöst wird, also $f_{OCF} = 2\ \text{Hz}$ gilt.

$$f_{OCF} = \frac{f_{TC}}{TOP} = \frac{f_{Sys}}{TOP * Prsc} \qquad \text{(2a) Interrupt Frequenz}$$

$$TOP = \frac{8\ MHz}{Prsc * 2\ Hz} = \frac{4 * 10^6}{Prsc} \qquad \text{(2b) TOP-Berrechnung}$$

Aufgrund der Timer-Registergrößen von 8 bzw 16 Bit muss der Prescaler so gewählt werden, dass TOP kleiner 255, respektive 65.535, ist. Für einen Prescaler von 256 ergibt sich aus Gleichung 2b für TOP der Wert **15.625**.

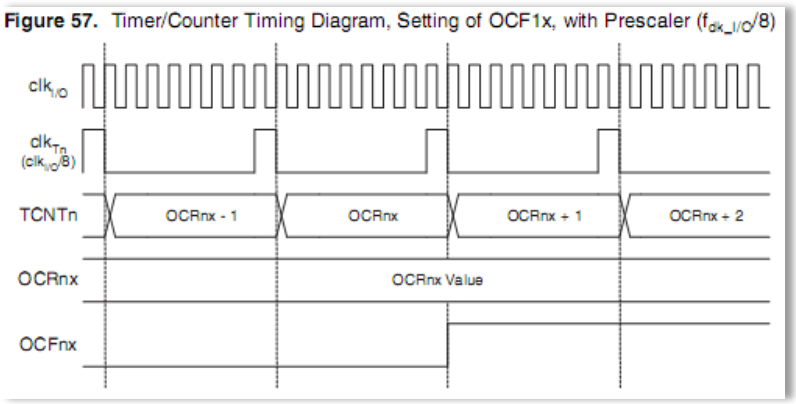


Abbildung 2.31: MCU Timer im CTC-Mode (Atmel Corporation, 2010)

In späteren Entwurfsschritten erhält das Programm eine Vorgabe für f_{OCF}, die sich aus der gewünschten Umdrehung pro Sekunde ergibt. Der Wert TOP muss dann softwareseitig berechnet und eingestellt werden. Hier ist der oben errechnete Wert für TOP noch fix und wird in der Initialisierungsmethode des Timers benutzt. Alle übrigen Einstellungen lassen sich aber wiederverwenden. Folgende Methode initialisiert den Timer1 im CTC-Mode, so dass alle 500ms ein Output-Compare-Match Interrupt ausgelöst wird.

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define TOP_CNT 15625 // -> 2Hz IntrFrq bei Prsc=256 und SysClk=8MHz
#define ON  1
#define OFF 0

volatile static uint8_t led_is_on = OFF; //Flag für die ISR

/**
 * Timer1 initialisieren:
 * Laesst eine ISR alle 0.5sek ausfuehren.*/
void init_Timer() {
    // Sicherheitshalber atomarer Zugiff, also Intr deaktivieren.
    unsigned char sreg;
    sreg = SREG; //save global interrupt flag
    cli();      //disable Interrupts

    TCCR1B |= _BV(CS12); // Prescaler 256 (Tbl. 54, S. 122)
    TCCR1B |= _BV(WGM12); //CRC-Mode mit TOP=OCR1A (Tbl. 53, S. 121)
    OCR1A = TOP_CNT; //TOP ins CompareReg (16 Bit Op!)
    TIMSK |= _BV(OCIE1A); //Enable OC-Interrup (S. 124)
    // OCF1A wird im TIFR-Reg gesetzt und autom. geloescht,
    // wenn der zugehoerige Interuptvector TIMER1_COMPA
    // ausgefuehrt wird (S. 125; S. 54, Tbl. 22)

    SREG = sreg;   //restore global interrupt flag
}
```

Listing 2.25: MCU-Timerinitialisierung zur Auslösung einer periodischen ISR

2.3.3.2 Interupt Service Routine

Die Interrupt Service Routine soll das Steuersignal auf den Ausgangsport legen, an dem in diesem Fall eine einfache LED angeschlossen ist. Zur Verwendung des Makros ISR() siehe AVR-libc Dokumentation (SourceForge, 2010)

```
ISR(TIMER1_COMPA_vect) {

    if (led_is_on) {
        PORTB &= ~_BV(PB0); //Licht aus
        led_is_on = OFF;
    }
    else {
        PORTB |= _BV(PB0); //Licht an
        led_is_on = ON;
    }
}
```

Listing 2.26: ISR; Steuersignale setzen

2.3.4 Entwurf 2 – RS-232 Schnittstelle

Sowohl das Entwicklungsboard STK500 als auch die MCU ATmega8515L unterstützen nativ eine RS232-Schnittstelle zur Kommunikation mit anderen Hardwarekomponenten. Für den endgültigen Entwurf ist daher vorgesehen, dass der Controller über den seriellen Bus ausgehend vom Com1-Port des PCs seine Befehle erhält. Der zweite Entwurfsschritt widmet sich der Aufgabe, eine solche Kommunikation zu implementieren.

Die Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) Komponente ist wie der Timer ein separates, integriertes Gerät der MCU, das die serielle Übertragung via RS-232 realisiert und dessen Logik parallel zur Software ausgeführt wird. Sie wird so konfiguriert, dass sie ein Interrupt auslöst, sobald ein Wort (8 Bit) empfangen wurde, woraufhin die zugehörige ISR den Timer aktiviert bzw. deaktiviert (Timer Interrupt disable).

Zwar soll (und muss) die ISR das empfangene Datum auslesen, verwirft es aber unbeachtet. In späteren Entwurfsschritten muss ein geeignetes Datenprotokoll entworfen werden, das die verschiedenen Befehle beinhaltet, die dann von der ISR (oder einer Polling-Routine) interpretiert werden.

2.3.4.1 RS-232

RS-232 bezeichnet eine Norm für eine serielle Schnittstelle, die ursprünglich hauptsächlich für die Kommunikation zwischen zwei Rechnern über das Telefonnetz diente. Die Übertragung der Datenwörter (5 bis 9 Bits) geschieht seriell und asynchron über je eine Leitung pro Übertragungsrichtung. Beginn und Ende eines Wortes wird dabei durch ein Start- bzw. Stopp-Bit gekennzeichnet, sodass eine beliebig lange Übertragungspause zwischen zwei Wörtern vorhanden sein kann. Asynchronität meint hier, dass es keine gemeinsame Taktleitung zur Synchronisation der Datenrate gibt. Stattdessen dient das Startbit der Synchronisation und es muss auf beiden Seiten eine gleiche Bitrate (Baud) eingestellt werden. Um zusätzlich Schwankungen und Störungen bei der Übertragung auszugleichen, werden die Bits mit einer höheren Frequenz als der Übertragungsrate abgetastet (sample rate) und der Bit-Wert aus den mittleren Abtastergebnissen ermittelt. Ein dem Wort optional angehängtes Paritätsbit ermöglicht eine einfache Fehlererkennung.

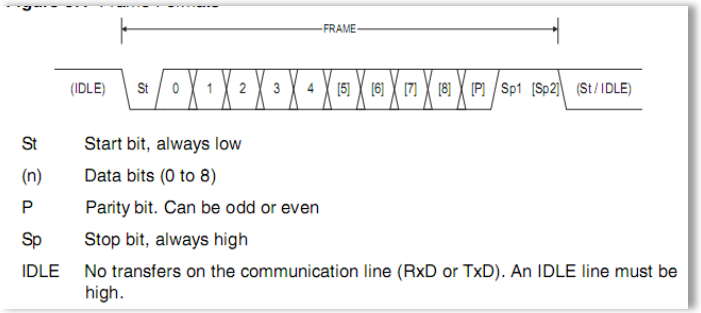


Abbildung 2.32: Frame Format (Atmel Corporation, 2010)

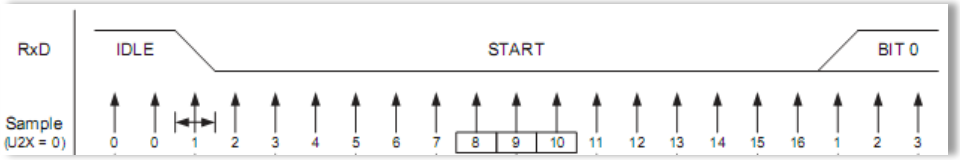


Abbildung 2.33: StartBit Sampling; DataBit Sampling äquivalent (Atmel Corporation, 2010)

2.3.4.2 Intialisierung

Die Wahl der Bitrate hängt unter anderem von der verwendeten Leitungslänge ab, da sie sich durch Leitungswiederstand und –Kapazität negativ auf den Spannungspegel auswirkt. Nach unbestätigten Quellen (Wikipedia) sind bei Längen bis zu zwei Metern, wie sie in einem Rescue Robot wahrscheinlich sind, bis zu 115.200 Baud unbedenklich. Ein weiterer Aspekt ist die Differenz zwischen der empfangenen und der intern erzeugten Baud-Rate. Je größer der Unterschied ist, desto eher laufen Sender und Empfänger bei langen Frames, also mehreren direkt aufeinander folgenden Wörtern, auseinander, bis letztlich ein Daten- oder

Stopp-Bit verpasst wird. Gemäß der Spezifikation des ATmega8515 generiert die MCU eine relativ genaue Baud-Rate von 19.200 mit einer maximalen Abweichung von 0,2%. Bei höheren Raten steigt der Fehler auf bis zu 8,5%. Letztlich spielt bei zeitkritischen Anwendungen, die eine beidseitige Kommunikation erfordern (Request-Response), noch die akzeptable Antwortverzögerung eine Rolle. Je mehr Zeit an die Datenübertragung verloren geht, desto weniger bleibt für die Rechenzeit der MCU.

In diesem Entwurfsschritt geht es vorrangig darum, eine RS-232-Schnittstelle zu realisieren. Daher wird auf eine Fehlererkennung mittels eines Paritäts-Bits verzichtet. Prinzipiell bietet der ATmega8515 eine Paritätsüberprüfung und eine Interrupt gesteuerte Fehlerbehandlung an. Schließlich stellt sich noch die Frage nach ein oder zwei Stopp-Bits. Einige UART-Komponenten erfordern 1,5 bis 2 Stopp-Bits zwischen zwei aufeinander folgenden Wörtern. Dies ist hier aber nicht der Fall. Nichts desto trotz kann durch die Wahl von zwei Stopp-Bits die Fehlerquote bei langen Frames reduziert werden (siehe Abschnitt RS-232). Es ist jedoch vorgesehen, nur sporadisch einzelne Wörter zu übertragen. Also fällt die Wahl auf ein Stopp-Bit.

Gemäß der MCU-Spezifikation wird dem Entwurf 1 eine init_rs232 Methode hinzugefügt, die den USART mit 19.200 Baud, 1 Stopp-Bit und keinem Paritäts-Bit initialisiert. Außerdem wird der Interrupt auf Receive Complete (RCX) aktiviert und im Folgenden die zugehörige ISR implementiert.

2.3.4.3 ISR

Sobald ein Wort über die serielle Schnittstelle eingegangen ist, soll die ISR vorerst lediglich den Blink-Modus abwechselnd ein- oder ausschalten, indem sie den Timer-Interrupt aktiviert beziehungsweise deaktiviert. Das empfangene Wort wird zwar ausgelesen, um den Empfangspuffer zu leeren, aber nicht interpretiert.

```
ISR(USART_RX_vect) {
    uint8_t data = UDR; //clear Rcv-Buff
    /*do something with data or following stuff: */
    if(timer_is_on) {  //stop blink (disable Intr)
        TIMSK &= ~_BV(OCIE1A);//disable Intr
        timer_is_on = OFF;
        PORTB &= ~_BV(PB0); //switch LED off
        led_is_on = OFF;
    }
    else {  //start blink
        TIMSK |= _BV(OCIE1A);
        timer_is_on = ON;
    }
}
```

Listing 2.27: ISR zur Behandlung von seriell empfangenen Daten

2.3.5 Entwurf 3a (MCU final) – Motorsteuerung

Nachdem durch Entwurf 1 und 2 geklärt ist, wie Timer und UART auf der Atmel-MCU initialisiert und benutzt werden, kann man sich der finalen Entwurfsphase zuwenden, der Realisierung einer Motorsteuerung.

Es sei an dieser Stelle nochmals angemerkt, dass es sich hier insgesamt nur um einen beispielhaften Entwurf handelt, um die Frage der Einbindung von Hardwarekomponenten wie MCUs in die Robotics Studio Architektur zu beleuchten. Daher wird Vieles nur vereinfacht realisiert und das Augenmerk auf die Kernaspekte dieses Themas gelenkt.

Folgende Tabelle listet die Motor-Funktionen, die die Steuerung unterstützen soll, auf.

Befehl	Beschreibung
Motor an	Setzt den Schrittmotor unter Spannung, sodass ein Haltemoment (Drehmoment im Stand) wirkt.
Motor aus	Setzt die Steuerspannung auf 0V. Der Motor ist frei drehbar.
Drehe_uzgs(Umdr./Sek.)	Der Motor dreht im Uhrzeigersinn mit der angegebenen Geschwindigkeit in Umdrehungen pro Sekunde.
Drehe_guzgs(Umdr./Sek.)	Dreht den Motor gegen den Uhrzeigersinn mit der angegebenen Geschwindigkeit.
Stopp	Beendet eine Drehung. Der Motor bleibt unter Spannung.

Tabelle 2.5: Steuerbefehle für den Motor-Controller

2.3.5.1 Protokoll

Bei der Verwendung peripherer, eingebetteter Hardware-Komponenten mit Robotics Studio gilt generell, dass unabhängig von der Kommunikationsart (TCP/IP, CAN, RS-232,…) ein Datenprotokoll entworfen werden muss, damit sich Service und Hardware verstehen. Schließlich ist die Komponente selbst kein DSS-Service, sondern wird von einem solchen benutzt und repräsentiert.

Analysiert man die oben angegeben Befehle aus Sicht des Controllers, kann man die Befehle „Motor an“ und „Stopp“ zusammenfassen. Die zum Zeitpunkt des Befehls aktuellen Steuerbits (ggf. Default-Werte) werden einfach beibehalten. Somit gibt es nur noch vier

verschiede Befehle, die sich durch zwei Bit codieren lassen. Über die serielle Schnittstelle können Wörter von fünf bis neun Bit Länge übertragen werden. Es sollen hier, wie im Entwurfsschritt 2, acht Bit sein, womit abzüglich der Befehlscodierung noch sechs Bit übrig bleiben. Diese können für den Geschwindigkeitsparameter der beiden Drehbefehle genutzt und als vorzeichenlose ganze Zahlen interpretiert werden. Dadurch werden Rotationsgeschwindigkeiten von 0 bis 63 U/s unterstützt.

Diese Entscheidungen beruhen lediglich auf der Intention, das Beispiel einfach zu halten. Ein einziges Wort beinhaltet so die Operation inklusive Parameter. Eine Übertragung multipler oder längerer Daten über mehrere Wörter verteilt, hat lediglich Einfluss auf Implementierungsdetails, nicht auf das Prinzip. Somit ergibt sich folgendes Bit-Protokoll:

Op1	**Op0**	**P5**	**P4**	**P3**	**P2**	**P1**	**P0**	**Befehl**
0	0	-	-	-	-	-	-	Motor aus
0	1	-	-	-	-	-	-	Motor an / Stopp
1	0	n_5	n_4	n_3	n_2	n_1	n_0	Drehe_uzgs(n) n=1..63 U/s
1	1	n_5	n_4	n_3	n_2	n_1	n_0	Drehe_guzgs(n) n=1..63 U/s

Tabelle 2.6: Befehlsprotokoll des Motor-Controllers

2.3.5.2 Zeitrestrektionen

Im vorigen Abschnitt wurde unreflektiert festgelegt, dass Rotationen mit 0 bis 63 U/s unterstützt werden. Ob dies tatsächlich zu realisieren ist hängt natürlich vom Schrittmotor selbst und seiner Belastung, aber auch vom Controller ab. Auf elektrotechnische Details des Motors soll an dieser Stelle nicht eingegangen werden, aber ein wenig auf das Zeitverhalten des Controllers.

Sei RPS (revolutions per seconds) die gewünschte Rotations-Geschwindigkeit und STP die Schrittzahl, die der Motor für eine Umdrehung benötigt. Die erforderliche Schrittfrequenz ergibt sich dann durch:

$$F_{STP} = RPS * STP\ Hz \qquad \text{(3) Schritt-Frequenz}$$

Eine über einen Timer getriggerte ISR soll die Steuersignale für den jeweils nächsten Schritt ausgeben. Der Timer wird dazu wie im Entwurf 1 im CRC-Modus betrieben. Dafür muss entsprechend der Gleichung (2b) ein geeigneter Prescaler gewählt und der TOP-Wert ermittelt werden:

$$TOP = \frac{F_{TC}}{F_{STP}} = \frac{F_{Sys}}{PRSC * F_{STP}} = \frac{8 * 10^6 Hz}{1 * RPS * 250Hz} = \frac{32000}{RPS}$$

(4) Counter-TOP in Abhängigkeit von rps

Die Gleichung (4) ist beispielhaft mit einer MCU-Frequenz von 8 MHz und einem Prescaler von 1 weitergeführt. Was sie aber vor allem veranschaulicht ist, dass dafür gesorgt sein muss, dass die TOP-Werte weder zu groß noch zu klein werden. Nicht zu groß, um innerhalb der verfügbaren Registergröße zu bleiben, und nicht zu klein, damit keine Interruptflut verursacht wird. Bei 1 bis 63 U/s ergeben sich hier für TOP Ergebnisse von 507,9 (gerundet zu 507) bis 32.000, was die 16-Bit Register nicht überschreitet. Doch wie sieht es mit der möglichen Interruptflut bei hohen Schrittfrequenzen aus?

Es gilt durch Messungen oder Codeanalyse zu ermitteln, wieviel Rechenlast die ISR beansprucht. Dazu wird hier auf einige Details der Implementierung eingegangen. Der Controller hat die Aufgaben, empfangene Befehle zu decodieren und entsprechend den Motor anzusteuern. Es wird eine verhältnismäßig geringe Befehlsrate erwartet, sodass der UART per Polling im Hauptprogramm abgefragt werden kann. Es gibt somit nur noch eine Timer-ISR zur Ansteuerung des Motors. Das Assembly beinhaltet etwa siebzig Befehle für die ISR, was aufgrund der RISC-Architektur der MCU auch etwa siebzig Takte Rechenzeit entspricht, die wir großzügig auf einhundert aufrunden. Desweiteren wird festgelegt, das 50% Auslastung durch die ISR akzeptabel sei. Mit diesen Angaben kann dann eine maximale Schrittfrequenz F_{STP} beziehungsweise schnellste Umdrehung U_{max} angegeben werden:

$$F_{STP} = F_{ISR} \leq 50\% * \frac{1}{100 T_{Sys}} = 0{,}5\% * F_{Sys}$$

(5a) maximale Schrittfrequenz

$$U_{max} = F_{STP} * STP = \frac{F_{Sys}}{200 * STP}$$

$$\Rightarrow U_{max} = \frac{8Mhz}{200 * 250} = 160\, {}^{U}/_{s}$$

(5b) maximale Rotationsgeschw.

2.3.5.3 Genauigkeit

Durch Abrundungen bei der Berechnung der Timerwerte kann die generierte Geschwindigkeit von der angeforderten abweichen, sie ist potentiell etwas größer. Es ist folglich angebracht, die maximale Abweichung als Kenngröße des Controllers anzugeben.

Die Ungenauigkeit entsteht bei der Berechnung von TOP und wird tendenziell umso größer, je schneller sich der Motor drehen soll (siehe Glg. 4). Näherungsweise lässt sich die Abweichung d folgend berechnen:

$$d = 1 - \left\lfloor \frac{F_{TC}}{RPS_{max} * STP\ Hz} \right\rfloor \Big/ \left\lceil \frac{F_{TC}}{RPS_{max} * STP\ Hz} \right\rceil \qquad \text{(6a) Abweichung der Geschw.}$$

$$d = 1 - \left\lfloor \frac{32.000}{63} \right\rfloor \Big/ \left\lceil \frac{32.000}{63} \right\rceil = 1 - \frac{507}{508} \cong 0{,}2\% \qquad \text{(6b) Beispiel Abweichung}$$

Nach Gleichung 6b gilt für diesen Controller eine maximale Abweichung von +0,2%.

2.3.5.4 Entwurfsergebnis

Das Aktivitätsdiagramm zeigt die Realisierung der Motor-Steuerung seitens der MCU. Die Steuersignale an den Motor sind in ein globales Array hart codiert, sodass die ISR lediglich den Pointer darauf je nach Drehrichtung de- bzw. inkrementieren und den eingetragenen Wert auf den Ausgang legen muss.

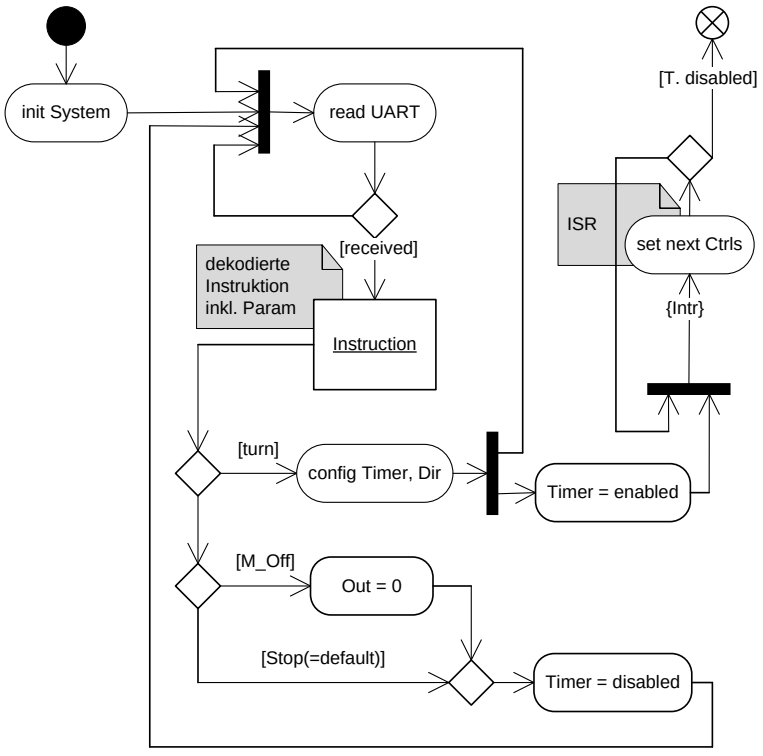


Abbildung 2.34: Aktivitätsdiagramm Motor-Controller

2.3.6 Entwurf 3b – Motorservice

Eine typische Aufgabe eines DSS-Service für Roboter-Anwendungen ist es, eine Hardware-Komponente zu repräsentieren. Sei es ein vollständiger, ferngesteuerter Roboter oder, wie in dem vorangegangenen Beispiel, ein einzelnes Gerät, den Motor. Um den Motor aus Entwurf 3a, bzw. dessen Controller, in eine RDS-Anwendung zu integrieren, muss nun ein Service implementiert werden, der die serielle Ansteuerung kapselt und als Service-Operation anderen Diensten anbietet.

2.3.6.1 .NET SerialPort

Es ist nicht weiter schwer, in C# eine serielle RS-232 Schnittstelle zu implementieren, denn seit .NET 2.0 ist die Klasse SerialPort im Namespace System.IO.Ports Bestandteil des Basisframeworks und kann selbstverständlich in einem DSS-Service benutzt werden. Folgende Tabelle listetet eine Auswahl für uns nützlicher Klassen-Member auf:

Member	Beschreibung
BaudRate : int	Gibt/setzt die Baud Rate
DataBits : int	Gibt/setzt Anzahl der Daten-Bits
StopBits : StopBits	Gibt/setzt Anzahl Stopp-Bits
Parity : Parity	Gibt/setzt Paritätsprotokoll (default None)
Encoding : Encoding	Gibt/setzt Codierung des Textes (default ASCIIEncoding)
DataReceived	Event, das ausgelöst wird wenn Daten eingegangen sind
Open() : void	Öffnet die serielle Portverbindung
Close() : void	Schließt die serielle Verbindung
Read(byte[] buf, int offset, int count) : int	Liest die angegebene Anzahl von Bytes aus dem Empfangspuffer ins angegebene Array.
Write(byte[], int, int) : void	Schreibt aus dem Array die angebene Anzahl Bytes.

Tabelle 2.7: Klassenmember System.IO.Ports.SerialPort

Enum StopBits	
None	Kein Stopp-Bit
One	1
Two	2
OnePointFive	1,5 (Zeitdauer, ein Bit zu senden)

Tabelle 2.8: System.IO.Ports.StopBits

Enum Parity	
None	Keine Paritätsprüfung
Odd	Ungerade Anzahl „Einsen“
Even	Gerade Anzahl „Einsen“
Mark	Paritätsbit ist immer gesetzt
Space	Paritätsbit ist immer gelöscht

Tabelle 2.9: System.IO.Ports.Parity

2.3.6.2 Service Design

Als Repräsentant für den Motor soll der DSS-Service dieselben Operationen wie der Controller und zusätzlich den Benachrichtigungsdienst zur Verfügung stellen. Die entwickelte Steuereinheit hat einige Restriktionen was die Geschwindigkeit angeht, nur ganze Zahlen im Bereich +/-63, und muss über die serielle Schnittstelle angeschlossen werden. Um eine Weiterentwicklung oder einen Austausch des Controllers zu unterstützen, bietet sich ein generischer Service an. Auf diese Weise kann der spezialisierte Service jederzeit ausgetauscht werden, ohne die Orchestration zu beeinflussen.

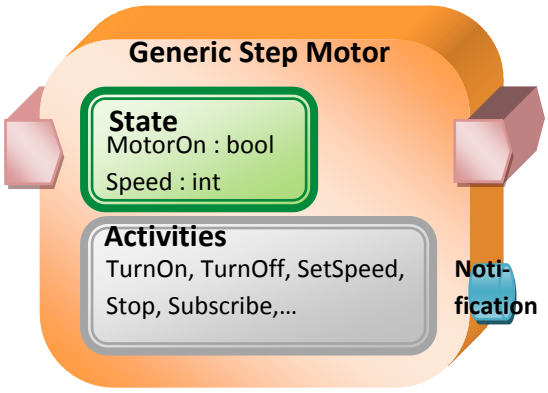


Abbildung 2.35: Übersicht Generischer Motor Service

2.3.6.3 Service Implementierung

Im generischen Motor Service werden gemäß Abbildung 2.35 Zustand, Operationen und dafür notwendige Requesttypen[13] definiert. Der Service „SerialStepMotor“ implementiert den generischen speziell für den Controller aus Entwurf 3a.

Für die **serielle Kommunikation** sind Hilfsmethoden implementiert. Die erste initialisiert den Port, die zweite ist für das Senden zuständig. In diesem Beispiel ist es nicht nötig, auch zu lesen, da der Controller kein Feedback sendet. Ansonsten sollte in einem separaten Thread gelesen oder die serielle Kommunikation komplett in einen eigenen Service ausgelagert werden.

Die Einstellungen für den seriellen Port sind in ein config-File geschrieben, sodass Änderungen vorgenommen werden können, ohne neu kompilieren zu müssen. Die Initialisierungsmethode wird noch vor `base.Start()`, also vor allen anderen Initialisierungen, aufgerufen und beendet bei einem Fehler den Service mit einer Exception.

```
private void initSerialPort() {
    try {
        NameValueCollection appSettings = ConfigurationManager.AppSettings;
        string name = appSettings["PortName"];
        int baud = Int32.Parse(appSettings["BaudRate"]);
        int data = Int32.Parse(appSettings["DataBits"]);
        Parity parity = (Parity)Int32.Parse(appSettings["Parity"]);
        StopBits stop = (StopBits)Int32.Parse(appSettings["StopBits"]);

        _serialPort = new SerialPort(name, baud, parity, data, stop);
        _serialPort.Open();
    }
    catch (Exception e) {
        LogError("Failed to initilize serial port", e);
        throw;
    }
}
```

Listing 2.28: Initialisierung des seriellen Ports per config-Datei

In den **ServiceHandlern** gilt es, die Befehle in das Protokoll des Controllers umzuwandeln und ihm seriell zuzusenden (siehe Tabelle 2.6: Befehlsprotokoll des Motor-Controllers). Um dies zu vereinfachen, sind die Operations-Bitmasken im enum-Typ **ByteOperation** festgehalten. Dem aufmerksamen Leser ist vielleicht bereits aufgefallen, dass der generische Service nur die Operation ***SetSpeed*** anstelle von *Drehe_uzgs* und *Drehe_guzgs* anbietet. Diese Designentscheidung wurde zwecks größerer Verallgemeinerung gefällt und wird im

[13] Z.B. SetSpeedRequest mit einer Speed-Property

SerialStepMotor so umgesetzt, dass negative Werte einer Drehung gegen, positive Zahlen einer mit den Uhrzeigersinn entsprechen. Die Implementierung eines ServiceHandlers ist folgend beispielhaft für SetSpeed gezeigt.

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> SetSpeedHandler(genSM.SetSpeed request) {
    //check, if motor is on:
    if (!_state.MotorOn) {
        LogWarning("SetSpeed request with Motor's state = off");
        ReasonText[] reasons = {
            new ReasonText { Lang="en", Value="Motor is turned off"},
            new ReasonText { Lang="de", Value="Motor ist ausgeschalted"}};
        request.ResponsePort.Post(new Fault { Reason = reasons });
        yield break; //finished with failure
    }
    //create and send instruction:
    int speed = request.Body.Speed; //requested speed
    int instruction;
    if (speed >= 0) {
        if (speed >= 63) { //check if speed is in range
            speed = 63;
            LogWarning("That's too fast! Speed reduced to 63 rps.");
        }
        instruction = (int)ByteInstruction.TurnClock | speed;
    }
    else {
        if (speed <= 63) { //check if speed is in range
            speed = -63;
            LogWarning("That's too fast! Speed reduced to -63 rps.");
        }
        instruction = (int)ByteInstruction.TurnCounter | -speed;
    }
    if(!writeSerial((byte)instruction,"SetSpeed Error",request.ResponsePort.P1))
        yield break;
    //change state, respond, notify subscribers:
    _state.Speed = speed;
    request.ResponsePort.Post(DefaultUpdateResponseType.Instance); //success
    base.SendNotification<genSM.SetSpeed>(_submgrPort, request);
    yield break;
}
```

Listing 2.29: SetSpeedHandler, schickt den Befehl seriell an den Motor-Controller

Anmerkung:
Die Ausführung des Handlers ist nur wegen der Zustandsmanipulation exklusiv. Die Gefahr, dass zwei Tasks gleichzeitig auf den seriellen Bus schreiben, besteht nicht, da sie jeweils nur ein einzelnes Byte senden, das vor der eigentlichen Übertragung in einem Puffer zwischen-gespeichert wird.

2.3.7 Anwendung

Der generische und der implementierende Service für den Schrittmotor kann nun von jeder Anwendung benutzt werden. Ob sie in C# geschrieben, mit dem Manifest Editor zusammengestellt oder, wie in diesem Beispiel, mit VPL „gezeichnet“ wird.

Mittels des in Robotics Studio enthaltenen Service *ControlDialog* kann man eine simple graphische Benutzeroberfläche erstellen, über die hier der Motor bedient werden soll. Sie beinhaltet ein Textfeld zur Eingabe der Drehgeschwindigkeit und je einen Button zum Übernehmen der Geschwindigkeit, zum Anschalten, zum Abschalten und zum Anhalten des Motors. Dem generischen Service *GenericStepMotor* ist per Manifest der implementierende Service *SerialStepMotor* zugewiesen.

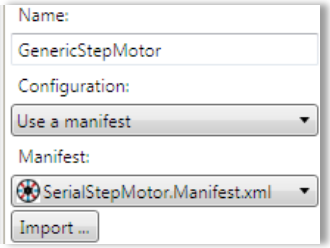


Abbildung 2.36: Konfiguration per Manifest

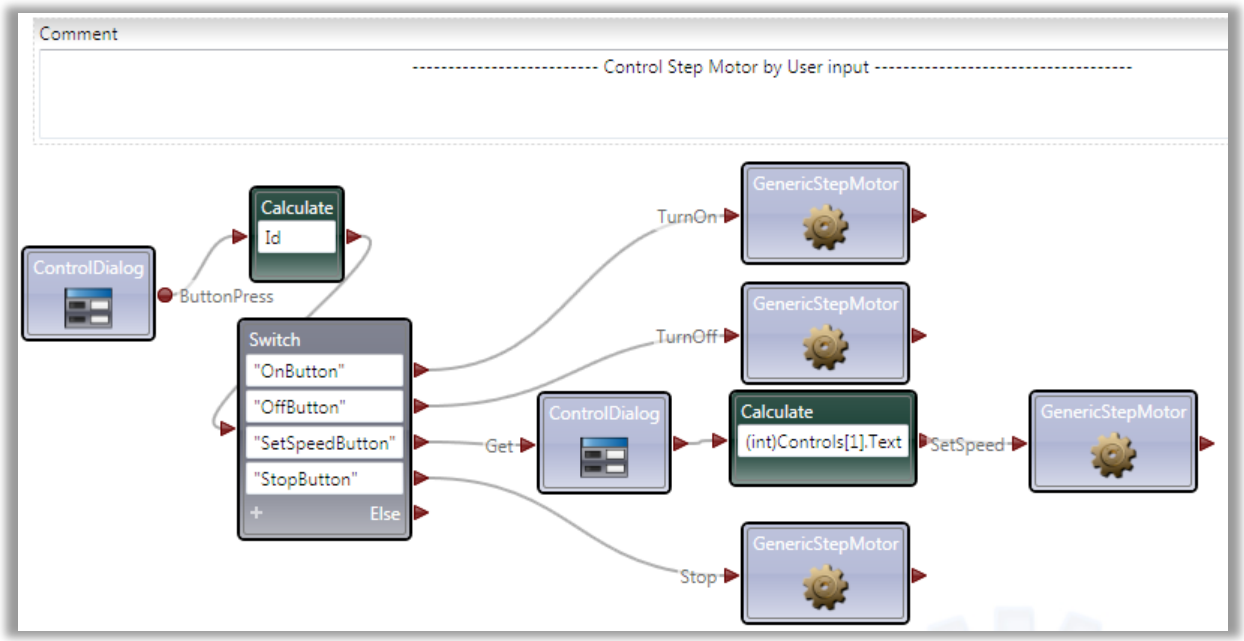


Abbildung 2.37: StepMotor Anwendung in VPL, Teil 1

VPL ist als ein Datenflussdiagramm zu lesen, das eine kleine Anzahl von Elementen zur Daten-Manipulation und –Erzeugung, sowie alle mit installierten und selbst erstellten Services anbietet. Der Datenfluss von einem Element zu einem anderen wird durch

Verbindungslinien dargestellt. Services sind in VPL als hellblaue Rechtecke mit Icon dargestellt, deren Aktivitäten per Request angefordert werden können, auf die dann eine Response erfolgen kann. Beides sind Nachrichten, die wiederum nichts anderes als Daten sind, die ein-, respektive ausgehen. Dies wird durch rote Pfeilspitzen an den Services symbolisiert. Rote Kreise bilden den Datenausgang für Notifications.

Wird in der GUI des *ControlDialog* ein Button betätigt, so sendet der Service die Notification „ButtonPress“. Sie beinhaltet den Datenmember „Id“ mit dem Namen des Buttons der gedrückt wurde. Über das Switch-Element teilt sich der Datenpfad nun in Abhängigkeit von „Id“ und der entsprechende Befehl (die Request) wird dem *GenericStepMotor* erteilt. Lediglich beim „SetSpeedButton“ müssen noch einige Zwischenschritte erledigt werden, um den Wert aus dem Textfeld der GUI auszulesen. Dies geschieht über eine „Get“ Request an die GUI und anschließendem parsen zu int, sodass der Wert in die „SetSpeed“ Request eingebunden werden kann. Man beachte in der Abbildung, dass jeder *GenericStepMotor* ein und dieselbe Instanz ist. Dies kann für den Anfänger noch recht verwirrend sein, ist aber gut geeignet, um voneinander unabhängige oder parallele Datenpfade darzustellen.

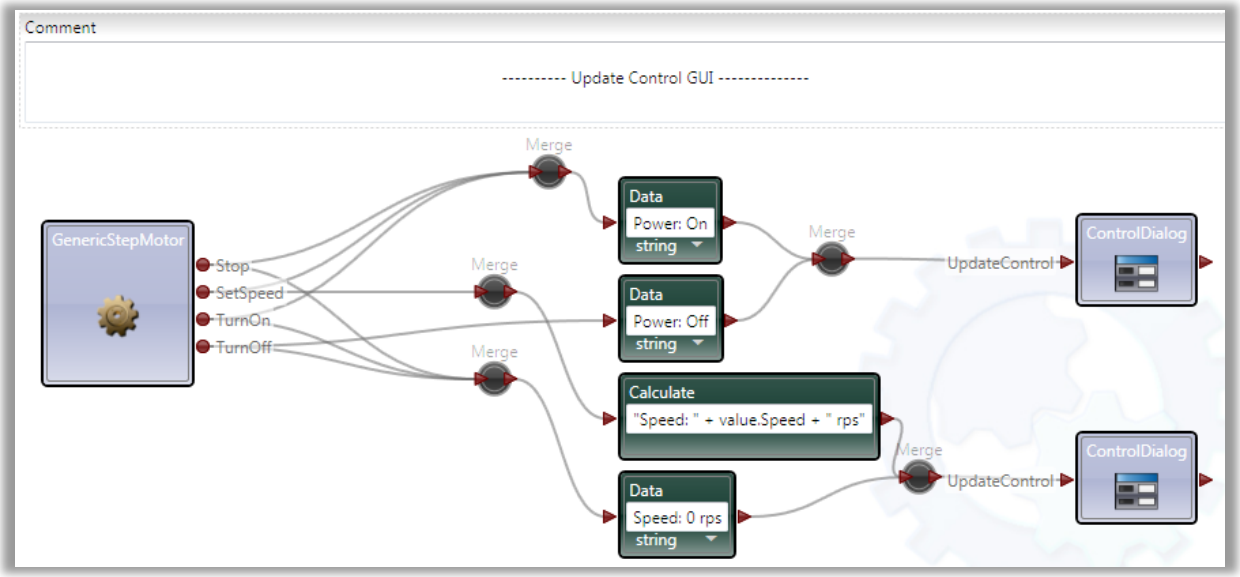


Abbildung 2.38: StepMotor Anwendung in VPL, Teil 2

Eine Zustandsänderung des Schrittmotors, herbeigeführt durch eine Request, wird vom Service per Notification mitgeteilt. Dies nutzt die Anwendung aus, um die Statusanzeige im *ControlDialog* zu aktualisieren. Die *Merge* Elemente in Abbildung 2.38 vereinen Datenpfade und reichen das eingehende Datum unverzüglich weiter. In den *Data* Elementen werden Strings zur Anzeige im *ControlDialog* erzeugt, *Calculate* berechnet diesen mit dem in der „SetSpeed“ Notification enthaltenen Wert „Speed“. Damit ist die Anwendung vollständig.

3 Resümee

Die ersten Kapitel haben die An- und Herausforderungen an Rescue Robots im Allgemeinen und AMEE im Speziellen umrissen. Der erste Prototyp muss unter anderem in der Lage sein, sich auf vier Beinen fortzubewegen und die Umgebung selbständig zu erkunden und zu kartographieren. Die Daten sollen an ein entferntes Operator-System übertragen werden, von dem auch außerordentliche Steuerbefehle eingehen können. Anschließend beschrieben die Kapitelabschnitte CCR und DSS das dem Robotics Studio zugrunde liegende Programmier- und Architekturmodell und gaben einen Einblick, wie damit eine Service Orientierte Roboteranwendung modelliert und programmiert werden kann. Die beispielhafte Integration eines Schrittmotors demonstrierte, wie separate Hardwarekomponenten in die Architektur einzubinden und durch Services zu abstrahieren sind.

Als Ergebnis der erfolgten Untersuchung der Möglichkeiten von Robotics Studio für USAR-Roboter gibt der Abschnitt „AMEE und DSS“ zunächst einen Ausblick, wie eine DSS-Anwendung für AMEE gestaltet sein könnte. Eine abschließende Beurteilung bewertet das Robotics Studio für den künftigen Einsatz im Projekt.

3.1 AMEE und DSS

Ein Ausblick

Eine erste grobe Vision zeigt, wie DSS für AMEE eingesetzt werden kann. Die Abbildung 3.1 zeigt dazu eine Systemübersicht. Hardwarekomponenten wie MicroController sind als Raute dargestellt, die Windowssysteme mit einer DSS-Laufzeitumgebung als Rechtecke. Als Bussystem wird, auch zur Kommunikation mit den Controllern, Ethernet benutzt; Operator-system und AMEE sind per WLAN miteinander verbunden. Eine genauere Beschreibung und Begründung zum Systemaufbau ist in der (leider noch nicht fertigen) Bachelorarbeit von Jan Ruhnke nachzulesen, in der er sich mit einem Laufsystem für vierbeinige Roboter auseinandersetzt. Entsprechend seiner bisherigen Ergebnisse ist die Logik verteilt auf vier MicroController und einem Embedded-Board mit beispielsweise Win CE OS, hier als große Raute dargestellt. Denn trotz Windows OS wird dort kein DSS Knoten laufen. Aus DSS-Sicht ist das Board mit dem Laufsystem ebenso eine Hardwarekomponente wie ein Motorcontroller.

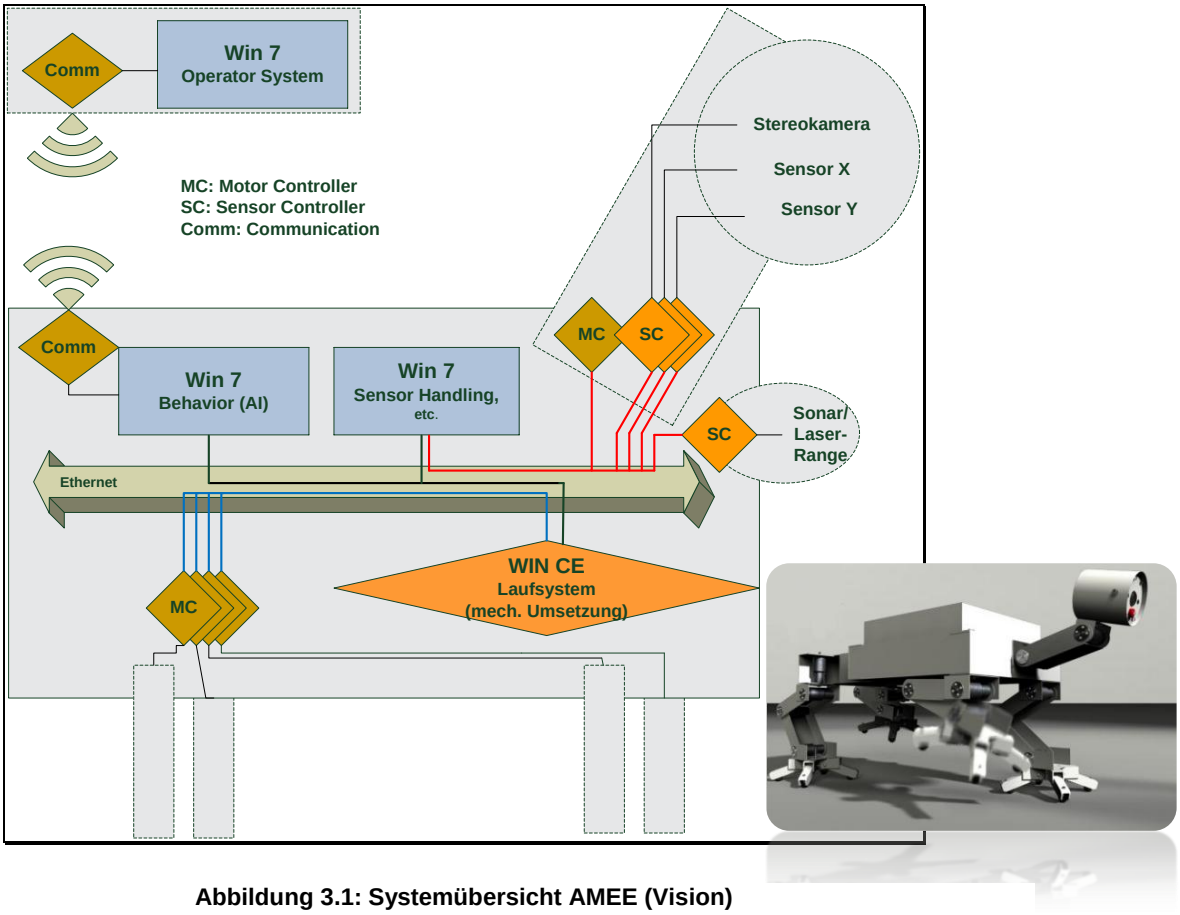


Abbildung 3.1: Systemübersicht AMEE (Vision)

Das Servicedesign (siehe Abbildung 3.2) beschränkt sich hier vorerst auf die Mobilität des Roboters inklusive einer Visualisierung und Steuerung durch ein entferntes Operatorsystem. Jede Hardwarekomponente ist durch einen Service repräsentiert, seien es die Sensoren, das Laufsystem oder die Datenspeichereinheit (Datenbank, Filesystem,…). Durch Orchestration erreicht man sukzessive das letztlich gewünschte Verhalten. So könnte ein Service „ImagePreprocessing“ Merkmalsextraktionen oder Kantenalgorithmen auf die Kamerabilder anwenden. Das aufgearbeitete Bild dient dann gemeinsam mit den Daten eines „LaserRangeSensors“ der Kartographierung durch ein „Mapping“-Service. Es ist anzunehmen, dass eine persistente Datenspeicherung von Nutzen ist. Ein generischer „Storage“-Service könnte dies anbieten und die Wahl des Speichermediums offen halten. Anhand der Mapping-Daten und übergeordneter Aufträge („gehe einen Pfad“, „finde Ausgang“,…) berechnet der Service „MotionPlanning“ die nächsten Pfadpunkte, die mittels des durch „MotionControl“ repräsentierten Laufsystems in Bewegung umgesetzt werden. Ein außerplanmäßiger Kontakt mit Hindernissen meldet ein „BumperService“, welcher aufgrund seiner Sonderrolle direkten Einfluss auf „MotionControl“ hat. Der Roboter AMEE bildet im

Gesamtsystem wiederum nur eine Komponente und wird durch den Service „AMEE“ repräsentiert. Dieser bildet die Schnittstelle zum Operatorsystem, gekapselt durch den Service „Operator“. Die von AMEE erfassten Daten werden auf diesem System visualisiert und gespeichert. Über die graphische Oberfläche und angeschlossene (Game-)Controller kann der Benutzer Befehle an AMEE geben.

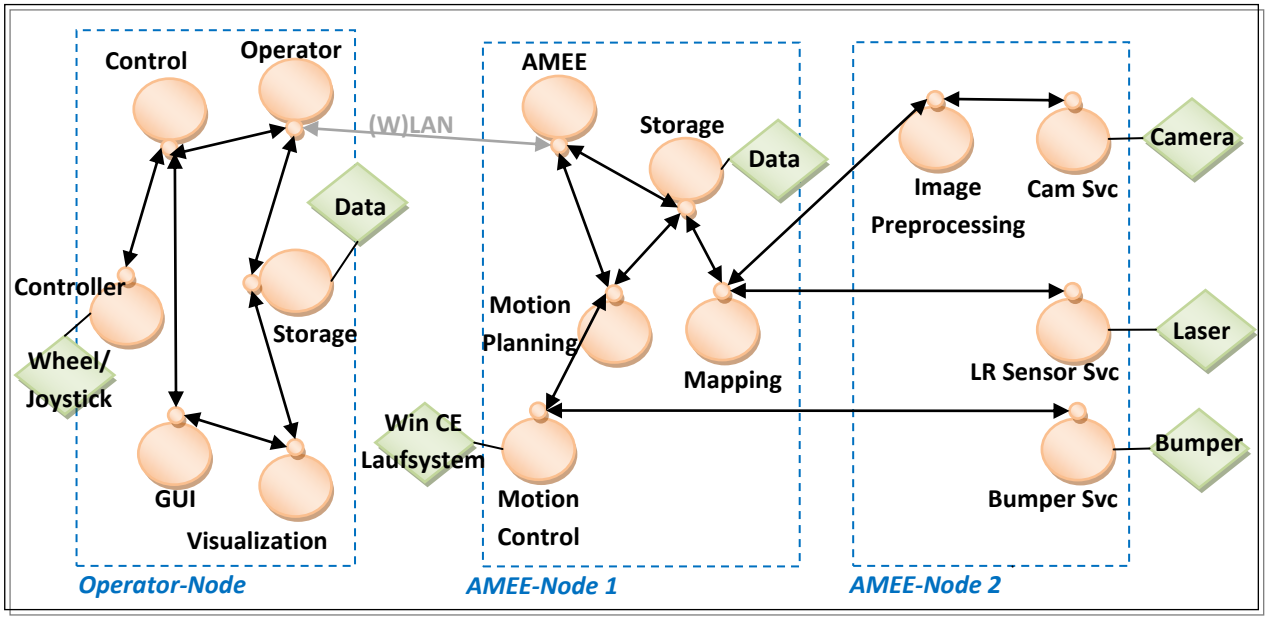


Abbildung 3.2: DSS-Designübersicht AMEE (Vision)

Wie der Abbildung 3.1 zu entnehmen ist, sieht diese Vision zwei Win7-Rechner und somit auch zwei DSS-Nodes auf dem Roboter vor (siehe Abbildung 3.2). Dieses „Devide & Conquer“ oder „viel hilft viel“ Prinzip beruht darauf, dass mehr Prozessorkerne zu einer besseren Lastverteilung führen. Der nahezu lineare Performanzgewinn wurde im CCR-Kapitel gezeigt. Die hier verwendete Strategie ist, hochfrequente Berechnungen niederer Funktionen auf einer Recheneinheit und Daten-/Zeitintensive Berechnungen höherer Funktionen auf einer anderen Recheneinheit anzusiedeln, um eine möglichst optimale Balance zu erzielen. Ein großer Vorteil von DSS ist, dass die Konfiguration des Systems ohne großen Aufwand durch Manifeste immer wieder verändert, die Services zwischen den Knoten hin und her „geschoben“ werden können, um letztlich die beste Verteilung zu erzielen. Prinzipiell ist dies sogar dynamisch zur Laufzeit möglich, was aber fortgeschrittene Kenntnisse zu DSS und .NET erfordert.

3.2 Beurteilung

Ziel dieser Arbeit ist im Wesentlichen eine Beurteilung des potentiellen Einsatzes von Robotis Developer Studio (RDS) für das Rescue Robot Projekt AMEE zu erstellen. Die Erkenntnisse dieser Arbeit und das folgende Fazit sind in vielerlei Hinsicht aber auch generell auf Feldroboter anzuwenden.

3.2.1 CCR & DSS

Die Concurrency and Coordination Runtime ist kein genereller Ersatz für Threads, Locks, Delegates und Callbacks oder dergleichen. Hauptaugenmerk und somit auch wesentlicher Nutzen von CCR liegen vielmehr bei der Koordination asynchroner Prozesse. Sie bietet eine vereinfachte Handhabung und einigen „Syntactic Sugar“ für eine sonst recht komplexe und fehleranfällige Thematik. Der doch erhebliche Vorteil von CCR wird besonders deutlich, versuchte man dieselben Funktionaltäten mit herkömmlichen Sprachmittel von C# nachzustellen. Unabhängig davon, dass diese Library Basis für die Programmierung von DSS-Services ist, ist der Einsatz von CCR gerade für Roboteranwendungen zu empfehlen.

Eine Architektur lose gekoppelter Module, auf gängigen Netzwerktechnologien aufgesetzt, für Robotersystem einzusetzen, kommt der Natur der Robotik insofern entgegen, dass insbesondere Feld-Roboter wie AMEE an sich schon aus mehreren Hardware-Komponenten zusammengebaut sind und zusätzlich mit Fernsteuerungs- und Fernwartungssystemen interagieren. Das Konzept der Decentralized Software Services ist daher so naheliegend wie plausibel. Der hervorstechendste Nutzen von DSS liegt im Entwicklungsdesign einer Roboteranwendung und der Flexibilität der Architektur. Das Framework insgesamt und ein einzelner Service im Speziellen sind übersichtlich und klar strukturiert. Im Zusammenhang mit CCR kann sich der Entwickler im Wesentlichen auf die Semantik und Logik der Anwendung konzentrieren. Nachteilig wirkt sich dagegen die Notwendigkeit einer Common Language Runtime aus.

3.2.2 Manifest Editor, VPL & Simulation

Das Robotics Studio besteht nicht nur aus dem CCR & DSS Toolkit. Vielmehr ist es eine Gesamtlösung, die unter anderem auch Authoring, Distribution und Simulation Tools beinhaltet, welche in dieser Arbeit aber nicht näher betrachtet wurden.

Der Manifest Editor ist ein graphisches Tool, mit dem sich das System sehr leicht konfigurieren lässt, indem Services per Drag & Drop zu einer Gesamtanwendung zusammengestellt werden. Es ist möglich, dies für verteilte Rechner zu erstellen, Services mit gespeicherten Daten zu initialisieren und ein „Deploy Package“ erstellen zu lassen.

Eine Datenfluss Orientierte, visuelle Sprache, die Visual Programming Language (VPL, siehe auch Abschnitt 2.3.7), ist gut für den Einstieg in Robotics Studio geeignet, und ermöglicht eine graphische Orchestration der Services zu einer Roboteranwendung. Services und einfache Berechnungen können zu neuen „Activities“ zusammengestellt und beliebig verschachtelt werden. VPL ist auch gut geeignet, um die Zusammenhänge zwischen Services und parallele Abläufe zu visualisieren und somit das Verständnis der Gesamtanwendung zu verbessern.

Ein 3D Simulationstool mit Physics Engine ermöglicht die eigene Anwendung an mitgelieferten oder selbst erstellten Robotermodellen zu testen. Dies schont nicht nur die Hardware, sondern auch das Projektbudget und ermöglicht parallele Tests, auch wenn nur einer oder noch gar kein physischer Roboter zur Verfügung steht.

3.2.3 Robotics Studio für AMEE

AMEE ist ein autonomer Roboter, der aufgrund der Einsatzszenerie komplexe Aufgaben bewältigen muss, auch wenn der Kontakt zum Operator-/Monitoring-System für längere Zeiträume unterbrochen ist. Es können also nicht alle rechenintensiven Aufgaben ausgelagert werden, AMEE ist auf relativ leistungsstarke Rechner onboard angewiesen. Einer der größten Nachteile von RDS für autonome Roboter ist, dass es auf der CLR aufgebaut ist und somit ein Windows-OS erfordert. Da nach einer groben Voreinschätzung der notwendigen Ressourcen aber sowieso Windows fähige Rechner eingesetzt werden sollen, erübrigt sich dieser Aspekt. Unter diesen Voraussetzungen kommt dann hinzu, dass die CLR aufgrund ihrer Entwicklung und Optimierung für Windows Betriebssysteme mit die performanteste Laufzeitumgebung ist. Es gilt jetzt aber noch zu bedenken, dass RDS nur bis zur Version 2008 R2 auch das Compact Framework unterstützt.

Was bedeutet das, wenn man trotzdem beispielsweise Windows CE einsetzen möchte? Es heißt lediglich, dass dort kein DSS-Node also auch keine DSS-Services laufen können. Nichts desto trotz könnten dort separate Komponenten angesiedelt werden, die in der gesamten Verhaltensarchitektur hierarchisch eine mittlere Rolle haben und letztlich durch ein DSS-Service auf dem Hauptrechner repräsentiert werden. Ob der Service nun über serielle, parallele oder Ethernet-Schnittstelle auf die Hardware zugreift, ist prinzipiell egal. Die gleiche Überlegung gilt auch für Coprozessor-Systeme oder dem ergänzenden Einsatz eines RTOS. Dies kann oft sinnvoll sein, widerspricht aber nicht der generellen Benutzung von DSS.

Die Grundvoraussetzungen für Robotics Studio sind also gegeben. Zusammenfassend mit den oben und im DSS-Kapitel genannten Vorteilen von CLR und DSS gibt es Einiges, das für Robotcs Studio spricht:

- Die Einarbeitung ist aufgrund des übersichtlichen Programmier- und Architektur-Modells, sowie einer Vielzahl leicht verständlicher Tutorials, Beispielen und Dokumentationen in maximal zwei Wochen möglich.
- Robotics Studio bietet eine einheitliche und weitgehend generische Gesamtlösung für die Entwicklung von Roboteranwendungen.
    - Einfacher Austausch von Algorithmen und Hardware, durch die Architektur lose gekoppelter und generischer Services
    - Erhöhte Möglichkeit der parallelen Entwicklung
    - Unabhängigkeit von herstellerspezifischen Programmierlösungen (CT'Bot, Pioneer, Lego, IRobot,...)
    - Dynamische Entwicklung, da weitere Ergänzungen parallel und vor allem auch zu späteren Entwurfsphasen entwickelt und hinzugefügt werden können
- gute Skalierbarkeit des Systems:
    - mehr Prozessoren führen zu einem nahezu linearen Geschwindigkeitsvorteil
    - Services lassen sich beliebig hinzufügen und entfernen, egal auf welchem Knoten

- Einfaches Monitoring/Debugging
    - Mit einem beliebigen Webbrowser ist es möglich, das laufende System zu beobachten und manuell einzugreifen. Es ist keine eigene Software dafür notwendig.
    - Es kann zusätzlich der viel umfangreichere VS-Debugger benutzt werden.
- Quick Prototyping und hervorragende Visualisierung der Anwendungsarchitektur und nebenläufiger Prozesse durch VPL.
- Ein Simulationstool mit 3D Physics Engine bietet Simulations- und Testmöglichkeiten und beschleunigt den Entwicklungsprozess.
- Der Entwicklungsprozess von und die Community zu Robotics Studio ist lebendig. Es ist anzunehmen, dass dies auch in mittelbarer Zukunft noch gilt.
    - Namhafte Unternehmen setzen Robotics Studio ein oder unterstützen es (Microsoft Corporation, 2010f), es gibt also wirtschaftlichen Nutzen und Feedback. Robotics Studio ist kein Labor-Spielzeug!

Typisch für Service Orientierte Architekturen ist, dass sie einen hohen Initialaufwand haben, da viel Arbeit in die Identifizierung und Entkopplung von Diensten gesteckt werden muss. Die von einem Service angebotenen Aktivitäten sollten allgemein genug gehalten werden, um einen hohen Grad an Wiederverwendbarkeit, Adaptivität und Flexibilität zu erzielen und gleichzeitig speziell genug sein, um sie effektiv einsetzen zu können. Für Roboteranwendungen mit DSS spielt der Grad der Granularität ebenfalls eine wichtige Rolle. Normalerweise gibt es je DSS-Node einen Threadpool (Dispatcher) und je Service eine Dispatcherqueue. Da alle Queues und somit alle Services vom Scheduler gleichberechtigt behandelt werden, sollten beispielsweise seltene aber wichtige Tasks in einem separaten Service erzeugt werden. Es kann also nicht generell gesagt werden, dass eine grobe Granularität wegen höherer Unabhängigkeit in verteilten Systemen, oder eine feine Granularität wegen des besseren Loadbalancing zu bevorzugen ist. Beides gilt es stets sorgfältig abzuwägen.

Man sollte auch nicht davon ausgehen, RDS als alleiniges Entwicklungswerkzeug benutzen zu können. Es gibt mit hoher Sicherheit gerade im Bereich der Sensordatenverarbeitung immer wieder Situationen, in denen es angebracht ist, eingebettete Assistenzsysteme zu entwerfen oder einzusetzen, eventuell inklusive der Notwendigkeit, Windows Treiber zu programmieren, damit man diese Systeme in RDS integrieren kann.

Berücksichtigt man die Anforderung an das Design und die Einschränkungen von RDS in Hinsicht auf hard realtime Anwendungen, so können oben genannte Vorteile einen nachhaltigen Mehrnutzen für die Entwicklung von Roboteranwendungen bringen und machen Robotics Developer Studio zu einer interessanten und erfolgsversprechenden Lösung, nicht nur in der Robotik.

3.3 Offene Fragen

Wenn RDS als Gesamtlösung eingesetzt werden soll, ist es unbedingt ratsam, Möglichkeiten und Umfang des Simulationstools zu untersuchen. Zwar werden Umgebungen und Robotermodelle wie der Logo NXT Mindstorm oder ein Sechs-Achsen-Roboterarm von Kuka mitgeliefert. Einen wirklichen Nutzen erzielt man aber erst durch die Erstellung oder Integration eigener Modelle. Mit Tools wie MatLab und SimuLink lassen sich gut dynamische Prozesse modellieren. Es wäre von besonderem Interesse zu wissen, ob der Simulator von Robotics Studio ähnliches ermöglicht oder zumindest auf eine möglichst effiziente Weise mit zuvor genannten Tools zu verknüpfen ist.

Der im Zuge dieser Arbeit durchgeführte Benchmark zum Nachrichtendurchsatz blieb weit hinter den Erwartungen zurück. Die Ursachen konnten bis dato nicht geklärt und sollten weiter untersucht werden. Die aktuellen Testergebnisse sind bei der Vielzahl und Häufigkeit von Sensordaten eines RescueRobots so jedenfalls nicht akzeptabel.

Das von DSS standardmäßig genutzte Übertragungsprotokoll ist TCP. Laut der Dokumentation zur Einführung in DSS werden auch andere Protokolle wie UDP oder gar gänzlich eigene unterstützt. Es ist zwar einfach und bequem, sich auf Ethernet und TCP/IP zu beschränken. Unter Umständen kann es aber angebracht sein, andere Bussysteme zu nutzen. Wie dies umzusetzen ist, wäre also eine nähere Betrachtung wert.

Literaturverzeichnis

Atmel Corporation. 2010. *ATmegea8515 Specification.* 2010.

Basener, Andreas. 2008. *Integration des c't-Bots in das RoboticsDeveloperStudio.* Hamburg : Hochschule für Angewandte Wissenschaften Hamburg, 2008.

Brooks, R. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation, Vol. 2.* New York : IEEE Journal, 1986, pp. 14-23.

Chrysanthakopoulos, George. 2008. Podcast 2008, The CCR and DSS Toolkit. *Channel9.* [Online] November 2008. [Cited: Juni 5, 2010.] http://channel9.msdn.com/pdc2008/TL55/.

Hoffman, Antony. 2000. *Red Planet.* Warner Bros. Pictures, 2000.

Jackson, Jared. 2007. Microsoft Robotics Studio: A Technical Introduction. *IEEE Robotics & Automation Magazine Vol. 14.* New York : IEEE, 2007.

Microsoft Corporation. 2010f. *Microsoft Robotics Developer Studio.* [Online] Microsoft Corporation, 2010f. [Cited: August 9, 2010.] http://www.microsoft.com/robotics/.

—. 2010a. Asynchronous Programming Design Patterns. *MSDN Library.* [Online] 2010a. [Cited: August 7, 2010.] http://msdn.microsoft.com/en-us/library/ms228969(v=VS.80).aspx.

—. 2010b. Concurrency and Coordination Runtime. *MSDN Library.* [Online] 2010b. [Cited: Juli 14, 2010.] http://msdn.microsoft.com/en-us/library/bb905470.aspx.

—. 2010c. DSS Service Components. *MSDN Library.* [Online] 2010c. [Cited: Juni 23, 2010.] http://msdn.microsoft.com/en-us/library/bb483067.aspx.

—. 2010d. DSS Subscription Model. *MSDN Library.* [Online] 2010d. [Cited: Juni 25, 2010.] http://msdn.microsoft.com/en-us/library/dd771983.aspx.

—. 2010e. Lamda Expressions. *MSDN Library.* [Online] 2010e. [Cited: Juli 22, 2010.] http://msdn.microsoft.com/en-us/library/bb397687(v=VS.100).aspx.

—. 2010g. Microsoft.Ccr.Core Namespace. *MSDN Library.* [Online] 2010g. [Cited: Juli 17, 2010.] http://msdn.microsoft.com/en-us/library/bb809434.aspx.

Murphy, Robin R. and Burke, Jennifer L. 2008a. From Remote Tool to Shared Roles. *IEEE Robotics & Automation Magazine, Vol. 15.* New York : Institute of Electrical and Electronics Engineers, 2008a, pp. 39-49.

Murphy, Robin R., et al. 2008b. Search and Rescue Robotics. [ed.] Bruno Siciliano and Khatib Oussama. *Springer Handbook of Robotics.* Berlin : Springer, 2008b, pp. 1151-1173.

National Institute of Standards and Technology. Performance Metrics and Test Arenas for Autonomous Mobile Robots. *Intelligent Systems Division.* [Online] [Cited: August 10, 2010.] http://www.isd.mel.nist.gov/projects/USAR/.

Nielsen, Henrik F. and Chrysanthakopoulos, George. 2007. *Decentralized Software Services Protocol - DSSP/1.0.* Seattle : Microsoft Corporation, 2007.

OASIS;. 2006. OASIS SOA Reference Model TC. [Online] 12. Oktober 2006. [Zitat vom: 22. Juni 2010.] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm.

SourceForge. 2010. WinAVR. *SourceForge.NET.* [Online] Januar 10, 2010. [Cited: Juli 30, 2010.] http://sourceforge.net/projects/winavr/.

Tadokoro, Satoshi, Noda, Itsuki and Nardi, Daniele. 2009. *RoboCup Rescue Home.* [Online] 2009. [Cited: August 10, 2010.] http://www.robocuprescue.org/.

Wikipedia. Schrittmotor. *Wikipedia.* [Online] [Cited: Juli 8, 2010.] http://de.wikipedia.org/wiki/Schrittmotor.

Glossar

A

APM
Asynchronous Programming Model in .NET 36
Arbiter
Coordination Primitve, koordiniert Item und Handler zu einer Task 20

C

Causality
Beziehung zw. Ursache/Wirkung, CCR-Objekt zwecks impliziter Fehlerbehandlung 30
CCR
Concurrency and Coordination Runtime, .NET Library..... 13
CLR
Common Language Runtime, Interpreter des Zwischencodes ähnlich der JVM..... 13

D

Dispatcher
CCR Threadpool 14
DSS
Decentralized Software Services, Architektur Modell 39

I

IC
Integrated Circuit..... 59

M

MCU
MicroController Unit 59

N

Notification
Nachricht, die eine Service-Zustandsänderung signalisiert 50

O

Orchestration
Bezeichnung für die semantische Vereinigung einzelner Services zu einer höheren Funktionalität 40

P

Publisher
Service, der Notifications versendet..... 49

R

RDS
Microsoft Robotics Developer Studio..... 82
RTOS
Real Time Operating System 84

S

Subscriber
Service, der sich bei einem anderen anmeldet, um von ihm Notifications zu erhalten..... 49

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 19.08.2010

Ort, Datum	Unterschrift