Forschungszentrum Karlsruhe
Technik und Umwelt

Wissenschaftliche Berichte
FZKA 6029

MoMo

— eine verteilte, objektorientierte Realzeitarchitektur

Stefan Hepper [1]
Institut für Angewandte Informatik
Forschungszentrum Karlsruhe

Reinhold Oberle [2]
Institut für Angewandte Informatik
Forschungszentrum Karlsruhe

2. Dezember 1997

[1] hepper@iai.fzk.de
[2] oberle@iai.fzk.de

Zusammenfassung

MoMo ist ein verteiltes, objektorientiertes Framework für Realzeitanwendungen, das auf bereits vorhandenen Betriebssystemen aufbaut. MoMo besteht aus den drei Komponenten MoMo-Application-Unit, MoMo-Server und MoMo-Client. Die verteilte Anwendung wird mittels der MoMo-Application-Units realisiert, für die MoMo ein Framework zur Verfügung stellt. Der MoMo-Server sammelt Informationen von den MoMo-Applications-Units, leitet Kommandos von den MoMo-Clients an die Application-Units und überprüft die Zugriffsrechte der Clients. Die Clients realisieren eine Entwicklungsumgebung für den Anwendungsprogrammierer, graphische Konfigurations- und Überwachungsschnittstellen und die graphische Bedienoberfläche für den Endbenutzer. Als Betriebssysteme werden momentan iRMX (Realzeitbetriebssystem) und Linux (Public Domain Unix) und in naher Zukunft auch Windows NT unterstützt. In diesem Bericht liegt der Schwerpunkt auf den MoMo-Application-Units.

Anwendung hat das MoMo-System im ARTEMIS-Projekt gefunden. ARTEMIS ist ein Telepräsenzsystem, das dem Chirurgen ermöglicht, minimal invasive Eingriffe im Bauchraum von einer Arbeitsstation aus durchzuführen. Mittels des MoMo-Frameworks wurden sowohl Master- wie Slave-Einheiten und das Endoskopführungssystem realisiert.

MoMo - a distributed, objectoriented real-time architecture

Abstract

MoMo is a distributed, object oriented framework for real-time applications built on top of existing operating systems. It consists of three components: MoMo Application Units, MoMo Server, and MoMo Clients. Distribution is realized with the MoMo Application Units, for which a framework is available in MoMo. The MoMo Server collects informations from MoMo Application Units, forwards commands from the MoMo Clients to the Application Units, and checks the Clients' access permissions. The Clients offer a development environment for the application programmer, graphical configuration and monitoring interfaces, and the graphical user interface for the end user. At this point iRMX (real-time OS) and Linux (public domain Unix) are supported as operating systems. In the near future Windows NT also will be supported. The focus of this paper is on the MoMo Application Units.

A first application of MoMo is the ARTEMIS project. ARTEMIS features a telepresence system which enables the surgeon to perform minimally invasive surgery in abdominal cavity remotely via a man machine interface. Master units, slave units, and the endoscope guidance system have been implemented within the MoMo framework.

Inhaltsverzeichnis

Kapitel 1

Motivation

Die MoMo-Architektur entstand im Rahmen des ARTEMIS (Advanced Robot and Telemanipulator System for Minimal Invasive Surgery) Projektes. ARTEMIS ist ein Telepräsenzsystem, das dem Chirurgen ermöglicht, minimal invasive Eingriffe im Bauchraum von einer Arbeitsstation aus durchzuführen. Das Telepräsenzsystem besteht aus mehreren Master-Slave-Einheiten, einem Endoskopführungssystem und einer 3D-Realzeit-Simulation. Weitere Informationen zu ARTEMIS findet sich in [Voges *et al.*, 95] und [Holler, 95]. Die aus diesem komplexen Szenario resultierenden Anforderungen, wie

- verteilte Komponenten
- einfach erweiterbar
- portabel
- zuverlässig
- modular
- skalierbar und flexibel

waren mit einer konventionellen monolithischen Realzeitarchitektur nur unter hohem Aufwand zu realisieren. Deshalb wurde ein verteilter Ansatz gewählt — die MoMo-Architektur — der alle oben genannten Anforderungen erfüllt.

So entstand eine betriebssystemunabhängige Plattform für große Softwareprojekte, die eine Auftrennung des Problems in einzelne Objekte, bzw. Aufgaben unterstützt. Sie erlaubt eine Aufteilung in allgemeine Dienstleistungen und einen speziellen Anwendungsteil. Die Dienstleistungsobjekte und ein allgemeiner Task-Frame stehen bei jedem Projekt von vorne herein zu Verfügung und für die Anwendungsobjekte stehen verschiedene Schablonen zur Verfügung, damit eine Anwendung einfach und schnell erstellt werden kann.

Nachfolgend soll diese Architektur im Detail vorgestellt werden und auf die Realisierung der Beispielanwendung ARTEMIS mit der MoMo-Architektur eingegangen werden.

Kapitel 2

MoMo-Architektur

Da komplexe Echtzeitsysteme durch eine hohe Lebensdauer gekennzeichnet sind und zum Teil permanenten Änderungen unterliegen, ist ein Hauptziel des Entwurfs von Softwaresystemen die leichte Änderbarkeit der Systemarchitektur. Dementsprechend stellen gerade nebenläufige, verteilte und echtzeitfähige Softwaresysteme diejenigen Softwaresysteme dar, bei denen die Erstellung einer verständlichen und wartbaren Systemarchitektur im Vordergrund steht. Dies wird auch durch [Tanenbaum, 95] bestätigt, nachdem verteilte Systeme folgende Vorteile haben:

- Besseres Preis-/Leistungsverhältnis als ein großes System
- Zuverlässiger
- Schrittweise erweiterbar

Dem stehen folgende Nachteile gegenüber:

- Abhängig von einem Netzwerk
- Sicherheit wird wichtig (wer darf auf welche Daten zugreifen?)

Da bei größeren Echtzeitsystemen Zuverlässigkeit und Erweiterbarkeit eine sehr wichtige Rolle spielen, überwiegen die Vorteile von verteilten Systemen bei weitem deren Nachteile. MoMo ist per se kein Echtzeitsystem, da es nur auf vorhandenen Betriebssystemen aufbaut. Es erlaubt aber, den Teil der Anwendung, der Realzeitfähigkeiten braucht (der im Vergleich zum Gesamtsystem meist nur 10-20% ausmacht), zu kapseln und diesen auf einen Rechner mit Realzeitbetriebssystem (oder einen Mikrocontroller) auszulagern. Dies führt zu modularen und somit überschaubareren Systemen.

Objektorientierte[1] Softwaresysteme bieten durch das modul-artige Konzept sehr gute Voraussetzungen für die einfache Wiederverwendung von Software (Software-Reuse). Deshalb wurde für MoMo eine verteilte und objektorientierte Architektur gewählt.

[1] Hier wird von einer breiteren Sicht der Objektorientierung als nur die Konzepte von Objekt, Klassen und Vererbung ausgegangen. Es wurde die Definition aus [InfoDuden, 88] zugrunde gelegt: Objektorientiert heißt eine Programmierumgebung, wenn sie Systeme zu spezifizieren oder zu programmieren gestattet, die aus dynamischen Objekten zusammengesetzt sind, welche über den Austausch von Nachrichten zusammenwirken .

In diesem Kapitel wird die von einer konkreten Anwendung unabhängige MoMo-Architektur vorgestellt. Zuerst wird das ganze MoMo-System vorgestellt und dann auf einzelnen Komponenten genauer eingegangen.

2.1 Übersicht

Die MoMo-Architektur ist ein verteiltes System bestehend aus:

- den MoMo-Application-Units (MoMo-AUs), auf denen die Realzeit-Anwendung läuft
- diversen Clients zur Interaktion mit einem/mehreren Benutzer/Benutzern
- einem zentralen MoMo-Server als Bindeglied zwischen den MoMo-Application-Units, den Benutzeroberflächen (Clients) und den externen Datenquellen (z.B. Datenbanken)

In Abbildung 2.1 ist diese Architektur schematisch skizziert. Aufgabe des MoMo-Servers ist es, Informationen von den MoMo-Application-Units zu sammeln (z.B. Daten von der angesteuerten Hardware, interne Statistiken), diese in einer Datenbank zu verwalten und den entsprechenden Clients zur Verfügung zu stellen. Die Kommunikation zwischen Client und Server wird über das HTTP-Protokoll[2] realisiert. So ist es möglich, sich mit jedem Java-fähigen WWW-Browser (z.B. Communicator 4.0 von Netscape) mit dem MoMo-Server zu verbinden, von dort den entsprechenden Client als Java-Programm[3] zu laden und dann lokal auszuführen. Die Befehle der Clients leitet der MoMo-Server zu den entsprechenden MoMo-Application-Units weiter und die Daten der MoMo-Application-Units werden an die Clients gegeben. Mit den MoMo-Application-Units kommuniziert der Server über eine standardisierte Schnittstelle (genannt MoMo-Handler, oder MoMoH), der eine ähnliche Funktionalität wie Remote Procedure Calls[4] aufweist. Der MoMo-Server stellt eine Schnittstelle zwischen dem HTTP-Protokoll und dem MoMo-Protokoll dar und nimmt auch administrative Aufgaben wahr. Er enthält eine Datenbank mit Informationen über die Peripheriegeräte und kann auch auf externe Datenbanken zugreifen. Er prüft die Zugriffsrechte der Clients, protokolliert alle relevanten Aktionen der Clients und erstellt Statistiken wichtiger Systemparameter der MoMo-Application-Units.

Die Clients in Abbildung 2.1 stellen die graphische Schnittstelle zwischen den MoMo-Application-Units und dem oder den Benutzern dar. Beispiele

[2] HyperText Transfer Protocol: Standardtransferprotokoll im World Wide Web. Jede Interaktion besteht aus einer ASCII-Anfrage, gefolgt von einer MIME (Multipurpose Internet Mail Extension) ähnlichen Antwort.

[3] Interpretierte Programme, mit verfügbarem Interpreter für fast alle gängigen Architekturen (diverse Unix, Windows NT, Windows 95, MacOS, ...). Die Java Programmiersprache ist objektorientiert und hat spezielle Sicherheitsmechnismen für Internet/Intranet-Anwendungen und wurde von SUN entwickelt [Arnold & Gosling, 96, Javateam, 97].

[4] Remote Procedure Calls (RPCs) erlauben den Aufruf von Prozeduren auf anderen Rechnern, so als wären es lokale Prozeduraufrufe. Ein sogenannter stub erledigt hierbei die Anpassung von verschiedenen Rechnerarchitekturen untereinander. Ein Beispiel für einen RPC ist unter Unix das Kommando rsh, das eine Shell auf einem anderen Rechner öffnet. Im MoMo-Konzept entspricht der MoMo-Handler dem stub, er hat allerdings eine größere Funktionalität (z.B. überwachen des eigenen Rechners und der Verbindung zu dem Remote-Rechner).

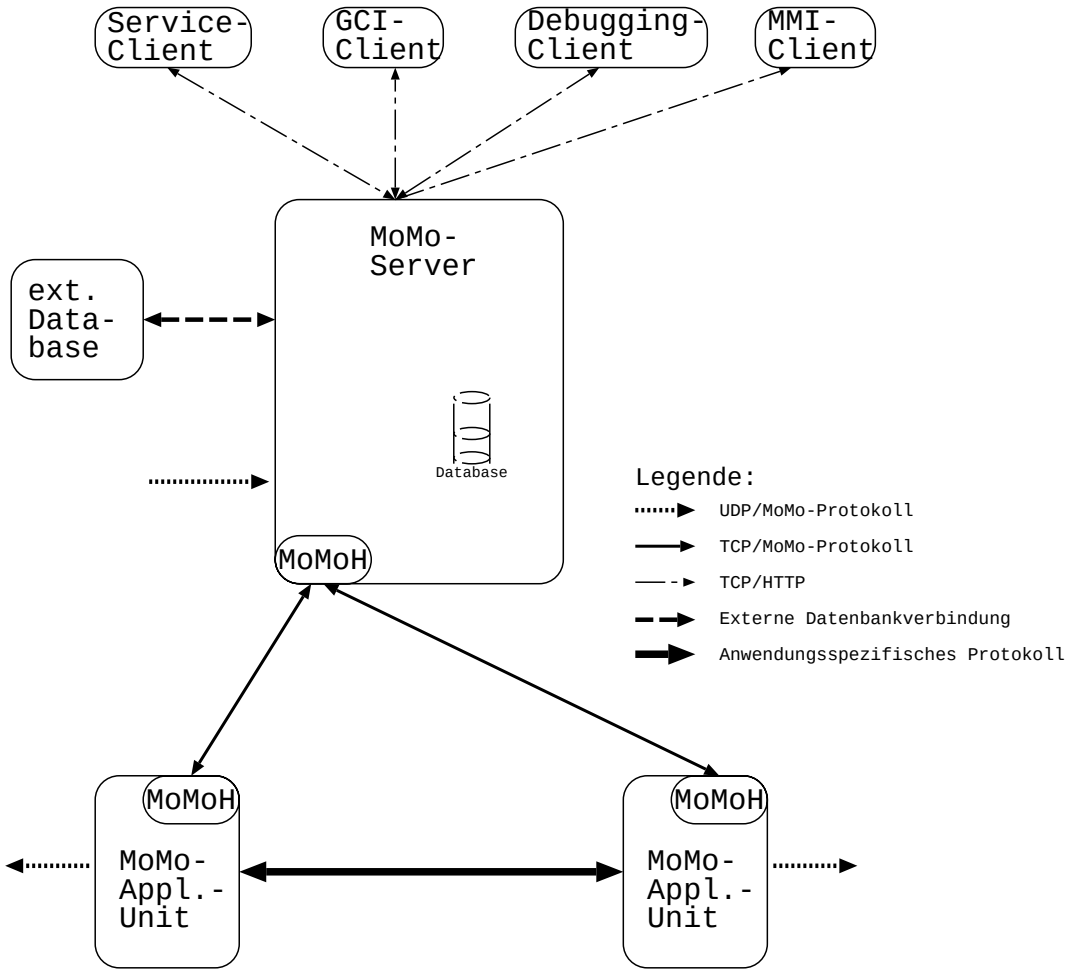


Abbildung 2.1: Globale MoMo-Architektur

hierfür sind Clients zur Konfigurierung (GCI[5]), Überwachung und Debugging von MoMo-Systemen oder anwendungsspezifische graphische Benutzungsoberflächen (Man-Machine-Interfaces, MMI's).

Die MoMo-Application-Units realisieren die spezielle Anwendung. Die Anwendung kann hierbei in mehrere MoMo-AUs zerlegt werden. Die MoMo-AUs kommunizieren mit dem MoMo-Server über den MoMo-Handler und falls nötig auch untereinander (z.B. über einen Feldbus oder Ethernet).

Das Protokoll zwischen zwei MoMo-Handlern basiert auf TCP[6], was den Vorteil einer sicheren Übertragung bietet. Der Nachteil ist, daß es, wie auch Remote Procedure Calls, nur eine Point-to-Point Verbindung zuläßt. Es ist somit nicht möglich, Informationen an eine ganze Gruppe zu schicken. Um die-

[5]Graphical Configuration Interface: Graphischer Editor zur Konfiguration eines MoMo-Systems

[6]Transmission Control Protocol: zuverlässiges verbindungsorientiertes Transportmodell zur Übertragung von IP (Internet Protocol) Datagrammen. Weitere Informationen zu Netzwerken und Transportmodellen findet sich in [Tanenbaum, 97].

sen Nachteil zu beseitigen, gibt es im MoMo-Konzept zusätzlich eine, allerdings unsichere, One-to-Many Kommunikation über UDP[7]. So können MoMo-AUs, MoMo-Server und andere Tools (z.B. graphische Simulation) periodisch Daten an alle interessierten Rechner schicken (z.B. Roboterpositionsdaten).

Die Vorteile dieses verteilten Konzepts für Realzeitanwendungen sind:

- Alle beteiligten Realzeitrechner (MoMo-AUs) lassen sich von einem beliebigen Client-Rechner überwachen und managen. Durch die Benutzung von Java-Applets besteht eine große Freiheit in der Wahl des Client-Betriebssystems.

- Einfach erweiterbar:
  Es können einfach neue Realzeitkomponenten hinzugefügt werden, da jede MoMo-AU über eine Standardschnittstelle verfügt, über die sie sich bei dem MoMo-Server anmelden kann.

- Höhere Sicherheit und größere Zuverlässigkeit:
  Durch die Kapselung von den verschiedenen Realzeitaufgaben in einzelne MoMo-AUs entsteht eine erhöhte Sicherheit, da Fehlfunktionen nur auf eine MoMo-AU begrenzt bleibt. Zudem ist es mit diesem Konzept einfach, sicherheitsrelevante Teile redundant auszulegen.

Als Nachteile ergeben sich:

- Hoher Kommunikationsaufwand:
  Durch die vielen verteilten Rechner entsteht ein höherer Kommunikationsaufwand als bei einem monolithischen System. Dies bedeutet, daß das Netzwerk ein Flaschenhals werden kann. Bei wichtigen Steuerungsclients muß man deshalb darauf achten, daß sie in einem von anderen Rechnern abgekoppelten Ethernet betrieben werden. Somit können Rechner, die nichts mit der Realzeitanwendung zu tun haben, nicht das Netzwerk belasten. Bei reinen Überwachungsclients oder bei Ferndiagnoseclients ist auch ein Einsatz über das Internet denkbar, da hier der Zeitfaktor nicht kritisch ist.

- Netzwerkabhängig:
  Durch die Verteilung ist das ganze System abhängig von dem Netzwerk. Wenn das Netzwerk zusammenbricht oder unterbrochen wird, bekommen die MoMo-AUs keine Kommandos mehr von dem MoMo-Server. Dies muß bei der Realisierung der MoMo-AUs berücksichtigt werden, so daß reale Geräte nie durch einen Netzwerkausfall in einen kritischen Zustand geraten können. Hierfür ist das anwendungsspezifische Protokoll gedacht, das eine sichere und echtzeitfähige Kommunikation zwischen den realzeitfähigen MoMo-AUs ermöglicht.

Nach [Tanenbaum, 95] ist Transparenz[8] für den Entwurf eines verteilten Betriebssystems eines der wichtigsten Kriterien. Da sich MoMo aus der Sicht des Applikationsprogrammierers wie ein Betriebssystem darstellt, wurden in der Tabelle 2.1 alle Transparenzarten nach [Tanenbaum, 95] für MoMo bewertet.

[7] User Data Protocol: unzuverlässiges verbindungsloses Transportmodell zur Übertragung von IP (Internet Protocol) Datagrammen.

[8] Transparent ist ein verteiltes System dann, wenn es sich von Außen wie ein nichtverteiltes System darstellt. Bei Softwaresystemen kann Transparenz auf zwei Ebenen stattfinden: auf der Benutzer- und der Programmebene [Tanenbaum, 95].

Transparenzart	MoMo	Bedeutung
Location	Ja	Ressourcenstandort (HW u. SW) verborgen
Migration	Ja	Ressourcen können ihren Standort ändern
Replication	Ja	Es können verborgene Kopien existieren
Concurrency	Ja	Gleichzeitige Nutzung von Ressourcen
Parallelism	Nein	Interne Parallelisierung von Aktivitäten

Tabelle 2.1: Transparenzarten nach [Tanenbaum, 95] in verteilten Systemen und der Bewertung in Bezug auf MoMo

Location Transparency wird in MoMo durch Verteilung einer MoMo-Application-Unit (Kap. 2.2.13) und der OS-Abstraction-Layer (Kap. 2.2.2) ersichtlich. Sie basiert auf dem MoMo-Handler (Kap. 2.2.7), der Dienste ähnlich Remote Procedure Calls zur Verfügung stellt, und den standardisierten Interprozeßkommunikations-Diensten.

Migration Transparency erlaubt den Wechsel einer Ressource zu einem anderen Standort (Rechner), ohne daß sich der Name der Ressource ändert. Auch dies ist unter MoMo möglich, da der MoMo-Katalog die Ressourcen verwaltet und somit die Verteilung der Ressourcen verbirgt (siehe dazu auch Kap. 2.2.13).

Replication Transparency ermöglicht es dem Software System Kopien von Ressourcen anzufertigen, ohne daß dies von der Anwendung bemerkt wird. In MoMo wird dies für Realzeit-Ressourcen nicht ermöglicht, wohl aber zur Datensammlung auf dem MoMo-Server von den MoMo-AUs (per UDP).

Concurrency Transparency ermöglicht die gemeinsame Nutzung von Ressourcen. Eventuell auftretende Konflikte werden vor dem Programmierer versteckt. Bei MoMo werden alle Ressourcen per Lock-Mechanismus vor gleichzeitiger Nutzung geschützt und haben somit immer sequentiellen Zugriff, da i.A. eine gleichzeitige Nutzung (z.B. Roboter-Hardware) in Realzeitsystemen nicht sinnvoll ist.

Parallelism Transparency wird von MoMo nur eingeschränkt unterstützt. Zwar kann eine MoMo-AU auf mehrere Rechner/CPUs verteilt werden, allerdings können immer nur ganze Tasks ausgelagert werden (siehe Kap. 2.2.13). Eine weitere Parallelisierung ist dann von dem zugrunde liegenden Betriebssystem abhängig (z.B. Microkernel Structure). Da MoMo aber bewußt kein eigenes Betriebssystem sein sollte, sondern nur ein Framework zur Erstellung von verteilten Realzeitanwendungen mit objektorientierten Mitteln, mußten sowohl bei der Concurrency Transparency wie auch bei der Parallelism Transparency Einschränkungen gemacht werden.

MoMo unterstützt auch die in [Blair & Lea, 92] geforderte Flexibilität des Transparenzmodels, so daß verschiedene Prinzipien (Remote Procedure Calls, Proxies, Distributed Virtual Memory) je nach Kontext wählbar sind. MoMo realisiert dies durch die standardisierten Schnittstellen, die den darunterliegenden Service verbergen. Dieser muß erst zur Laufzeit konfiguriert werden (siehe Kap. 2.2.2 und 2.2.11).

Der Applikationsentwickler wird bei MoMo auf mehreren Ebenen unterstützt:

1. Implementierung
   Unterstützung durch einen hohen Grad an SW-Reuse durch Task-

Frame/Task-Pattern und einer umfangreichen Bibliothek für Realzeit-Anwendungen.

2. Inbetriebnahme
   Durch Funktionsdebugging, das auch Remote möglich ist, und Ereignis- und Fehlerprotokollierung.

3. Protokollierung/Archivierung
   Durch Ereignisprotokollierung und Speichern ausgewählter Daten in einer Datenbank auf dem Server.

Die Implementierungsebene wird in Kapitel 2.2.15, die Punkte 2 und 3 in Kapitel 2.2.14 näher erläutert.

2.2 MoMo-Application-Unit

Das Verhalten eines Realzeit-Systems läßt sich als Stimulus/Response-System beschreiben. Die Stimuli lassen sich nach [Sommerville, 95] in zwei Kategorien unterteilen:

1. **Periodische Stimuli**. Diese Stimuli treten periodisch zu vorhersagbaren Zeiten auf.

2. **Aperiodische Stimuli**. Diese Stimuli sind nicht vorhersehbar, sondern von Ereignissen abhängig (event-driven).

In MoMo sind für beide Stimuliarten Mechanismen vorgesehen: Jede Application Task hat eine einstellbare Frequenz, mit der sie ihre Stimuli abarbeitet. Den aperiodischen Stimuli wird man am besten dadurch gerecht, daß man für jeden Stimulus eine eigene Task zur Bearbeitung bereit stellt. Zudem besitzt jede Task einen Zustandsautomaten, um zeitliche Abfolgen von Stimuli zu berücksichtigen.

Die MoMo-Application-Units (MoMo-AUs) realisieren immer abgeschlossene Aufgaben. Sie sind die gröbste Aufteilungsmöglichkeit innerhalb von MoMo. Die Aufgabe einer MoMo-AU wird wiederum in verschiedene Teilaufgaben unterteilt werden, für die jeweils eine eigene Task existiert statt einer einzigen Task mit einer großen „while“-Schleife. Dies bringt folgende Vorteile [Gallmeister, 95]:

- **Einfachheit**. Jede Task ist eine abgeschlossene Einheit, so wird die Gesamtstruktur besser überschaubar. Zudem kann jede Task mit der ihrer Tätigkeit angepaßten Frequenz arbeiten.

- **Skalierbarkeit**. Jede Task kann einfach auf einen anderen Prozessor oder anderen Rechner verlagert werden.

- **Modularität**. Zusätzliche MoMo-Objekte können einfach hinzugefügt oder gelöscht werden (z.B. MoMo-Logger).

- **Speicherschutz**. Viele Betriebssysteme (z.B. Unix, Windows NT) unterstützen das Konzept des virtuellen Adreßraumes. Jede Task hat einen eigenen Adreßraum und kann im Fehlerfall keine andere Task durch Speicherschutzverletzungen beeinträchtigen.

Als Nachteile dieses Konzepts ergeben sich:

- **Adminstrations-Overhead**. Das Betriebssystem muß alle Tasks verwalten und mit einem Schedulingalgorithmus ihre Ausführungsreihenfolge bestimmen. Dies benötigt sowohl Speicher- wie Rechenkapazität.
- **Langsamer**. Jeder Taskwechsel kostet Rechenzeit, somit sind mehrere Prozesse, auf Systemen mit nur einem Prozessor, langsamer als ein einzelner Prozeß.
- **Isoliert**. Tasks können globale Variable nur über Shared Memory miteinander teilen, was aufwendiger ist als eine globale Variable innerhalb einer Task. Um diesen Aufwand für den Anwendungsprogrammierer so gering wie möglich zu halten, wurde in MoMo der Katalog eingeführt. Über ihn können die Tasks beliebige Speicherbereiche gemeinsam nutzen.

2.2.1 Übersicht

Die MoMo-Application-Units (MoMo-AUs) realisieren die Funktionalität der Anwendung. In Abbildung 2.2 ist eine solche MoMo-AU zu sehen. Eine MoMo-AU besteht immer aus einer MoMo-Control-Unit und pro Rechner einem MoMo-Handler und einem MoMo-Katalog. Eine MoMo-AU kann auf mehrere Rechner verteilt werden (siehe Kapitel 2.2.13), die aber von einer Control Unit gesteuert werden (z.B. einfache Auslagerung rechenintensiver Tasks). Ein MoMo-System kann wiederum aus mehreren MoMo-AUs bestehen. Somit ist eine gute Modularisierung und Skalierung möglich.

Wie in der Abbildung zu sehen ist, sind die Anwendungsprogramme vollständig gegenüber dem darunter liegenden Betriebssystem gekapselt. Somit sind alle Anwendungsprogramme durch einfaches Übersetzen auf dem jeweiligen Zielbetriebssystem (Target-OS) ohne Änderungen lauffähig. Momentan unterstützt MoMo als Target-OS iRMX[9], Linux[10]und in naher Zukunft auch Windows NT[11] und als Programmiersprache C.

In den nachfolgenden Kapiteln wird zunächst die MoMo-AU aus der funktionalen (Kap. 2.2.2 bis 2.2.8) und der objektorientierten Sichtweise (Kap. 2.2.9 und 2.2.10), dann aus der Organisationssicht (Kap. 2.2.11 und 2.2.12) betrachtet. Anschließend wird noch auf die Verteilung einer MoMo-AU (Kap. 2.2.13) und die Unterstützung für den Anwendungsprogrammierer (Kap. 2.2.14 und 2.2.15) eingegangen.

2.2.2 OS-Abstraction Layer

Um sowohl die MoMo-Software als auch die Anwendungssoftware von dem darunter liegenden Betriebssystem abzukoppeln, wurde die Operating System Abstraction Layer (OS-Abstraction Layer) eingeführt. Die Abstraktion der Betriebssystemdienste und der Hardware hat mehrere Vorteile: Anwendungen lassen sich betriebssystemunabhängig erstellen und können durch einfaches Neuübersetzen auf ein anderes (von MoMo unterstütztes) Betriebssystem

[9] Realzeitbetriebssystem, ehemals von Intel, nun von Radisys unterstützt. Läuft unter Intel x86 und Multibus-Hardware

[10] Public Domain Unix, weitgehend POSIX-konform, verfügbar für Intel x86, Motorola 68k, Digital Alpha, SPARC, Mips und Motorola PowerPC.

[11] PC-Betriebssystem für Intel x86, Digital Alpha, Mips und PowerPC von Microsoft.

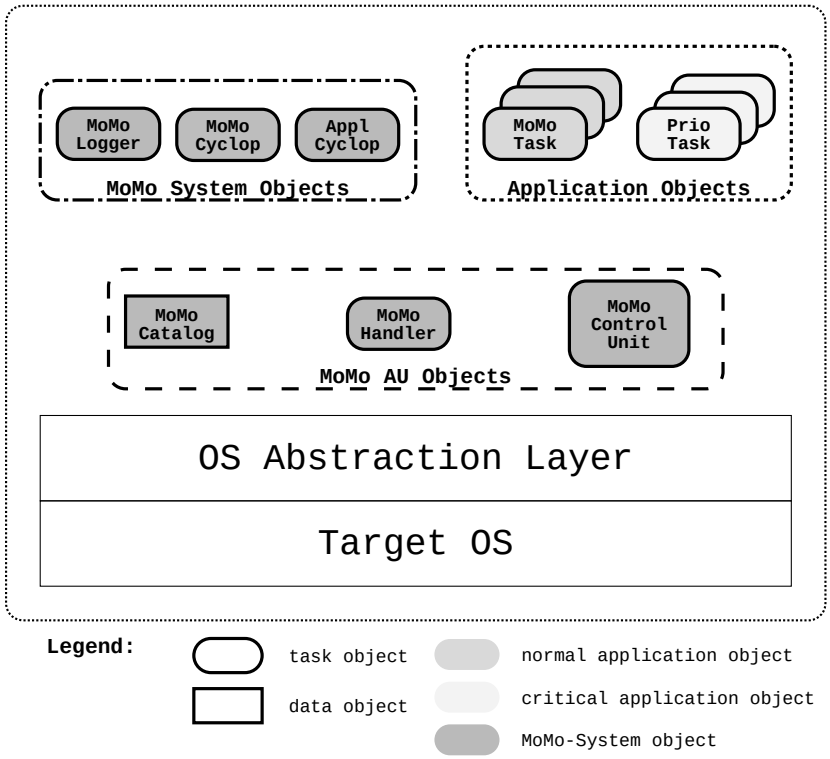


Abbildung 2.2: Aufbau einer MoMo-Application-Unit

übertragen werden. Des weiteren läßt sich das MoMo-System leicht auf neue Betriebssysteme portieren, da nur die OS-Abstraction Layer angepaßt werden muß[12]. Diese Zwischenschicht ermöglicht auch ein betriebssystemunabhängiges Debuggen der Anwendung und eine Überwachung/Aufzeichnung der Betriebssystemaufrufe. Nachteile sind eine geringere Performanz wegen einer größeren Zahl von Prozeduraufrufen und die nur eingeschränkte Ausnutzung aller Betriebssystemfähigkeiten.

Folgende Dienste stellt die OS-Abstraction Layer zur Verfügung:

- **IPC (InterProcess Communication)**. Mit den IPC-Diensten lassen sich einfach Nachrichten/Daten zwischen verschiedenen Tasks austauschen. Die wichtigsten Dienste sind:
  - Lookup (Handle, Name)
    Nachschauen im MoMo-Katalog, ob dort eine Schnittstelle mit dem Namen *Name* registriert ist. Wenn ja, dann wird das Token *Handle* zurückgegeben, das bei allen zukünftigen Operationen auf diese Schnittstelle angegeben werden muß.
  - Send (Handle, Data)
    Sendet an die Schnittstelle mit dem Token *Handle* die Daten *Data*.

[12] Ein neues Target OS sollte FIFO-Scheduling (für Echtzeit-Verhalten nötig) und möglichst POSIX.1 und POSIX.4 unterstützen. Dann sollte eine Portierung von MoMo in weniger als einem Monat realisierbar sein.

- Receive (Handle, Data, MaxLen)
  Empfängt von der Schnittstelle mit dem Token *Handle* maximal *MaxLen* Daten und speichert sie in *Data*.

Diese Dienste sind von der darunter liegenden Realisierung unabhängig und der Task nicht bekannt (Transparenz auf Programmebene). Dies ermöglicht unter anderem eine einfache Verteilung von Tasks auf verschiedene Rechner. Ein weiterer Vorteil dieser Implementierungstransparenz ist die optimale Ausnutzung des jeweiligen Target-Betriebssystems, z.B. verfügen Realzeit-Betriebssysteme wie iRMX über verschiedene, effiziente Interprozeßkommunikationsmöglichkeiten. Als Realisierungen stehen unter Linux und iRMX zur Auswahl:

- Shared Memory (Linux, iRMX)
  Hier erfolgt der Datenaustausch über einen gemeinsamen, zusammenhängenden Speicherbereich.
- TCP/IP (Linux, iRMX)
  Verbindungsorientiertes, zuverlässiges Protokoll zum Austausch von Daten zwischen Rechnern.
- UDP/IP (Linux, iRMX)
  Verbindungsloses, unzuverlässiges Protokoll zum Austausch von Daten zwischen Rechnern/Rechnergruppen.
- Mailbox (iRMX)
  Effizienter Mechanismus für Message-Parsing innerhalb eines Rechners, basierend auf einer Queue mit Messages. Weiteres hierzu und zu anderen Interprozeßkommunikationsmöglichkeiten unter iRMX findet sich in [Vickery, 1993].

- **HAL (Hardware Abstraction Layer)**. Die HAL bietet eine standardisierte Schnittstelle zu dem technischen Prozeß. Sie erlaubt die Entwicklung von Anwendungssoftware ohne Kenntnis des späteren Prozeßinterface sowie eine dynamisch anpassungsfähige Verkabelung. Die Zuordnung der einzelnen Prozeßsignale zu der in der Software implementierten I/O-Funktion ist konfigurierbar und somit ohne Softwareänderung an unterschiedliche Interfacekarten anpaßbar. Dies führt zu einer strikten Codetrennung zwischen Anwendungssoftware und Prozeßinterfacesoftware (Karten-Treiber).
  Die HAL bietet, abhängig von der I/O-Art, folgende Dienste:

  - Analog I/O
  - Digital I/O
  - Timerfunktionen

- **Infrastrukturdienste**. Hierunter wurden alle sonstigen Dienste zusammengefaßt. Dies sind momentan:

  - Taskadministration
    (z.B. Starten, Beenden, Ändern der Priorität, TaskID, Task-CPU-Zeit, ...)

- File-I/O
  Standardroutinen zum Einlesen von lokalen Parameterdateien.
- CCH (Communication Channel Handler)
  Der CCH ist ein separates Programm, das über Schnittstellen mit anderen Programmen kommuniziert. Er stellt eine fehlerfreie, gepufferte und zeitüberwachte TCP-Verbindung zur Verfügung. Dies ist besonders für zeitkritische Kommunikation (z.B. Master-Slave-Steuerung) in Echtzeitsystemen wichtig. Alternativ zur TCP-Verbindung läßt sich auch eine gepufferte UDP-Verbindung realisieren.
- MoMo-Katalog
  Shared Memory Bereich zur Verwaltung der Schnittstellenobjekte (siehe Kap. 2.2.5).
- MoMo-Pool
  Pool von Shared Memory Segmenten zum Datenaustausch zwischen verschiedenen Tasks (siehe auch Kap. 2.2.6).
- Debugging
  Debug-Utilities und Ausgabefunktionen.
- Sonstiges
  (z.B. Mathematik-Library, Bit-Library)

2.2.3 Task-Automat

Alle Tasks auf einer MoMo-AU basieren auf einem Task-Automaten[13], der in Abbildung 2.3 dargestellt ist. Der Task-Automat ist ein endlicher Zustandsautomat (Finite State Machine, FSM) mit folgenden Zuständen:

1. **Init-State**. Im Initialisierungszustand erfolgt die externe Parametrisierung der Task (z.B. über Einlesen einer Init-Datei) und das Holen der IDs aller benötigten Objekte (mit Ss_Lookup, siehe Kap. 2.2.5).

2. **Basic-State**. Grundzustand, in dem alle Variablen und eventuelle Hardware initialisiert werden.

3. **Ready-State**. Im Bereit-Zustand ist alles initialisiert und korrekt. Die Task wartet nur auf das Aktiv-Kommando, um ihre Anwendungsfunktion auszuführen.

4. **Active-State**. Im Aktiv-Zustand erledigt die Task ihre eigentliche Aufgabe. Z.B. ist jetzt eventuell angeschlossene Hardware aktiv.

5. **End-State**. Beenden der Task.

Der Übergang von *Init-State* zu *Basic-State* nimmt eine Sonderstellung ein, da er nicht von außen beeinflußbar ist. Der Übergang wird von der MoMo-Control-Unit (MoMo-CU) mit *Com_Basic* initiiert, und der Task wird eine Task-ID zugewiesen. Mit dieser ID meldet sich die Task künftig bei der MoMo-CU.

[13] Natürlich können auch Programme auf dem Rechner gestartet werden, die nicht auf dem MoMo-Task-Automaten basieren. Diese sind aber dann nicht in das MoMo-Konzept eingebettet, d.h. Remote-Debugging, Überwachung durch die Control Unit und andere MoMo-Features sind nicht möglich.

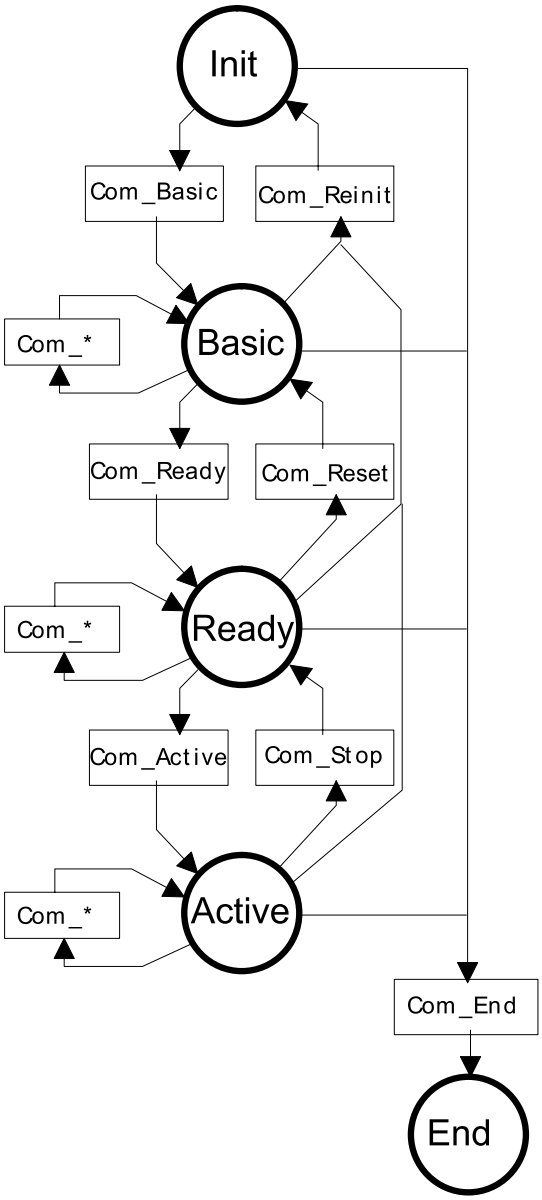


Abbildung 2.3: MoMo-Task-Automat
Zustandsübergänge werden von der MoMo-Control-Unit mittels Kommandos *Com_* initiiert, während alle Kommandos, die keinen Zustandsübergang erzeugen, unter *Com_** zusammengefaßt wurden.

Im *Basic-State* gibt es vier Möglichkeiten: Bei dem Kommando *Com_Reinit* von der MoMo-CU geht die Task in den *Init-State*, bei *Com_End* in den *End-State* und bei *Com_Ready* in den *Ready-State* über. Bei allen anderen Kommandos bleibt die Task in ihrem Zustand. Analoges gilt für *Ready-* und *Active-State*.

Realisiert ist dieser Task-Automat im TaskFrame (siehe Kap.2.2.15), und die Steuerung der Zustandsübergänge wird von der MoMo-CU, die in jeder MoMo-AU vorhanden ist, erledigt. Im Normalfall wird eine Task durch die MoMo-CU in den gleichen Zustand wie die MoMo-CU selbst gesetzt (Zustands-Transparenz auf MoMo-AU-Ebene), aber es kann auch explizit per MoMo-Handler der Zustand einer bestimmten Task von außen gesetzt werden (z.B. für Debugging-Zwecke).

2.2.4 MoMo-Objekte

Die Objekte auf einer MoMo-AU können in folgende Kategorien unterteilt werden:

- **MoMo-AU Objects**. Hierzu zählen alle MoMo-Objekte, die Voraussetzung für eine MoMo-AU sind, also MoMo-Catalog, MoMo-Control-Unit und der MoMo-Handler
- **MoMo-System Objects**. Alle jeder Anwendung von vorne herein zur Verfügung stehenden Dienste. Dies sind:
  - MoMo-Logger
    Speichern von Fehlern und sonstigen Ausgaben in einer Datei (Logfile, siehe Anhang F). Dient zum ereignisorientierten Verarbeiten von Daten.
  - MoMo-Cyclop
    Versenden von allgemeinen Systeminformationen per UDP. Dies passiert zyklisch und ermöglicht ein periodisches Versenden von Daten.
  - Appl-Cyclop
    Versenden von anwendungsspezifischen Informationen per UDP. Dient ebenso wie der MoMo-Cyclop zum periodischen Versenden von Daten.
- **Application Objects.** Diese Taskobjekte sind applikationsspezifisch und basieren auf den MoMo-Task-Objekten (siehe Kap. 2.2.10). Grundsätzlich unterscheidet man zwischen zwei Typen:
  - Prio Tasks
    Die Prio Tasks sind anwendungsspezifische Tasks mit sicherheitsrelevanten Aufgaben. Die MoMo-Control-Unit überwacht zyklisch alle Prio Tasks und wenn sie einen Fehler entdeckt, oder eine Prio Task nicht im Aktiv-Zustand ist, wird die ganze Anwendung in einen sicheren Zustand (Grundzustand) gebracht.
  - MoMo Tasks
    MoMo Tasks sind anwendungsspezifische Tasks, der im Gegensatz zu Prio Tasks keine sicherheitskritischen Tasks sind. Auch sie werden von der MoMo-CU zyklisch überwacht, aber da ihr Ausfall oder gezieltes Herunterfahren nur zu einer partiellen und tolerierbaren Funktionseinschränkung der gesamten MoMo-AU führt, werden die anderen Tasks auf der MoMo-AU dadurch nicht beeinflußt. So können z.B. während des Betriebs neue Versionen der MoMo Tasks eingespielt werden.

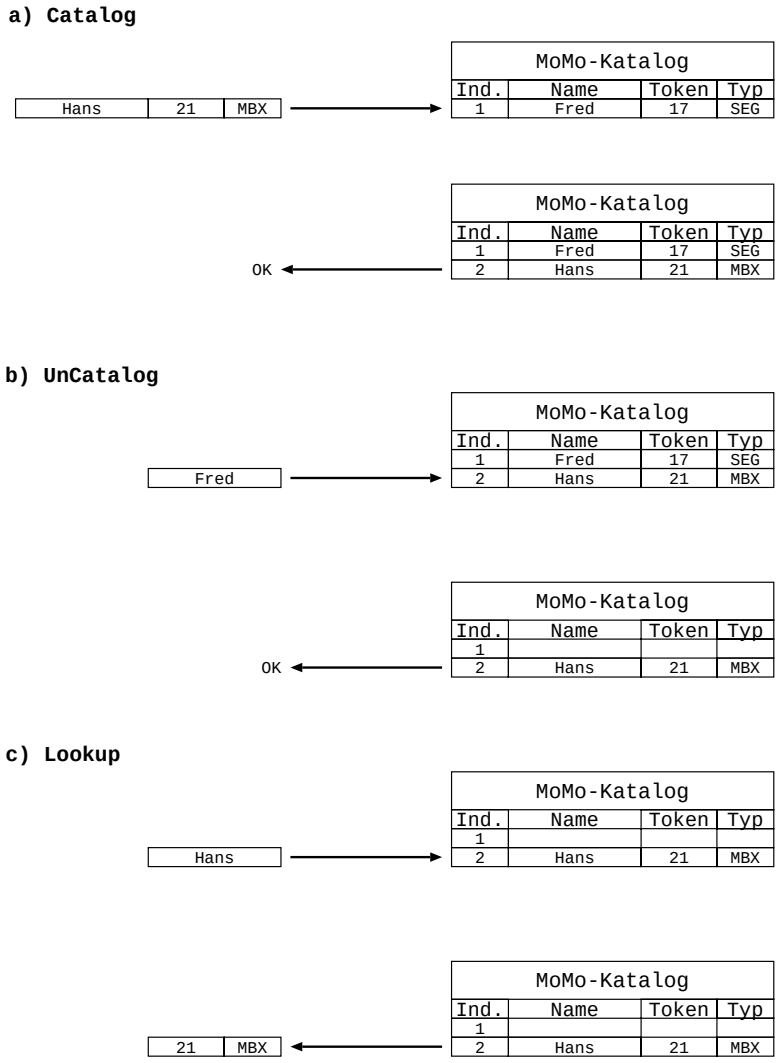


Abbildung 2.4: Mögliche Operation auf dem MoMo-Katalog

2.2.5 MoMo-Katalog

Der MoMo-Katalog basiert auf einem Shared Memory Bereich, in dem alle Schnittstellenobjekte (darunter fallen auch Task-Objekte, siehe Kap. 2.2.10) verwaltet werden. Dies ermöglicht ein von den Tasks unabhängiges Anlegen, Löschen und Umkonfigurieren der Schnittstellenobjekte (z.B. mit dem MoMo-Handler, siehe Kap. 2.2.7). Der Katalog entspricht einer primitiven Datenbank, auf die alle lokalen Tasks direkt und alle Remote Tasks über den MoMo-Handler zugreifen können. Natürlich läßt sich der Katalog auch für Datenbankzwecke nutzen.

In Abbildung 2.4 sind die möglichen Operationen auf dem MoMo-Katalog zu sehen:

- Catalog

Hiermit wird der Name, das Token und der Typ in den Katalog eingetragen. In dem abgebildeten Beispiel wird die Mailbox (Typ MBX), die die Shared Memory ID 21 hat (Token), unter dem Namen Hans katalogisiert.

- UnCatalog
  Löscht einen Eintrag aus dem Katalog.

- Lookup
  Liefert für einen Namen den Token und den Typ. In dem Beispiel könnte eine zweite Task ein Lookup auf Hans ausführen und bekommt die Information, daß es sich um eine Mailbox (Typ MBX) handelt, die die Shared Memory ID 21 hat.

Im MoMo-System wird der Katalog von den Schnittstellenobjekten benutzt; genaueres wird im Kapitel Schnittstellen-Objekte (2.2.9) erläutert.

2.2.6 MoMo-Pool

Der MoMo-Pool realisiert eine Shared Memory Verwaltung, die betriebssystemunabhängig ist. MoMo benötigt sehr viele Shared Memory Segmente, da jedes Objekt mindestens ein Schnittstellensegment (ICS) besitzt. Da bei den meisten Betriebssystemen die Anzahl der Shared Memory Segmente sehr gering ist, wird nur ein großes Segment belegt und dann von dem MoMo-Pool verwaltet.

Der MoMo-Pool stellt zwei unterschiedliche Funktionsschichten zur Verfügung:

1. **Pool-Verwaltung**. Unter die Poolverwaltung fallen:

   - Erzeugen des Pools
     Es können mehrere Standardgrößen für die Segmente vorgegeben werden und eine maximale Anzahl für die jeweilige Größe.
   - Löschen des Pools
   - Informationen über Poolnutzung
     Hierüber erlangt man unter anderem Auskunft über die Anzahl der belegten Segmente für jede Größe und die Task-IDs der auf diese Segmente zugreifenden Tasks.
   - Konsistenzcheck Pool
     Jedes Segment besitzt ein spezielles Muster, um es von dem nachfolgenden Segment zu trennen. Über den Konsistenzcheck kann überprüft werden, ob über die Grenze eines Segmentes hinausgeschrieben wurde und Daten in anderen Segmenten beschädigt wurden.

2. **Pool-Dienste**. Die Pool-Dienste stehen den MoMo-Tasks zur Verfügung. Statt betriebssystemspezifische Shared Memory Calls nutzen sie die Pool-Dienste. Im einzelnen sind dies:

   - Anlegen eines Segments
     Anfordern einer Segment-ID für ein Shared Memory Segment der Größe n.

- Löschen eines Segments
  Freigeben des Segments mit Segment-ID.
- Adresse eines Segments
  Pointer auf das Segment abgebildet auf den Adressbereich der MoMo-Task.
- Informationen über ein Segment
  Dient zum Erlangen segmentspezifischer Daten wie Segmentgröße, benutzte Größe und ID der Task, die dieses Segment angefordert hat.
- Konsistenzcheck Segment
  Prüft, ob dieses Segment beschädigt wurde.

Somit existiert ein sicherer und betriebssystemunabhängiger und effizienter Mechanismus zum Austausch von Daten zwischen zwei Tasks.

2.2.7 MoMo-Handler

Funktion des MoMo-Handlers ist es, Dienste für Remote-Rechner zur Verfügung zu stellen (analog zu Remote Procedure Calls).

Wie in Abbildung 2.5 zu sehen ist, kann jeder MoMo-Handler nur von genau einem anderen MoMo-Handler (z.B. dem des MoMo-Servers) Kommandos empfangen. Dies dient zur Vermeidung von Kommando-Konflikten. Ein MoMo-Handler kann aber zu mehreren Remote MoMo-Handlern Kommandos schicken (z.B. schickt der MoMo-Server an jede MoMo-AU seine Kommandos).

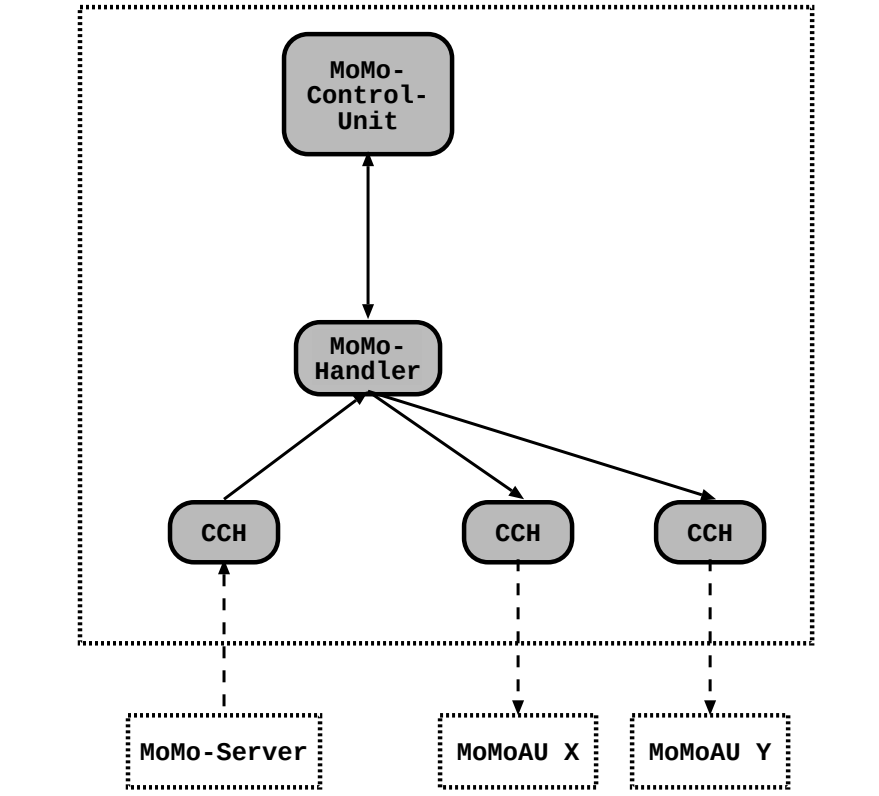
Die Remote-Verbindungen werden über einen Communication Handler (CCH) abgewickelt. Der CCH ist eine eigene Task, die die TCP-Daten zwischenpuffert und die Verbindung mit Alive-Telegrammen kontinuierlich überwacht. Somit ist eine Verbindung über einen CCH sehr robust.

Der MoMo-Handler stellt folgende Dienste zur Verfügung:

- **Connect/Disconnect**. Verbindungsaufbau/-abbau zu einem Remote-Rechner (auf dem auch ein MoMo-Handler installiert sein muß).
- **List, Delete**. Anzeigen aller Objekte und löschen eines/aller Objekts/Objekte auf dem Remote-Rechner.
- **Execute**. Ausführen einer Konfigurationszeile oder einer kompletten Konfigurationsdatei (siehe Kap. 2.2.11).
- **Send/Receive**. Daten oder komplette Dateien von dem lokalen zum Remote-Rechner übertragen oder vom Remote-Rechner empfangen.
- **Set_TCS**. Schreiben in den Public Remote Teil des Task Control Segments (siehe Kap. 2.2.10).

2.2.8 MoMo-Control-Unit

Die MoMo-Control-Unit (MoMo-CU) ist die zentrale Verwaltungseinheit einer MoMo-AU. Jede Task muß sich bei der MoMo-CU anmelden und bekommt dann eine ID, unter der sie sich zukünftig immer bei der Control-Unit meldet. Die Control-Unit nimmt folgende Aufgaben war:



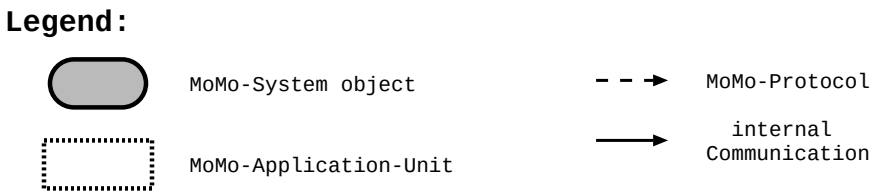
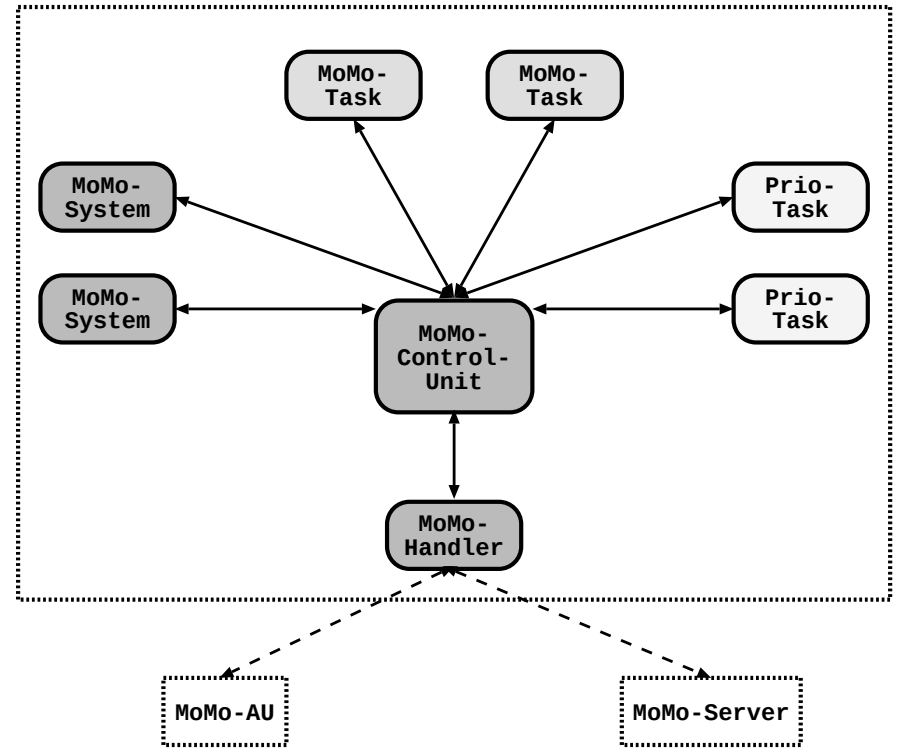


Abbildung 2.5: MoMo-Handler Kommandofluß

- Zustandssteuerung aller Tasks auf der MoMo-AU (siehe Kapitel 2.2.3).
- Überwachung aller Tasks. Im Fehlerfall wird zwischen Prio-Tasks und anderen Tasks unterschieden: Im Fehlerfall einer Prio-Task werden alle Tasks in einen sicheren Zustand geschaltet und eine Meldung an der MoMo-Server geschickt. Bei den anderen Tasks wird versucht, die fehlerhafte Task nach einer Reinitialisierung wieder in den vorherigen Zustand zu bringen. Wenn dies nicht gelingt, wird eine Meldung an der MoMo-Server geschickt, aber die anderen Tasks werden davon nicht beeinflußt.
- Weiterleitung der Remote-Kommandos, die sich auf alle Tasks in einer MoMo-AU beziehen.
- Bearbeitung aller MoMo-Kommandos.



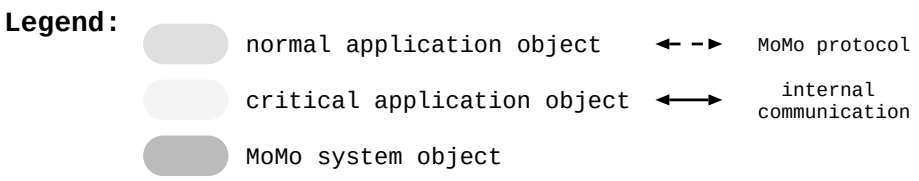


Abbildung 2.6: MoMo-Control-Unit

- Erweiterbar um applikationsspezifische Kommandos (z.B. in der ARTEMIS-Anwendung wird hier das MONSUN-Protokoll realisiert)
- Sammeln aller systemspezifischen Daten (z.B. CPU-Auslastung)

In Abbildung 2.6 ist die MoMo-Control-Unit im Umfeld der MoMo-Application-Unit zu sehen. Sie hat Verbindung zu allen Tasks auf dieser MoMo-AU, da sie ihnen die Kommandos schicken können muß. Die MoMo-CU bekommt ihrerseits die Kommandos über den MoMo-Handler (z.B. zum Weiterschalten der Tasks), die entweder von einer anderen MoMo-AU oder dem zentralen MoMo-Server kommen können.

2.2.9 Schnittstellen-Objekte

Ein Schnittstellenobjekt wird durch ein Shared Memory Segment (genannt Interface Control Segment, ICS) repräsentiert. Jedes Objekt auf einer MoMo-

Application-Unit besitzt ein ICS; somit sind alle Objekte (z.B. Tasks, Mailboxen, TCP-Verbindungen, ...) auf einer MoMo-AU über eine einheitliche Schnittstelle ansprechbar. Dieses ICS gliedert sich in folgende Bereiche:

- **Header**. Der Header enthält allgemeine Daten, die für jedes ICS-Objekt benötigt werden (z.B. Länge des ICS, Parent-ID, Segmenttyp, Objekttyp, ...).
- **Statischer Teil**. Im statischen Teil sind allgemeine Informationen über das Schnittstellenobjekt gespeichert, die sich selten ändern (z.B. Name des ICS, Anzahl der zugreifenden Tasks, ...).
- **Dynamischer Teil**. Im dynamischen Teil sind dagegen allgemeine Informationen über das Schnittstellenobjekt gespeichert, die etwas über den aktuellen Zustand des Objekts aussagen (z.B. Anzahl gesendeter/empfangener Nachrichten, aufgetretene Fehler, ...).
- **Spezifischer Objektdatenteil**. Hier werden die objektspezifischen Daten gespeichert. Es existieren folgende Objekttypen:
  - MoMo-Mailbox
    Dient zum Austausch von Daten zwischen zwei Objekten auf einem Rechner und besteht aus einer Sende- und Empfangs-Mailbox mit mehreren MoMo-Segmenten. Alle MoMo-Segmente werden bei der Initialisierung angelegt und dann als Queue verwaltet.
  - MoMo-Segment
    Dient zum Austausch von Daten zwischen zwei Objekten auf einem Rechner und beruht auf dem MoMo-Pool (siehe Kap. 2.2.6). Ein MoMo-Segment besitzt wenig System-Overhead, ist dafür aber nicht synchronisiert.
  - Netzwerk
    Dient zum Austausch von Daten zwischen zwei Rechnern. Unterstützt wird TCP/IP und UDP/IP.

2.2.10 Task-Objekte

Jedes MoMo Task-Objekt besteht aus zwei Teilen:

- **Task Control Segment (TCS)**. Das TCS ist spezieller Shared Memory Bereich dem ein ICS mit Objektdatenteil Segment zugeordnet ist (siehe Kap. 2.2.11).
  Das TCS stellt eine Schnittstelle zu anderen MoMo-Objekten dar. Innerhalb des TCS existieren mehrere verschiedenen Bereiche, die von folgenden Typen sein können:
  - Public Remote
    Auf diesen Bereich können andere Objekte (nur System-Objekte) schreibend zugreifen, das eigene Objekt jedoch nur lesend. Er dient zur Kontrolle der Task: hier können von extern Breakpoints, der MoMo-Taskzustand oder interne Parameter (z.B. Wartezeit pro Zustand) gesetzt werden.

- Public Local
  Hier kann das eigene Objekt schreibend und andere Objekte lesend zugreifen. Dieser Bereich dient zur Bekanntgabe wichtiger Daten an andere Objekte (z.B. MoMo-Taskzustand, zugehörige Task-ID und Taskname, Prozessorauslastung, ...).
- Private
  Dieser Bereich kann nur von dem eigenen Task-Objekt beschrieben und gelesen werden. Er ist vorgesehen zur Speicherung von persistenten globalen Variablen. Da das TCS unabhängig von der Task existiert, können hier Daten gespeichert werden, die nach einem fatalen Fehler (oder absichtlichem Löschen der Task) der neuen Task wieder zur Verfügung stehen müssen.

- **Task**. Die Task ist das ablauffähige Programm, das das dazugehörige TCS nutzt.

2.2.11 Konfigurationsmanagement

Ein MoMo-System kann entweder über eine Konfigurationsdatei und das Start-Programm oder Remote über den MoMo-Handler (z.B. mittels des Konfigurations-Client) konfiguriert werden. Für die Konfiguration macht das keinen Unterschied, da der MoMo-Handler von Remote nur einzelne Zeilen einer Konfigurationsdatei geschickt bekommt.

Eine Konfigurationsdatei ist zeilenweise aufgebaut, wobei jede Zeile genau ein MoMo-Objekt definiert. Das Beispiel einer Konfigurationsdatei findet sich im Anhang B. Der erste Buchstabe in einer Zeile bestimmt den Typ des MoMo-Objekts, wobei folgende Typen zur Auswahl stehen:

- **Standardisierte Schnittstellen („*n*“)**. Für diese MoMo-Objekte wird automatisch immer ein Interface Control Segment generiert und im MoMo-Katalog katalogisiert. Standardschnittstellen können Mailboxen oder Segmente sein, wobei ein Task Control Segment auch unter den Typ Segment fällt.

- **Standardisierte Remote-Schnittstellen („*r*“)**. Entspricht einer standardisierten Schnittstelle, wird jedoch nicht auf dem lokalen, sondern auf einem Remote-Rechner angelegt. Hiermit kann eine MoMo-AU auf mehrere Rechner verteilt werden (siehe Anhang C).

- **Alias („*a*“)**. Alias für eine standardisierte Schnittstelle. Hierüber kann das System unabhängig von den einzelnen Tasks konfiguriert werden. Per Alias lassen sich beliebige Task-Ausgabeschnittstellen auf beliebige Eingabeschnittstellen von anderen Tasks legen. Somit kann der Informationsfluß in einer MoMo-AU ohne Ändern des Programmcodes flexibel konfiguriert werden.

- **Tasks („*t*“)**. Hiermit können die MoMo-System Tasks und die Anwendungstasks gestartet werden.

- **Reale Kommunikationsobjekte („*k*“)**. Besteht aus einem katalogisierten MoMo-Segment, in dem die Parameter abgelegt werden, und dem Channel Communication Handler (siehe Kap. 2.2.2).

- **Virtuelle Kommunikationsobjekte („*v*“)**. Katalogisiertes MoMo-Segment mit der Verbindungsbeschreibung für ein reales Kommunikationsobjekt. Es können beliebig viele virtuelle Kommunikationsobjekte pro realem Kommunikationsobjekt angelegt werden, um damit das reale Kommunikationsobjekt dynamisch zur Laufzeit zu konfigurieren.

- **Hilfsfunktionen:**

  - Text („**c**“)
    Einzeiliger Text zur Ausgabe auf dem Bildschirm.
  - Wartezeit („**z**“)
    Zeit in Sekunden (als Gleitkommazahl), die vor der Ausführung der nächsten Konfigurationszeile gewartet werden soll. Damit kann man sicherstellen, daß die vorherige Aktion (z.B. Starten einer Task) auch komplett ausgeführt wurde, bevor ein darauf aufbauendes Objekt erzeugt wird.
  - Kommentar („*****“)
    Einzeiliger Kommentartext innerhalb der Konfigurationsdatei.

Die Syntax der einzelnen MoMo-Objekte findet sich in Anhang A.

2.2.12 Informationsfluß

Der Informationsfluß wird auf zwei Ebenen betrachtet: einerseits auf MoMo-Application-Unit Ebene, d.h. welche Schnittstellen hat eine MoMo-AU zum Rest des MoMo-Systems; und andererseits auf Application Task Ebene innerhalb der MoMo-AU, d.h. welche Schnittstellen hat eine Application Task zum Rest der MoMo-AU.

MoMo-Application-Unit

Eine MoMo-AU hat folgende Schnittstellen nach Außen:

- **MoMo-Handler**. Hierüber empfängt die Einheit MoMo Kommandos (meistens vom MoMo_Server), oder sendet an andere MoMo-Handler Kommandos (z.B. MoMo-Server an alle MoMo-AUs).

- **MoMo Cyclop**. Verschickt zyklisch per UDP allgemeine Systeminformationen.

- **Application Cyclop**. Verschickt zyklisch per UDP anwendungsspezifische Informationen. Es können beliebig viele Application Cyclop Tasks gestartet werden und mit verschiedenen Frequenzen versehen werden. So kann die Netzbelastung minimiert werde, da verschiedene Gruppen für die unterschiedlichen Änderungsfrequenzen der Daten gebildet werden können.

- **Anwendungsspezifische Netzschnittstellen (CCH, TCP, UDP)**. Hiermit können MoMo-AUs miteinander effektiv größere Datenmengen austauschen.

- **Logfile**. Ereignisorientierte Protokollierung des Systems in einer Datei (lokal oder Remote) mit Zeitstempel.

Application Object

Ein Application Object kann nur über IPC-Dienste (siehe Kap. 2.2.2) mit anderen MoMo-Objekten kommunizieren. Die IPC-Dienste lassen sich wie folgt unterteilen:

- **Kommando-Eingabe**. Alle kommando-orientierten Informationen von der MoMo-Control-Unit kommen über diese Schnittstelle herein. Dies können entweder MoMo-Kommandos (z.B. Wechsel in einen neuen MoMo-Zustand) oder anwendungsspezifische Kommandos sein.

- **Kommando-Ausgabe**. Diese Schnittstelle geht direkt zur MoMo-Control-Unit. Hierüber werden alle Kommandos bestätigt, und es können anwendungsspezifische Kommandos an den MoMo-Server geschickt werden.

- **Daten-Eingabe**. Standardeingabe-Schnittstelle jeder Task. Durch Konfiguration kann sie auf die Standardausgabe-Schnittstelle einer anderen Task gelegt werden (siehe auch Kap.2.2.11) .

- **Daten-Ausgabe**. Standardausgabe-Schnittstelle jeder Task. Auch diese Schnittstelle kann durch Konfiguration auf eine Eingabe-Schnittstelle einer anderen Task gelegt werden.

- **Anwendungsspezifische Schnittstellen**. Falls der Anwendung die zwei Standard-Schnittstellen nicht ausreichen, kann sie auch beliebig viele Schnittstellen selber anlegen. Im Unterschied zu den Standard-Schnittstellen muß jedoch die Anwendung diese Schnittstellen verwalten.

- **Task-Control-Segment**. Jede MoMo-Task hat ein ihr zugeordnetes Task Control Segment, das sich in drei Bereiche gliedert mit unterschiedlichen Segmenttypen (zu den verschiedenen Typen siehe auch Kap. 2.2.10):

  - Allgemeinteil (vom Typ Public Local)
    Dieser Teil wird nicht direkt von der Anwendungstask beschrieben, sondern von den MoMo-Hilfsfunktionen (z.B. check()). Daten in diesem Teil sind Taskdaten (z.B. Task-ID, Taskname, Priorität, Task-Zustand) und die Systembelastung durch diese Task. Von extern wird dieser Teil vom MoMo-Cyclop und vom MoMo-Logger gelesen, um die entsprechenden Informationen per UDP zu versenden (Cyclop), oder in eine Datei zu schreiben (Logger).
  - Steuerteil (vom Typ Public Remote)
    Über dieses Segment kann die Task von außen (über den MoMo-Handler) Steuerbefehle bekommen. Es gibt z.B. Felder zum Setzen von Breakpoints (oder Breakpoint-Bereiche), des MoMo-Zustands und der Priorität.
  - Externer anwendungsspezifischer Teil (vom Typ Public Local)
    Hier kann die Anwendungstask Informationen zur Verfügung stellen, die mit den Application-Cyclops per UDP versendet werden können. Hierüber werden z.B. in ARTEMIS von der Zyklustask die aktuellen Gelenkwinkel des Slaves in das Segment geschrieben und dann mittels eines Application-Cyclop versendet. Eine Monitoring-Einheit

fängt diese Information auf und kann so die Slave-Einheit graphisch monitoren (weiteres siehe Kap. 3.1 KISMET).

- Interner anwendungsspezifischer Teil (vom Typ Private)
  Dient zum persistenten Speichern von globalen Variablen (siehe Kap. 2.2.2).

Die Verwendung dieser Informationen zu Debugging-Zwecken wird in Kapitel 2.2.14 näherer erläutert.

2.2.13 Verteilung einer MoMo-Application-Unit

Wie schon in der Einleitung zur MoMo-Application-Unit erwähnt, braucht eine MoMo-AU nicht auf einen Rechner beschränkt zu sein. Eine MoMo-AU definiert sich durch einen Rechnerverbund, bei dem auf jedem Rechner ein MoMo-Handler und ein MoMo-Katalog und auf einem Rechner zusätzlich eine MoMo-Control-Unit vorhanden sind. Von außen ist nur der Rechner mit der MoMo-CU sichtbar. Auch die Konfigurationsdatei wird nur auf dem Rechner mit der MoMo-CU ausgeführt, wobei die Verteilung mit dem MoMo-Objekttyp „*r*“ vorgenommen wird (siehe Kap. 2.2.11).

In Abbildung 2.7 ist ein Beispiel einer Verteilung einer MoMo-AU zu sehen. Im oberen Bildteil ist die MoMo-AU komplett auf einem Rechner. Von außen bekommt die MoMo-AU ihre Kommandos von dem MoMo-Server. Der interne Informationsfluß ist folgender: Die Task *InTask* bekommt Eingaben, die sie über die MoMo-Mailbox-Schnittstelle *MBx2* an die *CalcTask* weitergibt. Diese berechnet aus den Eingaben neue Werte und schreibt diese in die MoMo-Mailbox *MBx1*. Von dort liest sie *OutTask* und schreibt sie auf den Bildschirm. Wenn nun dieser Rechner mit der *CalcTask* überfordert ist, kann man die MoMo-AU auf zwei Rechner verteilen. Dies ist im unteren Bildabschnitt zu sehen. *CalcTask* und die beiden MoMo-Mailboxen wurden auf einen zweiten Rechner ausgelagert. Das Schreiben und Lesen auf diesen MoMo-Mailboxen geht nun über die beiden MoMo-Handler dieser Rechner. Für die Tasks *InTask* und *OutTask* erfordert dies keine Änderungen; sie können keinen Unterschied zu der oberen Konfiguration feststellen. MoMo-Katalog und MoMo-Handler verstecken dies vor den Tasks.

In Anhang C findet sich eine Beispiel einer Konfigurationsdatei für eine verteilte MoMo-AU.

2.2.14 Debugging und Fehlerbehandlung

Grundsätzlich gibt es zwei Arten, wie man Informationen von einem System exportieren kann:

1. **Zyklisch**. Bei der MoMo-AU wird dies durch die zwei Cyclop-Typen realisiert, die zyklisch per UDP Daten versenden (siehe Kap. 2.2.4). Dies erfordert keinen Eingriff in das System und beeinflußt somit das System in keiner Weise (nicht-invasiv).

2. **Ereignisorientiert**. Bei einer MoMo-AU kann dies von außen durch den MoMo-Handler geschehen, oder wenn interne Ereignisse (z.B. Fehler, Erreichen eines Breakpoints) auftreten erfolgen. Informationen werden

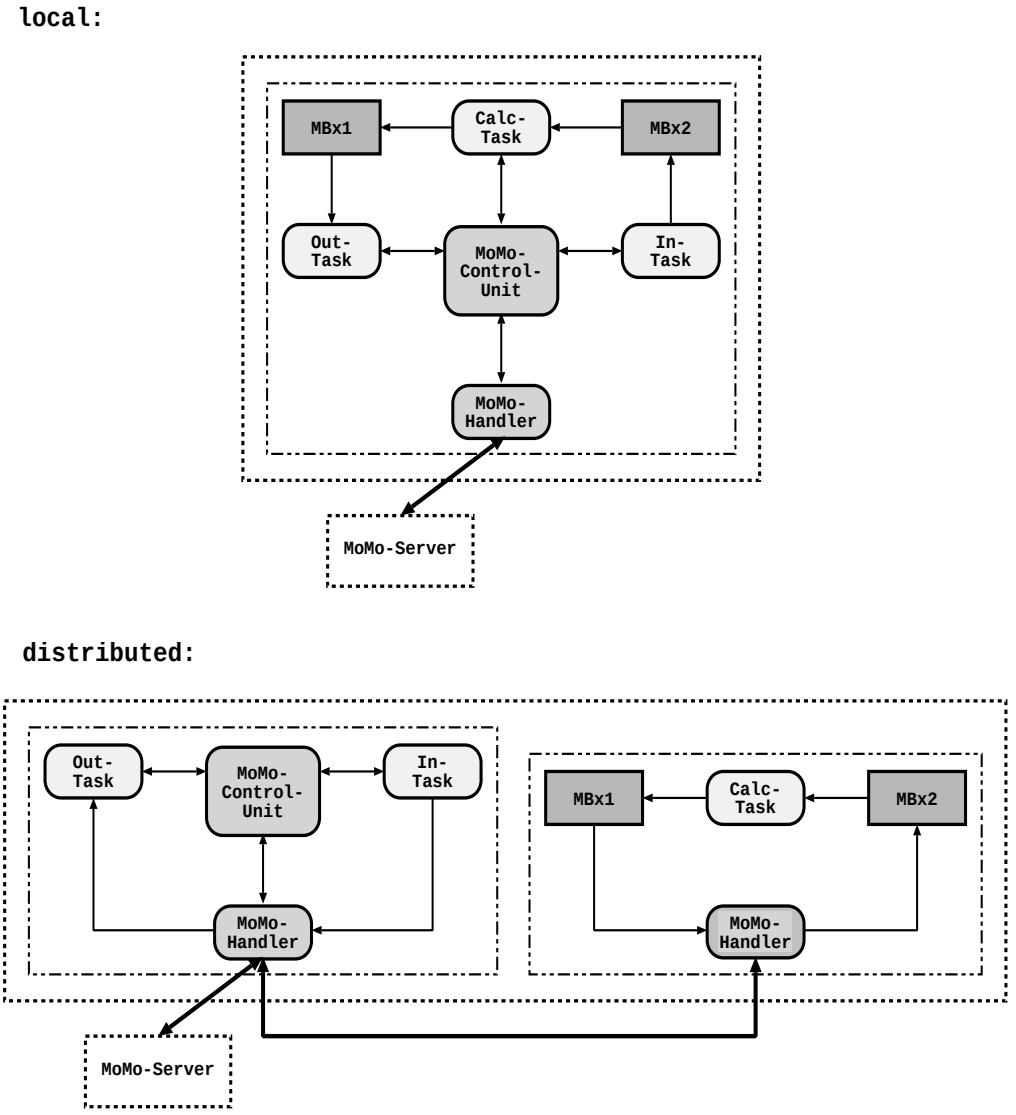


Abbildung 2.7: Verteilung einer MoMo-AU

per MoMo-Handler an den MoMo-Server weitergeben und/oder in einer Log-Datei gespeichert.

Beide Möglichkeiten werden bei MoMo genutzt, um Debugging und die Fehlerbehandlung so komfortabel und flexibel wie möglich zu realisieren.

Debugging

Das von MoMo zur Verfügung gestellte Debugging ist ein Funktions-Debugging, das plattformunabhängig und Remote zur Verfügung steht. Dies geschieht innerhalb der Task über den Task Control Block (siehe auch Kap. 2.2.12) und die Checkpoints (*check()*-Prozedur) und zwischen MoMo-AU und Remote-Debugging-Client mittels des Cyclop (für nicht-invasives Überwachen) und dem MoMo-Handlers (für direkte Interaktion mit der MoMo-AU).

Eine Unterbrechung der Task ist nur an den jeweiligen Checkpoints möglich[14]. Je mehr Checkpoints in dem Code sind, desto feiner wird die Debug-Granularität.

Der Task Control Block stellt folgende Informationen für das Debugging zur Verfügung:

- **MoMo Zustand**. $\in [Init, Basic, Ready, Active, End]$, Zustand des Taskautomaten (siehe Kap. 2.2.3).
- **Task Zustand**. $\in [Running, Stopped, Died]$, Betriebssystemzustand der Task:
  Running = Task läuft
  Stopped = Task wartet an einem Checkpoint
  Died = Task wurde vom Betriebssystem als ungültige Task eingestuft
- **Systembelastung**. CPU-Zeit (seit Start und seit letztem Checkpoint) und prozentuale Belastung des Rechners durch diese Task.
- **Letzter Checkpoint**. Nummer des letzten besuchten Checkpoints und Zeitstempel, wann dieser Checkpoint besucht wurde.
- **Zustandsabhängige Zeiten**. Zeiten, die nach jedem zyklischen Durchlauf eines Zustandes gewartet wird. Für *Basic*-, *Ready*- und *Active*-Zustand getrennt einstellbar. Außerdem wird noch protokolliert, wie lange die Task in dem jeweiligen Zustand verbracht hat.
- **Anwendungsspezifische Daten**. Hier können Daten, die beim Debugging dieser speziellen Task hilfreich sind, aufgenommen werden und exportiert werden (z.B. beim CCH Host und Portnummer der Remote-Verbindung).

Über den Steuerteil des TCS kann die Task wie folgt beeinflußt werden:

- **MoMo-Zustand**. Setzen eines neuen MoMo-Zustandes.
- **Task-Zustand**. Wechsel zwischen den Zuständen *Running* und *Stopped* möglich.
- **Priorität**. Setzen einer neuen Taskpriorität.

[14] Es gibt allerdings auch eine Möglichkeit, für kritische Abschnitte die Unterbrechung außerkraft zu setzen.

- **Schleifenzeiten**. Ändern der MoMo-Zustandsschleifenzeiten.
- **Breakpoints**. Es gibt die Möglichkeit, einen Breakpoint zu setzen auf:

$=$	Checkpoint Task hält, sobald Checkpoint erreicht wird (ein *Checkpoint* mit einer eindeutigen Nummer) .
$<$	Checkpoint Task hält, sobald ein Checkpoint mit einer größeren Nummer als der angegebene Checkpoint erreicht wird.
$>$	Checkpoint Task hält, sobald ein Checkpoint mit einer kleineren Nummer als der angegebene Checkpoint erreicht wird.
X, Y	Checkpoint1, Checkpoint2 Task hält, sobald ein Checkpoint innerhalb des Bereichs zwischen Checkpoint1 und Checkpoint2 erreicht wird.

  Durch geschickte Gruppierung der Checkpoint-Bereiche in der MoMo-Software kann der Anwender einfach Breakpoint auf Zustandswechsel, Aufruf von MoMo-Hilfsdiensten, Aufruf von Pattern-Funktionen, Aufruf von Schnittstellenfunktionen, ... setzen. Es können sowohl mehrere Breakpoints wie auch mehrere Breakpoint-Bereiche gesetzt werden.
- **Debug-Modes**. Mit den Modes teilt man der Task mit, wie sie auf Check- und Breakpoints reagieren soll:

Run	Start der Task und ignoriere alle Breakpoints.
Stop	Stop beim nächsten Checkpoint.
Cont	Starten der Task und stopp beim nächsten Breakpoint.
Step	Starten der Task und stopp beim nächsten Checkpoint.

Alle Einflußmöglichkeiten können sich entweder auf alle Tasks der MoMo-AU beziehen (wenn das Kommando an die MoMo-CU geschickt wird), oder nur auf eine spezielle Task innerhalb der MoMo-AU.

Fehlerbehandlung

Auch die Fehlerbehandlung wird mittels der Checkpoints (*check()*-Aufruf) realisiert. Es ist sinnvoll, nach dem Aufruf einer beliebigen MoMo-Funktion direkt anschließend einen Checkpoint zum Überprüfen des Return-Werts einzufügen. Als Parameter kann man einen Fehlertext und einen Hinweis auf das weitere Vorgehen im Fehlerfall mitgeben. Daraus ergeben sich mehrere Möglichkeiten: Je nach Klasse (fatal, tolerierbar) des Fehlers wird nur eine Meldung mit Zeitstempel und Task-ID in ein Logfile geschrieben (ist konfigurierbar), oder die Task in den Ende-Zustand versetzt. Man kann mittels der Checkpoints auch unabhängig davon, ob ein Fehler auftrat oder nicht, eine Meldung in das Logfile schreiben, um so wichtige Ereignisse zu dokumentieren.

Mit der Funktion *value_log()* lassen sich auch gezielt Werte in das Logfile zur Dokumentation schreiben. In Anhang F ist ein Beispiel eines Logfiles angeführt.

2.2.15 Programmierumgebung für den Applikationsentwickler

Um einen hohen Grad an Zuverlässigkeit und Software-Reuse zu erreichen, wurde jede Task in einen taskunabhängigen Taskframe und die taskabhängigen Task-Patterns unterteilt. Der Task-Frame liegt dem Anwendungsprogrammierer nicht als Sourcecode vor, sondern nur als ausführbares Modul. Somit kann eine strenge Trennung zwischen Anwendungscode und MoMo-Systemcode in jeder Task realisiert werden.

In prozeduralen Systemen ruft der Entwickler Prozeduren der Bibliotheken auf. Im Framework-Konzept wird dieses Prinzip herumgedreht. Hier ruft das Framework die Prozeduren des Entwicklers auf. Dies hat den Vorteil, daß die Infrastruktur und das Design im Framework festgelegt ist und der Entwickler sich auf sein zu lösendes Problem konzentrieren kann. Nach [Taligent, 97] bieten Frameworks einfache Erweiterbarkeit, Interoperabilität, einfache Wartung und erhöhte Verläßlichkeit. Weiteres zu Software Reuse findet sich in [Garlan *et al.*, 95], zu Frameworks in [Booch, 94] und zu Design-Patterns in [Gamma *et al.*, 94].

Task-Frame

Der für alle Tasks identische Task-Frame besteht aus:

- **Task initialisieren**. Initialisieren aller Task-Frame Variablen und Einlesen einer Task-Frame-Parameterdatei.
- **Taskautomat**
  (siehe Kap. 2.2.3)
- **Kommunikationsprotokoll mit MoMo-Control-Unit**. Behandeln aller Kommandos von der MoMo-CU (siehe auch Kap. 2.2.3 und 2.2.12). Alle dem Task-Frame unbekannten Kommandos werden an ein entsprechendes Task-Pattern übergeben.
- **Fehlerbehandlung**. Behandeln aller aufgetretenen Fehler und entsprechend Meldung an die MoMo-CU.
- **Task beenden**. Freigeben aller Ressourcen.

Task-Pattern

Die Task-Patterns sind die einzige[15] Möglichkeit des Anwendungsprogrammierers, den Anwendungscode in das MoMo-System zu integrieren. Dies hat den Vorteil, daß Task-Frame mit Task-Automat und Protokoll zur MoMo-CU ausgetestet und unveränderlich sind. Der Nachteil ist natürlich, daß eine Anwendung immer in die vordefinierte Taskschablone gepreßt werden muß, auch wenn es manchmal elegantere Lösungen mit mehr Taskzuständen gibt[16]. Der Vorteil eines sehr gut ausgetesteten Task-Frames wiegt diesen Nachteil unserer Ansicht nach aber meistens auf.

[15] Jedenfalls innerhalb des MoMo-Taskkonzeptes.

[16] Es gibt natürlich immer die Möglichkeit innerhalb eines Zustands einen extra Automaten für eine spezielle Anwendung zu definieren. Dies ist in Fällen hilfreich, wo ein MoMo-Zustand (z.B. *Aktiv-State*) in eine feinere Struktur aufgegliedert werden soll.

Folgende Task-Pattern sind möglich:

- **Init**. Wird nur im *Init_State* aufgerufen. Hierüber bekommen die Pattern die Taskparameter des Task-Frames übermittelt.
- **End**. Wird nur im *End_State* aufgerufen. Letzte Möglichkeit für den Anwendungsprogrammierer alle Ressourcen freizugeben und eventuell vorhandene Hardware in einen sicheren Zustand zu setzen.
- **<STATE>_init**. Wird aufgerufen, wenn die Task in diesen Zustand wechselt.
- **<STATE>_exec**. Implementiert die Funktionalität des jeweiligen Zustands. Diese Funktion wird zyklisch[17]aufgerufen, solange sich die Task in diesem Zustand befindet.
- **<STATE>_com**. Wird immer dann aufgerufen, wenn ein Kommando von der MoMo-AU kommt, das kein Standardkommando ist. Hierüber lassen sich anwendungsspezifische Kommandos verarbeiten (z.B. MONSUN-Protokoll in der ARTEMIS-Anwendung).
- **<STATE>_fin**. Wird beim Verlassen des Zustands aufgerufen.

mit <STATE> $\in [Basic, Ready, Active]$.

In Anhang D sind die Patterns für ein einfaches Beispiel, die Calc-Task aus Kapitel 2.2.13, angeführt. Als komplexeres Beispiel sind in Anhang E die Pattern für den Roboter-Zyklus des Endoskopführungssystems ROBOX aus dem ARTEMIS-System aufgeführt.

2.3 MoMo-Clients

Die MoMo-Clients stellen die Schnittstelle zwischen Benutzer und MoMo-System dar. Prinzipiell gibt es folgende Client-Klassen:

- **Anwendungsspezifisches MMI**. MMI für die realisierte Applikation.
- **Systemkonfigurierung und Systemwartung**. Hierunter fällt der Konfigurations-Client zur Konfigurierung des kompletten MoMo-Systems und die Überwachungs-Clients zur Protokollierung und Optimierung des Systems.
- **Entwickler MMI**. Zum einen wird die Erstellung einer neuen Applikation durch eine spezielle MoMo-Entwicklungsumgebung, den Applikations-Entwickler-Client, zum anderen das Debugging eines kompletten MoMo-Systems mit dem Debugging-Client unterstützt.

Konzipiert wurden die Clients als Java-Applets, so daß jeder Rechner mit einem Java 1.1 fähigen WWW-Browser (z.B. Netscape Communicator 4.0) die Clients vom MoMo-Server laden und lokal ausführen kann. Momentan realisiert sind ein anwendungsspezifischer MMI-Client (für die ARTEMIS Anwendung), ein Überwachungs-Client und der Debugging-Client.

Nachfolgend werden die einzelnen Clients vorgestellt.

[17]Die Zykluszeit ist für jeden Zustand getrennt einstellbar und von außen konfigurierbar (auch zur Laufzeit).

2.3.1 Anwendungsspezifische MMI-Clients

Mit diesem Client kann der Benutzer einer speziellen Anwendung Daten oder Befehle schicken und sich Daten der Anwendung darstellen lassen. Dies entspricht der klassischen graphischen Benutzeroberfläche.

Anwendungsspezifische Daten werden vom MoMo-Server einfach durchgereicht an die MoMo-Application-Unit. Dort nimmt sie die MoMo-Control-Unit entgegen und erkennt, daß es sich um kein MoMo-Kommando handelt und gibt das Kommando an das entsprechende Task-Pattern zur Bearbeitung weiter. Eine eventuelle Antwort nimmt dann den entgegengesetzten Weg.

2.3.2 Konfigurations-Client

Der Konfigurations-Client dient zur graphischen Konfiguration der MoMo-Application-Units. Damit kann das ganze MoMo-System mit allen MoMo-AUs, den MoMo-Objekten auf jeder MoMo-AU und den Verbindungen zwischen den einzelnen MoMo-AUs dargestellt und konfiguriert werden.

Die Konfigurationmöglichkeiten beruhen auf dem Konfigurationsmanagement aus Kapitel 2.2.11. Die Remote-Konfiguration per Client nutzt hierbei die entsprechenden MoMo-Handler-Dienste.

2.3.3 Überwachungs-Clients

Die Überwachungs-Clients dienen zur Darstellung von systemrelevanten Parametern für die Systemüberwachung. Sie beruhen auf den nicht-invasiven Debugging-Möglichkeiten der MoMo-AUs (siehe Kap. 2.2.14).

Prinzipiell sind mehrere Überwachungs-Clients zugelassen. Somit kann von mehreren Rechnern gleichzeitig das System überwacht werden, oder es können speziell auf eine Anwendung zugeschnittene Überwachungs-Clients erstellt werden.

Eine ausführliche Beschreibung eines Überwachungs-Clients findet sich in [Erhardt, 97].

2.3.4 Debugging-Client

Die Fehlersuche in verteilten Systemen stellt immer ein großes Problem dar. Da in MoMo mehrere Plattformen unterstützt werden, muß das Debugging betriebssystemunabhängig arbeiten.

Der Debugging-Client ist deshalb ein Systementwickler-MMI zum debugging der MoMo-AUs auf funktionaler Ebene. Die MoMo-Debug-Unterstützung ermöglicht dieses funktionale Debugging mit Hilfe der Checkpoints und des Task Control Blocks. Weiteres dazu findet sich in Kapitel 2.2.14.

Ein Beispiel für die Realisierung eines Debugging-Clients findet sich in [Höfele, 97].

2.3.5 Applikations-Entwickler-Clients

Dieses Systementwickler-MMI soll den Applikationsentwickler bei der Erstellung neuer MoMo-Anwendungen unterstützen. Hier können die einzelnen Pattern ausgewählt und mit Code gefüllt werden. Des weiteren werden die Möglichkeiten

der MoMo-Libraries interaktiv dargestellt und eine Inline-Dokumentation unterstützt.

2.4 MoMo-Server

Wie in Abbildung 2.1 ersichtlich, ist der MoMo-Server das Verbindungsglied zwischen den MoMo-Application-Units, den MoMo-Clients und den externen Datenquellen (z.B. Datenbanken, externe Geräte). Er nimmt außerdem administrative Aufgaben, wie Prüfung von Zugriffsberechtigungen, Konsistenzprüfungen und Protokollierung war.

Die Kommunikation zwischen Client und Server wird über TCP als Transportschicht und HTTP als Protokoll realisiert. Der Vorteil von HTTP liegt in seiner weiten Verbreitung als Standard-Internetprotokoll. So lassen z.B. alle Firewalls HTTP-requests ungehindert passieren.

Mit den MoMo-AUs kommuniziert der Server mittels des MoMo-Handlers über ein MoMo-eigenes Protokoll mit TCP als Transportschicht. Dieses Protokoll ermöglicht Remote Procedure Calls und ist somit weitaus mächtiger als HTTP. Aus dieser Mächtigkeit folgt aber auch die Möglichkeit zum Mißbrauch. Deshalb wird dieses Protokoll nur im sicheren Intranet verwendet und die Clients benutzen HTTP-Requests, die der Server nach einer Berechtigungsprüfung in MoMo Remote Procedure Calls transformiert.

Für den Client bietet der Server folgende Dienste an:

- Kommando X an MoMo-AU A schicken
- lies Datum X von Objekt Y, wobei
  X ein Datenfeld, ein MoMo Segment (TCS, ICS) oder eine Datei und
  Y eine MoMo-AU, externe Datenquelle oder ein MoMo Segment sein kann.
- schreibe Datum X von Objekt Y
  X und Y wie bei lesen

Realisiert wurde der Server in Java mit einer C-Anbindung für den MoMo-Handler. Zur Realisierung siehe auch [Kaufmann, 97]. Somit ist der Server nur auf Systemen lauffähig, auf denen auch der MoMo-Handler ablauffähig ist (zur Zeit Linux, IRIX, Windows NT).

Kapitel 3

Beispielanwendung: ARTEMIS

Am Beispiel von ARTEMIS, einer Master-Slave Anwendung, sollen die Vorteile des MoMo-Konzepts gezeigt werden. ARTEMIS (Advanced Robot and Telemanipulator System for Minimal Invasive Surgery) ist ein Telepräsenzsystem, das dem Chirurgen ermöglicht, minimal invasive Eingriffe im Bauchraum von einer Arbeitsstation aus durchzuführen. Das System besteht aus drei Slaves (zwei chirurgischen Arbeitseinheiten und einem Endoskopführungssystem) und aus diversen Master-Einheiten, wobei immer nur zwei Master-Einheiten gleichzeitig eingesetzt werden (pro Hand ein Master). Hieraus folgt, daß das System sehr flexibel sein muß, da sich die Koppelung zwischen Master und Slave (z.B. erst wird mit dem Master die chirurgische Arbeitseinheit bedient, dann das Endoskop) und auch der Typ des Masters sich ändern kann (z.B. kraftreflektierend/nicht kraftreflektierend, 1:1 Kinematik/Universalmaster). Das MoMo-Konzept entstand aus diesen Anforderungen und bietet deshalb eine hohe Flexibilität.

In diesem Kapitel werden die Anpassungen der MoMo-Architektur an ARTEMIS und die zwei Ausprägungen der MoMo-AUs (Master- und Slave-Einheit) für ARTEMIS vorgestellt.

3.1 Übersicht

In Abbildung 3.1 ist die MoMo-Architektur angepaßt auf die ARTEMIS-Anwendung zu sehen. Die wichtigsten Unterschiede zu dem allgemeinen MoMo-Konzept sind:

- Es gibt mehrere MMI-Clients, um die unterschiedlichen Master-Einheiten[1] mit ihren verschiedenen Funktionalitäten zu steuern.
- Die MoMo-Application-Units sind unterteilt in zwei Typen: Master- und Slave-Einheiten.

[1] Momentan stehen folgende verschiedene Typen zur Auswahl: 1:1 Kinematik, Universalmaster mit Kraftreflexion mit Handgriff, Universalmaster ohne Kraftreflexion als „Bleistift"-Master und ein frei im Raum positionierbarer Universalmaster ohne Kraftreflexion („Free-Flying-Master"). Einige der Master sind in zweifacher Ausführung für beidhändigen Betrieb vorhanden.

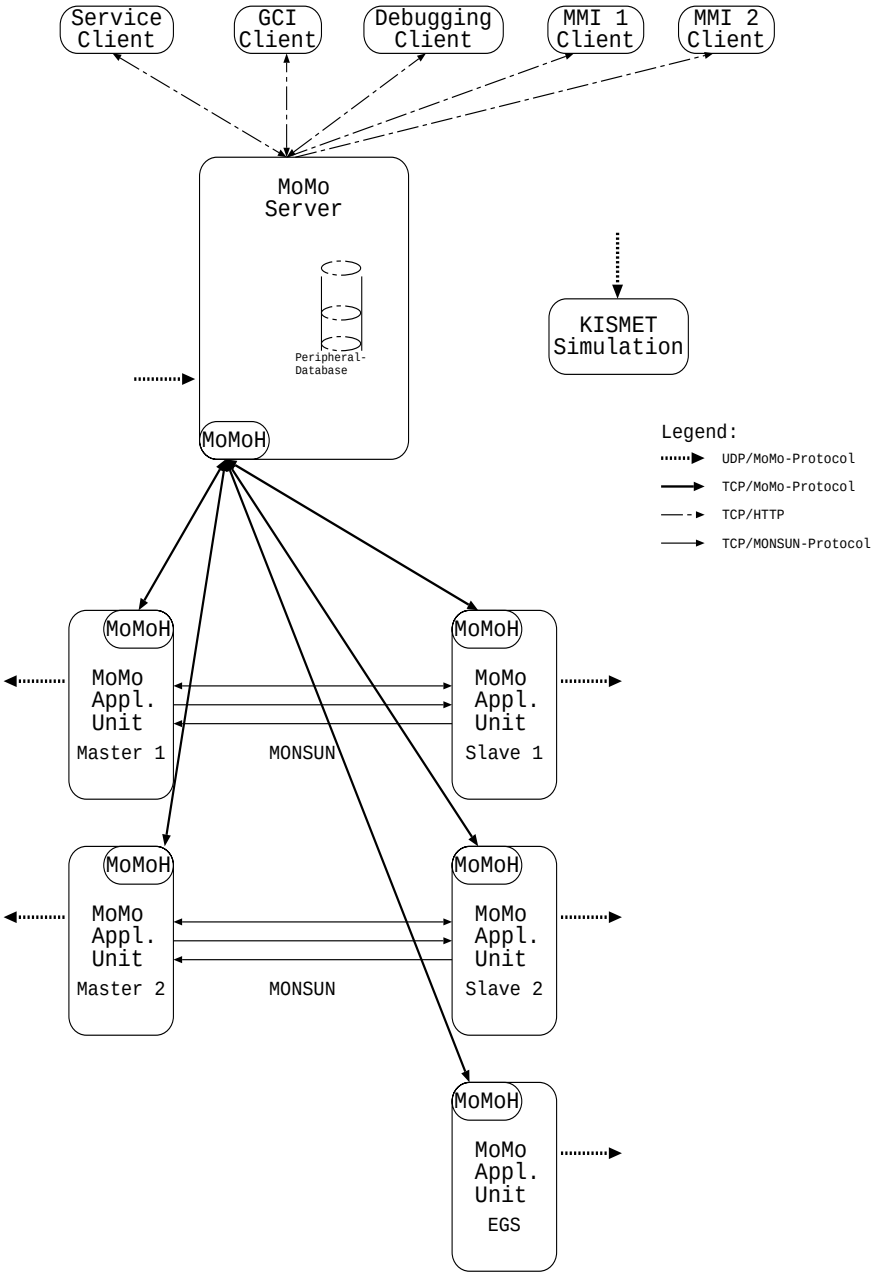


Abbildung 3.1: MoMo-Architektur für die ARTEMIS-Anwendung

- Je eine Master- und eine Slave-Einheit sind untereinander über drei TCP-Verbindungen verbunden, die als Protokoll das MONSUN-Protokoll[2] ver-

[2]Das MONSUN (Manipulator cONtrol System Utilizing Network technology) Protokoll entstand ebenfalls am Forschungszentrum Karlsruhe und ermöglicht die Steuerung von Slave-Einheiten mit Universal-Mastereinheiten über ein Ethernet LAN. Eine weitergehende Beschreibung von MONSUN findet sich in [Breitwieser & Weber, 96].

wenden. Dieses Protokoll erlaubt, beliebige Universal-Master mit beliebigen Slaves zu koppeln. So ist es auch einfach möglich, z.B. Master 2 mit dem Endoskopführungssystem (EGS) zu koppeln. Um eine schnelle, echtzeitfähige Kommunikation zwischen Master und Slave zu ermöglichen, existieren ein bidirektionaler Kommandokanal, ein unidirektionaler Datenkanal von Master zum Slave (Positionsdaten) und einer vom Slave zum Master (Kräfte- und Momentedaten).

- Die KISMET-Simulation[3] ist eine graphische 3D Monitoring-Einheit, die der Überwachung wie auch der Operationsplanung dient. Die Positionsdaten der einzelnen Slaves bekommt die Simulation über das MoMo-UDP-Protokoll mitgeteilt.
- Der MoMo-Server enthält eine Datenbank zur Koordinierung und Überwachung der Koppelungen zwischen den Benutzeroberflächen und den MoMo-AUs. Zudem werden im Server alle wichtigen Aktionen und Fehler protokolliert, was für eine medizinische Anwendung besonders wichtig ist.

3.2 ARTEMIS MoMo-Application-Units

Die ARTEMIS MoMo-Application-Units bestehen aus zwei Typen: den Master- und den Slave-Einheiten. In Abbildung 3.2 ist beispielhaft ein Master-Slave-Einheit zu sehen. Sowohl Master- als auch Slave-Einheit sind sehr ähnlich aufgebaut. Jede Einheit besteht aus Standard-MoMo-Objekten (CCH, Cyclops, MoMo-CU, MoMoH) und anwendungsspezifischen MoMo-Objekten, wie:

- **Cycle**. Die Cycle-Task ist ein kritisches Anwendungsobjekt vom Typ *Prio*, da es in direkter Verbindung mit der Hardware steht.
- **MSMDa, SetPoint, ActVal**. Diese Tasks fallen in die Klasse unkritische Objekte, da sie nur Daten aufbereiten.

Die Kommunikation zwischen Master und Slave wird über das MONSUN-Protokoll abgewickelt. Hierzu werden drei TCP-Verbindungen aufgebaut: Ein Kanal vom Master zum Slave zur Positionsvorgabe, ein Kanal vom Slave zum Master zur Kraftreflexion und ein bidirektionaler Kanal zum Austausch von Kommandos und Fehlermeldungen.

Anhand von Abbildung 3.2 kann man auch den Ablauf der Regelschleife zwischen Master und Slave erkennen:

1. Einlesen der Hardware (Winkel, Encoder) auf Masterseite durch die Cycle-Task (Zyklus), aufbereiten und kontrollieren der Werte und Weitergabe an ActVal (Actual Values; Istwerte-Aufbereitung).
2. ActVal berechnet die kartesischen Inkremente aus den Winkelvorgaben, transformiert diese in das MONSUN-Standardkoordinatensystem und gibt diese Werte über den Kanalhandler (CCH, Communication Channel Handler) des Masters an den Kanalhandler des Slaves weiter.

[3] Die KISMET (KInematic Simulation, Monitoring and off-line programming Environment for Telerobotics) Software ermöglicht Planung, Simulation, Programmierung und Monitoring von Telemanipulationsaufgaben und wurde auch im Forschungszentrum Karlsruhe entwickelt. Näheres zu KISMET findet sich in [Kühnapfel, 90].

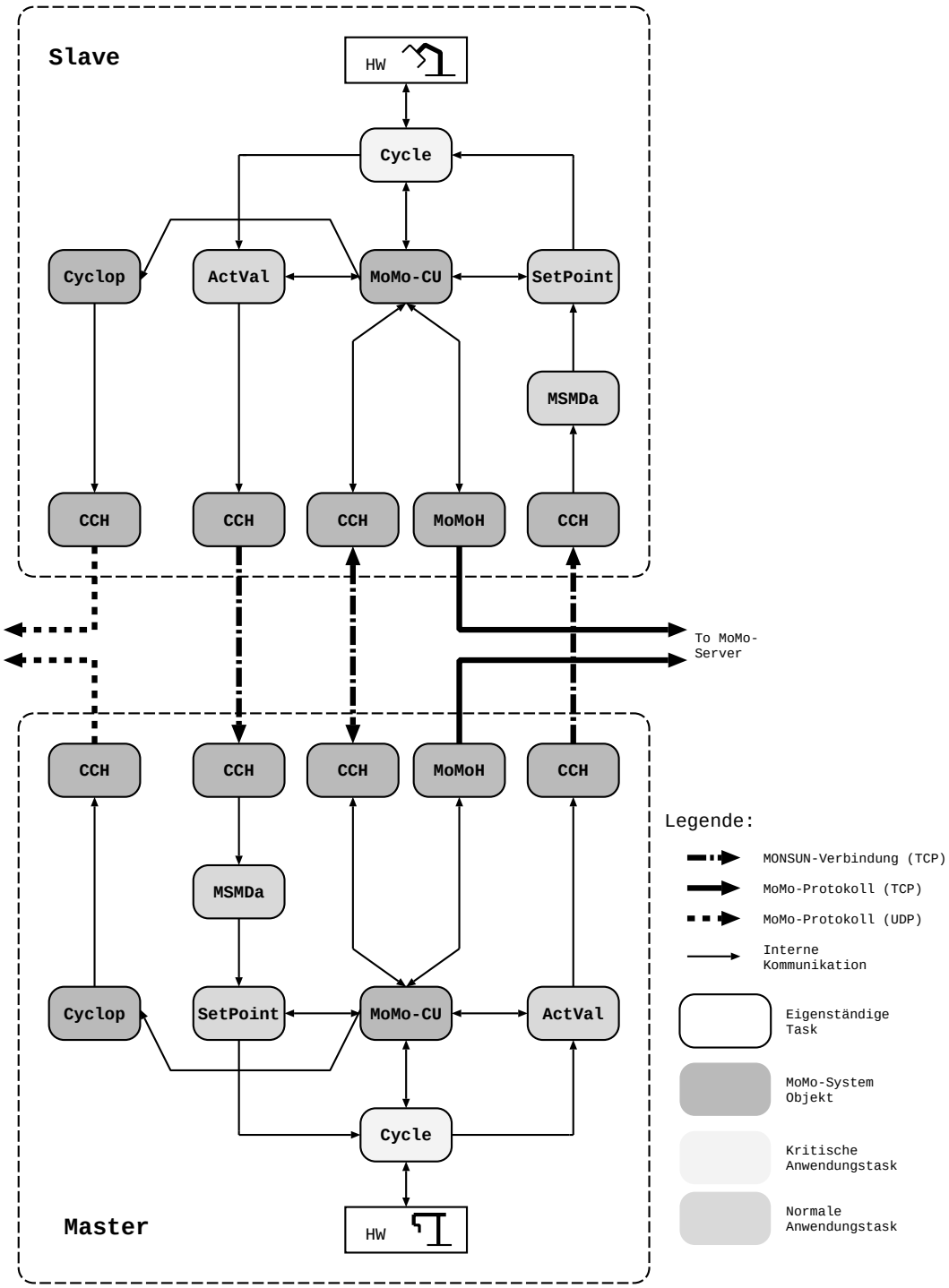


Abbildung 3.2: Master-Slave-Einheit

3. Auf Slave-Seite nimmt MSM-Da (Master-Slave-Manipulation-Data-Task) die kartesischen Inkremente entgegen und berechnet daraus Winkelinkremente für den Slave. Diese Inkremente werden an SetPoint (Sollwert-Aufbereitung) weitergegeben.

4. SetPoint (Sollwerte-Aufbereitung) berechnet aus den Winkelinkrementen Winkelvorgaben (geglättet) und gibt diese an den Zyklus weiter.

5. Cycle berechnet den Regler und stellt die Sollwerte, anschließend ließt sie die neuen Istwerte und eventueller Kraft-/Momentensensorik ein und gibt die Werte an ActVal weiter.

6. ActVal bereitet die Werte von Cycle auf (z.B. Eigendynamik eliminieren, Transformation in MONSUN-Standardkräfte und -momente) und übergibt sie per Kanalhandler an den Kanalhandler des Masters.

7. MSM-Da auf Master-Seite nimmt die Standardkräfte und -momente entgegen und transformiert sie in das Koordinatensystem des Masters.

8. SetPoint berechnet aus den Vorgabewerten neue Sollwinkel für den Master.

9. Berechnung des Reglers und Stellen der Sollwerte (Kraftreflexion) für den Master.

In Anhang E ist der Aktiv-Zustand eines allgemeinen Regelungszykluses beschrieben. Nach gleichen Muster ist auch der Zyklus für die Slave-Seite aufgebaut. Wie dort zu sehen ist, ist der Aufwand zur Anpassung minimal durch die Abtrennung von der Hardware durch die Hardware Abstraction Layer (Kap. 2.2.2, HAL) einerseits und das Framework mit vorgegebenen Automat (Kap. 2.2.15) andererseits.

Kapitel 4

Vergleich mit anderen verteilten Softwaresystemen

Grundsätzlich lassen sich wiederverwendbare Softwaresysteme, die auf einzelnen Komponenten beruhen (Component-Based Software Development, CBSD), in folgende Kategorien einteilen [Berg, 97]:

- **Componentware**. Hierunter fallen alle Systeme, die die Wechselbeziehung zwischen Komponenten mittels Objektmodellen (z.B. COM[1], CORBA[2]) oder Dokumentmodellen (OLE2[3], OpenDoc[4]) beschreiben. Es gehören auch alle graphischen Programmier-Tools (z.B. VisualAge von IBM und VisualBasic von Microsoft) und Komponenten-Frameworks (wie ActiveX[5] und JavaBeans[6]) in diese Kategorie. Weiteres hierzu findet sich in [Berg, 97].
- **Formale und wissensbasierte Techniken**. Hier wird die Komponentenerstellung und die Komponenten durch formale oder semi-formale Methoden beschrieben. Dieser Ansatz wird bisher nur in akademischen Projekten (z.B. SYNTHESIS von Prof. Kalinichenko, Russische Akademie der Wissenschaft) verfolgt.
- **Framework-Based**. Bei der Framework-Based Software Construction werden dem Systementwickler anwendungsspezifische Application Frameworks und Referenzarchitekturen zur Verfügung gestellt. Bei diesem Ansatz werden die Komponenten strukturiert im Hinblick auf eine spezielle Architektur, so daß die Interoperabilität zwischen den einzelnen Komponenten erhöht wird (z.B. DSOM[7]).

Während Componentware Reuse-by-Composition entspricht, ist dies bei Framework-Based Methoden hauptsächlich Reuse-by-Inheritance. MoMo fällt

[1] Component Object Model von Microsoft.
[2] Common Object Request Broker Architecture von OMG.
[3] Object Linking and Embedding von Microsoft
[4] von CILabs
[5] ActiveX von Microsoft erweitert Microsoft's OLE und COM Technologien zur Nutzung im World Wide Web.
[6] JavaBeans von Sun ist ein Framework zur Erstellung von Dokumenten und Applikationen basierend auf Java Komponenten.
[7] Framework einer CORBA Implementierung

in die Kategorie der Componentware und stellt ein anwendungsspezifisches Framework zur Verfügung. Von allen oben genannten Ansätzen ist MoMo der Einzige, der Realzeit-Anwendungen unterstützt.

Im Bereich der verteilten und objektorientierten Architekturen gibt es verschiedene Ansätze, die in drei Klassen unterteilt werden können:

1. **Entwurfsebene (design level)**. Beispiel eines Ansatzes auf der Entwurfsebene ist ANSA [Li, 1995]. ANSA ist eine offene, verteilte Architektur für Realzeit-Systeme und stellt einen Rahmen für den Entwurf eines verteilten Realzeitsystems zur Verfügung.

2. **Applikationsebene (application level)**. Ansätze auf der Applikationsebene nutzen existierende Betriebssysteme, um darauf ein spezielles Applikationssystem aufzusetzen. Ein Beispiel hierfür ist CommonPoint [Cotter & Potel, 95], eine Framework-basierte, objektorientierte und verteilte Umgebung für C++.

3. **Betriebssystemebene (OS level)**. Ziel dieses Ansatzes ist ein komplettes Betriebssystem zu realisieren. Ein Beispiel dieses Ansatzes ist COOL [Habert *et al.*, 90, Lea & Weightman, 91]. Chorus Object-Oriented Layer (COOL) ist ein System, das verteiltes objektorientiertes Programmieren unterstützt. Es basiert auf dem Chorus Betriebssystem-Kern, der ein verteiltes Betriebssystem realisiert.

Alle drei Konzepte haben Vor- und Nachteile. Für unsere Anwendung in der Telemanipulation erschien uns die Applikationsebene die meisten Vorteile zu bieten, da die Hersteller von Robotern und 3D-Eingabegeräten die verschiedensten Betriebssysteme (iRMX, Linux, IRIX) unterstützen. Um eine einheitliche Entwicklungsumgebung für alle Komponenten zu erreichen, mußte somit eine Schicht zwischen Betriebssystem und Anwendung eingefügt werden.

Deshalb ist MoMo ein verteiltes, objektorientiertes Framework auf Applikationsebene für Realzeitanwendungen, das auf bereits vorhandenen Betriebssystemen aufbaut und als Programmiersprache C unterstützt.

Anhang A

Konfigurationssyntax

Ein MoMo-System besteht aus einzelnen MoMo-Objekten, die entweder über eine Konfigurationsdatei und das Start-Programm oder Remote über den MoMo-Handler (z.B. mittels des Konfigurations-Client) konfiguriert werden. Eine Konfigurationsdatei ist zeilenweise aufgebaut, wobei jede Zeile genau ein MoMo-Objekt definiert. Das Beispiel einer Konfigurationsdatei findet sich im Anhang B. Der erste Buchstabe in einer Zeile bestimmt den Typ des MoMo-Objekts (siehe auch Kap. 2.2.11). Nachfolgend die Syntax der einzelnen Objekte:

n **SsName SsType NumSsSeg LenSsSeg TraceMode**
Konfiguration einer standardisierten Schnittstelle
SsName
Name, unter dem das ICS katalogisiert wird.
SsType [m, s]
Type der Schnittstelle (m = Mailbox-Schnittstelle, s = Shared Memory-Schnittstelle).
NumSsSeg
Anzahl der Schnittstellensegment (nur bei Typ m, bei Typ s gibt es immer nur ein Segment).
LenSsSeg
Länge der Schnittstellensegmente in Bytes.
TraceMode
Gibt an, wieviel Debug-Information über diese Schnittstelle gesammelt werden soll.

r **SsName SsType NumSsSeg LenSsSeg TraceMode Host**
Konfiguration einer standardisierten Remote-Schnittstelle
SsName, SsType, NumSsSeg, LenSsSeg, TraceMode
Siehe MoMo-Objekt „**n**“.
Host
Name des Remote-Rechners, auf dem die Schnittstelle eingerichtet werden soll. Auf diesem Rechner muß auch ein MoMo-Katalog und ein MoMo-Handler installiert sein.

a **AliasName SsName**
Alias für eine standardisierte Schnittstelle
AliasName

Aliasname für die standardisierte Schnittstelle
SsName
Name der standardisierten Schnittstelle

t **TaskName Prio**
Starten von Tasks
TaskName
Pfadname und Taskname des ausführbaren Programms.
Prio
Priorität der Task (betriebssystemspezifischer Wert!).

k **Prot Prio InSs OutSs Host Port Dir CCHName TraceMode**
Starten eines Communication Channel Handler (CCH)
Prot [TCP, UDP]
Protokolltyp
Prio
Priorität des CCH (betriebssystemspezifischer Wert!).
InSs
Name der Eingabeschnittstelle des CCH.
OutSs
Name der Ausgabeschnittstelle des CCH.
Host
Name des Remoterechners, zu dem die Verbindung aufgebaut werden soll.
Port
Port des Remoterechners.
Dir [a, p]
Richtung der Kommunikation (a = aktiver Verbindungsaufbau zu Remoterechner, b = passives Warten auf Verbindungsaufbau)
CCHName
Taskname des gestarteten CCH
TraceMode
Gibt an, wieviel Debug-Information der CCH an den MoMo-Logger schickt. Zudem kann das Urgent-Bit im TCP-Protokoll hier gesetzt werden.

v **Prot Host Port Dir ConnName TraceMode**
Virtuelle Verbindungsbeschreibung zur Konfiguration realer CCHs
Prot [TCP, UDP]
Protokolltyp
Host
Name des Remoterechners, zu dem die Verbindung aufgebaut werden soll.
Port
Port des Remoterechners.
Dir [a, p]
Richtung der Kommunikation (a = aktiver Verbindungsaufbau zu Remoterechner, b = passives Warten auf Verbindungsaufbau)
ConnName
Verbindungsname, unter dem die Verbindung aus der Anwendungstask angesprochen werden kann.

TraceMode
Gibt an, wieviel Debug-Information der CCH an den MoMo-Logger schickt. Zudem kann das Urgent-Bit im TCP-Protokoll hier gesetzt werden.

Hilfsfunktionen:

c **Text**
Einzeiliger Text zur Ausgabe auf dem Bildschirm

z **Time**
Wartezeit in Sekunden (als Gleitkommazahl)

***** **Text**
Einzeiliger Kommentar

Anhang B

Beispielkonfiguration einer MoMo-Application-Unit

Jede MoMo-AU kann durch eine Konfigurationsdatei beschrieben werden (siehe Kap. 2.2.11). Nachfolgend ist die Konfigurationsdatei der nicht-verteilten MoMo-AU aus Kapitel 2.2.13 (siehe auch Abb. 2.7).

```
*********************************************************
c  Konfiguration der MoMo-Anwendung
c  "nicht-verteilte MoMo-AU": (fuer iRMX)
*********************************************************
*
* 1. MoMo - Grundsystem:
*********************************************************
*  Logger und MoMoHandler werden vorab in den
*  Background geladen.
*********************************************************
*  Erzeugen und Katalogisieren der Kommando-
*  Schnittstellen (TCI) und der
*  Task-Kontroll-Segmente (TCS)
*-------------------------------------------------------
n  Control_TCS     s       1       512     1
*
n  TCI_Control     m       7       512     1                  Kom-Input
*                                                             der CU
*********************************************************
c  Start der MoMo-System-Tasks
*-------------------------------------------------------
t  :r:Control      208                                        Start MoMo-
*                                                             CU
*********************************************************
* 2. ANWENDUNG:
*
c  Erzeugen und Katalogisieren der
c  anwendungsspezifischen Schnittstellen
*-------------------------------------------------------
```

```
n  Calc_TCS         s       1       512     1
n  InTask_TCS       s       1       512     1
n  OutTask_TCS      s       1       512     1
*
*
*  Einrichten der anwendungsspez. Daten-Schnittstellen
*-------------------------------------------------------
n  Calc_In          m       3       512     1
n  Calc_Out         m       3       512     1
*
n  InTask_In        m       3       512     1
a  InTask_Out       Calc_In
*
a  OutTask_In       Calc_Out
n  OutTask_Out      m       3       512     1
*
*
*  Einrichten der Kommando - Schnittstellen zu
*  den Anwendungs-Tasks
*-------------------------------------------------------
n  TCI_Calc         m       3       512     1
n  TCI_InTask       m       3       512     1
n  TCI_OutTask      m       3       512     1
*
z  10
*******************************************************
c  Starten der Anwendungs-Tasks
*-------------------------------------------------------
k  TCP   224   0   OutTask_In any   sda   a   SollDa  1
z  10
t  :r:Calc          200
t  :r:OutTask       203
t  :r:InTask        201
*
*******************************************************
c   Initialisierung der Anwendung abgeschlossen !
*******************************************************
```

Die Bedeutung der einzelnen Konfigurationskommandos ist in Anhang A beschrieben. Die hier abgedruckte Konfigurationsdatei ist auf einem Rechner mit iRMX-Betriebssystem lauffähig (zu erkennen an dem logischen Laufwerksnamen „:r:“ und den Prioritäten).

Anhang C

Beispielkonfiguration einer verteilten MoMo-Application-Unit

In Anhang B wurde die Konfigurationsdatei der nicht-verteilten MoMo-AU aus Kapitel 2.2.13 beschrieben. Hier folgt nun die Konfigurationsdatei für die verteilte Variante dieser MoMo-AU (siehe auch Abb. 2.7).

```
*********************************************************
c  Konfiguration der MoMo-Anwendung
c  "verteilte MoMo-AU": (fuer iRMX)
*********************************************************
*
* 1. MoMo - Grundsystem:  - Host -
*********************************************************
*  Logger und MoMoHandler werden vorab in den
*  Background geladen.
*********************************************************
*  Erzeugen und Katalogisieren der Kommando-
*  Schnittstellen (TCI) und der
*  Task-Kontroll-Segmente (TCS)
*-------------------------------------------------------
n  Control_TCS     s       1       512     1
*
n  TCI_Control     m       7       512     1                  Kom-Input
*                                                             der CU
*********************************************************
c  Start der MoMo-System-Tasks
*-------------------------------------------------------
t  :r:Control      208                                        Start MoMo-
*                                                             CU
*********************************************************
* 2. ANWENDUNG:
*
```

```
c  Erzeugen und Katalogisieren der
c  anwendungsspezifischen Schnittstellen
*-------------------------------------------------------
n  InTask_TCS      s       1       512     1
n  OutTask_TCS     s       1       512     1
*
*
*  Einrichten der anwendungsspez. Daten-Schnittstellen
*-------------------------------------------------------
n  Calc_In         m       3       512     1
n  Calc_Out        m       3       512     1
*
n  InTask_In       m       3       512     1
r  InTask_Out      Trabant                                 Ausgelagerte
*                                                          Ss
a  OutTask_In      Calc_Out
n  OutTask_Out     m       3       512     1
*
*
*  Einrichten der Kommando - Schnittstellen zu
*  den Anwendungs-Tasks
*-------------------------------------------------------
n  TCI_InTask      m       3       512     1
n  TCI_OutTask     m       3       512     1
*
z  10
*********************************************************
c  Starten der Anwendungs-Tasks
*-------------------------------------------------------
k  TCP   224   0   OutTask_In any   sda   a   SollDa  1
z  10
t  :r:Calc         200
t  :r:OutTask      203
t  :r:InTask       201
*
*********************************************************
c   Initialisierung der Anwendung abgeschlossen !
*********************************************************

*********************************************************
* 3. Ausgelagerter ANWENDUNGS-Teil:  - Trabant -
*
c  Erzeugen und Katalogisieren der
c  anwendungsspezifischen Schnittstellen
*-------------------------------------------------------
n  Calc_TCS        s       1       512     1
*
*
*  Einrichten der anwendungsspez. Daten-Schnittstellen
*-------------------------------------------------------
```

```
n  InTask_Out      m       3       512     1
a  Calc_In         InTask_Out
r  Calc_Out        Host                                          Ausgelagerte
*                                                                Ss
*
*  Einrichten der Kommando - Schnittstellen zu
*  den Anwendungs-Tasks
*--------------------------------------------------------
n  TCI_Calc        m       3       512     1
*
z  10
**********************************************************
c  Starten der Anwendungs-Tasks
*--------------------------------------------------------
t  :r:Calc         200
*
**********************************************************
c   Initialisierung der Anwendung abgeschlossen !
**********************************************************
```

Die Bedeutung der einzelnen Konfigurationskommandos ist in Anhang A beschrieben. Die hier abgedruckte Konfigurationsdatei ist auf einem Rechner mit iRMX-Betriebssystem lauffähig (zu erkennen an dem logischen Laufwerksnamen „:r:“ und den Prioritäten).

Die *Calc*-Task wurde von dem Rechner *Host* auf den Rechner *Trabant* ausgelagert. Der erste Abschnitt (Punkte 1 und 2) der Konfigurationsdatei werden auf dem Rechner *Host* ausgeführt. Die Schnittstelle *InTask_Out* wird hierbei als Remote-Schnittstelle konfiguriert. Ihre Daten werden durch den MoMo-Handler an den Rechner *Trabant* weitergeleitet. Auf dem *Trabant* Rechner wird der dritte Teil der Konfigurationsdatei ausgeführt. Hier wird die *Calc*-Task gestartet und die Ausgabe dieser Task (*Calc_Out*) als Remote-Schnittstelle mit Ziel *Host* wieder an den *Host*-Rechner zurückgegeben.

Anhang D

Beispiel Pattern (einfach)

In diesem Kapitel werden die Pattern für die Calc-Task aus Kapitel 2.2.13 (siehe auch Abb. 2.7) vorgestellt.

```
/****** PROCEDURE *********************************/
unsigned short    Parameter_Input ()
/*****************************************************
 * @procname  Parameter_Input
 *
 * @return
 *        E_OK     if no errror detected
 *        E_EXIST  if fopen failed
*****************************************************/
{
  static FILE    *file;
  char           nModName[_MAX_STRING];
  char           dummy[_MAX_STRING];
  char           value_str[255];
  unsigned short exc;

  strcpy((char*)&nModName,":dat:");
  strcat((char*)&nModName,(char*)modul_name);
  strcat((char*)&nModName,".dat");
  check (xTKs,NO_ERR_CHECK,R_JEDE, _PARAINP+1,
        "Input from :dat:Appl.dat");
  /* @if (fopen failed) */
  if((file = fopen(nModName, " r")) == NULL)
    {
      sprintf(dummy, " Error: fopen(%s, r) failed",
              nModName);
      check (xTKs, ERR_REPORT,R_JEDE, _PARAINP+2, dummy);
      return (E_EXIST);
    }
  /* @end if (fopen failed) */
  /*BEGIN APPLCODE*/
  exc = get_int(file, &add_konst);    if (exc != 1)
```

Einlesen der Konstante, welche auf Input addiert wird.

```
    {
      fclose( file );
      check (xTKs, ERR_REPORT, R_JEDE, 3,
             " Add: Error reading add_konst ");
      exit(EXIT_FAILURE);
    }
  sprintf(value_str, "%d", add_konst);
  value_log (" Add add_konst: ", V_UNS_SHORT,
             (unsigned short)1, value_str);
  /*END APPLCODE*/
  fclose (file);
  check ( xTKs, E_OK, R_JEDE, _PARAINP+9, " Close file");
  return (E_OK);
}


/****** PROCEDURE ********************************/
unsigned short    ApplTsk_Activ_Exec()
/*****************************************************
 * @procname  ApplTsk_Activ_Exec
 *
 * @syntax
 *** End of Syntax
 *
 * @desc
 *** End of Description
 *
 * @param
 *** End of Parameterlist
 *
 * @return
 *    E_OK     if   no errror detected
 *             else Error_code of the Appl-Procedure
 *****************************************************/
{
  /*BEGIN APPLCODE*/
  xTKs->TG.exc = Ss_ReceiveDat ( hSsIn,                    Empfangen
                                 (char *)&sInBlock,        der Eingabe-
                                 NO_WAIT,                  daten aus
                                 (unsigned short)          der Input-
                                   sizeof(sInBlock) );     mailbox
  check ( xTKs, E_TIME, R_EREIGN, (int)361,                hSsIn.
          " Receive Msg von SsIn");
  if (xTKs->TG.exc==E_OK)
    {
      input = atoi(sInBlock.string);
      output = input + ADD_CONST;                          Aufaddieren
      printf (" Add-Task:   %d + %d  = %d\n",input,        der Kon-
               ADD_CONST,output );                         stante.
      sprintf (sOutBlock.string, "%d", output);
```

```
        sOutBlock.MH.MSG_Length = strlen(sOutBlock.string)
                                  + HEADER_LENGTH;
        xTKs->TG.exc = Ss_SendDat (hSsOut,                    Ergebnis
                                   (char *)&sOutBlock,        an die Out-
                                   NO_WAIT);                  putmailbox
        check ( xTKs, E_TIME, R_EREIGN, (int)361,             hSsOut
                " Send Msg an SsOut");                        schicken.
      }
    /*END APPLCODE*/

    return (xTKs->TG.exc);
  }
```

Die Input- und Outputmailboxen SsIn und SsOut werden vom Taskframe zur Verfügung gestellt. Sie werden über die IDs *hSsIn* und *hSsOut* angesprochen. Nur die Zeilen, die zwischen */*BEGIN APPLCODE*/* und */*END APPLCODE*/* stehen, wurden für die Anwendung in das Pattern eingefügt.

Anhang E

Beispiel Pattern Regelungszyklus

Die Anwendungstasks basieren auf einem von MoMo zur Verfügung gestellten Framework (Kap. 2.2.15). Nachfolgend sind die drei Pattern für den Aktiv-Zustand (zum Taskautomaten siehe Kap. 2.2.3) für einen allgemeinen Regelungszyklus für Roboter angeführt. Der Roboter-Zyklus ist eingebettet in das Framework, d.h. nur die Zeilen zwischen */*BEGIN APPLCODE*/* und */*END APPLCODE*/* wurden in das Pattern eingefügt. Auf diesem Zyklus basieren in ARTEMIS sowohl das Endoskopführungssystem Robox, wie auch die Steuerung des Slaves Tiska.

Da im Aktiv-Zustand des Roboter-Zykluses der Roboter verfahren wird (Servos aktiv), ist dieser Zustand kritisch. Hier dürfen keine Unterbrechungen oder Schwankungen der Zykluszeit auftreten. Deshalb wird in der *Init*-Prozedur Einträge in das Logfile und Unterbrechungen durch den Debug-Client unterbunden. Außerdem wird die Hardware aktiviert.

```
/****** PROCEDURE ***********************************************/
unsigned short    ApplTsk_Activ_Init()
/*************************************************************
 * @procname  ApplTsk_Activ_Init
 *
 * @desc
 * Diese Prozedur aktiviert das Geraet fuer den Betrieb.
 *** End of Description
 *
 * @returnq
 *        E_OK     if   no errror detected
 *                  else Error_code of the Appl-Procedure
 *************************************************************/
{
  char    checkstr[_MAX_STRING];
```

```
  /*BEGIN APPLCODE*/
  xTKs->TG.Activ_LP = 0;
  /** 1. Zyklusinterval fuer Activ-State festlegen */
  xTKs->TG.exc = Time_SetCycleTime(xTKs);
  sprintf(checkstr, "AZ_Init: CycleTime = %lu",
          xTKs->Appl.Cycle.Zykluszeit);
  check (xTKs, E_OK, R_JEDE, 1, checkstr);

  /** 2. Unterbrechung durch Logfile-Eintraege verhindern */
  xTKs->TG.exc = DEBUG_SetLogMode(R_FEHLER);

  /** 3. Unterbrechung durch den Application-Debugger verhindern */
  xTKs->TG.exc = DEBUG_AtomicAreaBegin();

  /** 4. Bremsen oeffnen */
  xTKs->TG.exc = DIGIO_AlleBremsenOeffnen();
  check (xTKs, E_OK, R_FEHLER, 1, "Bremsen oeffnen");

  /** 5. Servos aktivieren */
  xTKs->TG.exc = DIGIO_AlleServosEnable();
  check (xTKs, E_OK, R_FEHLER, 2, "Servos aktivieren");
  /*END APPLCODE*/

  return (xTKs->TG.exc);
}
```

Keine Ausgaben mehr in Logfile

Keine Unterbrechung möglich

Die *Exec*-Prozedur realisiert den Zyklus. Hier werden die für alle Roboter nötigen Funktionen (z.B. Position einlesen, Regler rechnen) durchgeführt

```
/****** PROCEDURE ***********************************************/
unsigned short    ApplTsk_Activ_Exec()
/*************************************************************
 * @procname  ApplTsk_Activ_Exec
 *
 * @desc
 * Diese Prozedur stellt die Implementierung des
 * aktiven Regelzykluses dar.
 *** End of Description
 *
 * @return
 *        E_OK     if   no errror detected
 *                 else Error_code of the Appl-Procedure
 *************************************************************/
{
  /*BEGIN APPLCODE*/
  /** 1. Schleifen timing */
  xTKs->TG.exc = Time_CycleTiming(xTKs);
  check (xTKs, E_OK, R_FEHLER, 1, "AZ_exec: Zyklus Start");

  /** 2. Stellwert-Ausgabe */
  xTKs->TG.exc = ANAOUT_PutStellwert (xTKs);
```

Am Zyklusanfang Stellwerte ausgeben, um Schwankungen zu minimieren

```
check (xTKs, E_OK, R_FEHLER, 2,
       "AZ_exec: Stellwert ausgegeben");

/** 3. Zyklusnummer erhoehen */
xTKs->Appl.Cycle.lZyklusNr++;
check (xTKs, E_OK, R_FEHLER, 3, "AZ_exec: Zyklusnr erhoeht");

/** 4.   System-Sicherheits-Zustand pruefen */
xTKs->TG.exc = DIGIO_GetK2Status(xTKs);
if (!xTKs->Appl.Cycle.IsK2)
  return(E_NOTK2);
check (xTKs, E_OK, R_FEHLER, 4, "AZ_exec: K2 gedrueckt");

/** 5.   Watchdog triggern */
xTKs->TG.exc = DIGIO_TriggerWatchdog ();
check (xTKs, E_OK, R_FEHLER, 5, "AZ_exec: Watchdog getriggert");

/** 6.   Einlesen der Ist-Position (q_roh) anstossen */
if (xTKs->TG.exc==E_OK) {
  xTKs->TG.exc = WINKEL_RequestPosition (xTKs);
  check (xTKs, E_OK, R_FEHLER, 6, "AZ_exec: Lies Position");
}

/** 7.   Ist-Position berechnen und im TCS ablegen */
if (xTKs->TG.exc==E_OK) {
  xTKs->TG.exc = WINKEL_GetPosition (xTKs);
  check (xTKs, E_OK, R_FEHLER, 7, "AZ_exec: Schreibe Position");
}

/** 8.   Ist-Position pruefen */
if (xTKs->TG.exc==E_OK) {
  xTKs->TG.exc = WINKEL_CheckPosition (xTKs);
  check (xTKs, E_OK, R_FEHLER, 8, "AZ_exec: Check Position");
}

/** 10.   Geschwindigkeit einlesen und/oder berechnen */
if (xTKs->TG.exc==E_OK) {
  xTKs->TG.exc = WINKEL_GetVelocity (xTKs);
  check (xTKs, E_OK, R_FEHLER, 10,
         "AZ_exec: Hole_IstGeschwindigkeit");
}

/** 11.   Geschwindigkeit pruefen */
if (xTKs->TG.exc==E_OK) {
  xTKs->TG.exc = WINKEL_CheckVelocity (xTKs);
  check (xTKs, E_OK, R_FEHLER, 11,
         "AZ_exec: Check_IstGeschwindigkeit");
}

/** 15.   Sollwerte holen */
if (xTKs->TG.exc==E_OK) {
  xTKs->TG.exc = SOLL_GetSollDat (xTKs, hSsIn);
  check (xTKs, E_OK, R_FEHLER, 15, "AZ_exec: Soll-Winkel geholt");
}

/** 16.   Sollwerte pruefen */
if (xTKs->TG.exc==E_OK) {
  xTKs->TG.exc = SOLL_CheckSollDat (xTKs);
  check (xTKs, E_OK, R_FEHLER, 16, " AZ_exec: Soll-Winkel
```

```
geprueft");
  }

  /** 17.   Schleppabstand pruefen */
  if (xTKs->TG.exc==E_OK) {
    xTKs->TG.exc = SOLL_CheckSchleppabstand (xTKs);
    check (xTKs, E_OK, R_FEHLER, 17,
           "AZ_exec: Schleppabstand geprueft");
  }

  /** 18.   Regler rechnen */
  if (xTKs->TG.exc==E_OK) {
    xTKs->TG.exc = REGLER_Calculation(xTKs);
    check (xTKs, E_OK, R_FEHLER, 18, " AZ_exec: Stellwerte
berechnet");
  }

  /** 19.   Werte an IstDat weiterleiten */
  if (xTKs->TG.exc==E_OK) {
    xTKs->TG.exc = IST_PutPosition (xTKs,hSsOut);
    check (xTKs, E_OK, R_FEHLER, 19,
           "AZ_exec: Istposition verschickt");
  }
/*END APPLCODE*/

return (xTKs->TG.exc);
}
```

In der *Fin*-Prozedur wird beim Verlassen des Aktiv-Zustands der Roboter in einen sicheren Zustand gebracht und Logfile-Einträge und Unterbrechungen durch den Debug-Client wieder ermöglicht.

```
/****** PROCEDURE ***********************************************/
unsigned short    ApplTsk_Activ_Fin()
/*************************************************************
 * @procname  ApplTsk_Activ_Fin
 *
 * @desc
 * Diese Prozedur sorgt nach dem Beenden des aktiven
 * Regelzykluses dafuer, dass das Geraet in einen
 * sicheren Zustand versetzt wird.
 *** End of Description
 *
 * @return
 *        E_OK     if   no errror detected
 *                  else Error_code of the Appl-Procedure
 *************************************************************/
{
  int            i;
  char           value_str[255];

  /*BEGIN APPLCODE*/
  /** Sicherheit herstellen  */

  /** 1. Bremsen schliessen */
  xTKs->TG.exc = DIGIO_BremsenZu();

  /** 2. Loggfile-Eintraege wieder zulassen */
  xTKs->TG.exc = DEBUG_SetLogMode(R_JEDE);
```

Logfile-Einträge wieder ermöglichen

```
  check (xTKs, E_OK, R_JEDE, 1, "Bremsen schliessen");

  /** 3. Servos sperren */
  xTKs->TG.exc = DIGIO_ServosDisable();
  check (xTKs, E_OK, R_JEDE, 2, "Servos sperren");

  /** 4. Stellstrom zuruecksetzen */
  xTKs->TG.exc = ANAOUT_Reset (xTKs);

  for ( i=0; i<3; i++ ) {
    value_str[0] = '\0';
    sprintf(&value_str[strlen(value_str)],
            "%f ",xTKs->Appl.Cycle.q[i]);
    sprintf(&value_str[strlen(value_str)],
            "%f ",xTKs->Appl.Cycle.qSoll[i]);
    value_log ("Cycle FIN : ", V_FLOAT, (unsigned short)2, value_str);
  }

  /** 5. Unterbrechung durch den Appl.-Debugger wieder ermoeglichen
*/
  xTKs->TG.exc = DEBUG_AtomicAreaEnd();
  /*END APPLCODE*/

  return (xTKs->TG.exc);
}
```

Debugging wieder ermöglichen

Anhang F

Beispiel eines Logfiles

In dem Logfile werden wichtige Ereignisse, Parameter und Fehler dokumentiert. Dies wird mit den in Kapitel 2.2.14 beschriebenen Funktionen *check()* und *value_log()* realisiert. Nachfolgend ein Logfile nach dem Start einer kompletten MoMo-Application-Unit.

In der ersten Spalte ist die Statusnummer zu sehen, die die Ausgabe erzeugt hat. Diese ist für jede Task eindeutig, so daß man den Ablauf innerhalb einer Task rekonstruieren kann. In der zweiten Spalte wird der Ergebniswert des vorherigen Funktionsaufrufes notiert. Dann folgt der Taskname und ein erklärender Text. In der letzten Spalte steht die Anzahl der Systemticks, die seit dem Start der Task vergangen sind. D.h. dieser Wert ist taskbezogen und ermöglicht den zeitlichen Ablauf zu rekonstruieren. In Zukunft ist geplant, diese Zeit auf Systemzeit umzustellen, so daß auch die zeitlichen Beziehungen zwischen den Tasks ersichtlich wird.

Eine Ausnahme bilden die Ausgaben mit *value_log()*. Hier wird ein erklärender Text und dann eine beliebige Anzahl von Integer- oder Float-Werten ausgegeben. So können wichtige Start- oder Zwischenwerte dokumentiert werden.

```
Logfile des Versuchs vom :    Wed Aug 06 1997   Zeit : 07:38:52
Status exc    Taskname   Text                                    Ticks
-------------------------------------------------------------------------
   22 OK      $Log       Logger-Schnittstelle intern ange          10
   99 OK      $Log       Initialisierung abgeschl.                 10
    6 OK      MoMoH      Initialisierung TKS                       14
    1 OK      MoMoH      Read Param-File                           17
    4 OK      MoMoH      MBx InMoMoh angelegt                      18
    5 OK      MoMoH      MBx InMoMoH gefunden                      18
    6 OK      MoMoH      MBx InMMI angelegt                        19
    7 OK      MoMoH      MBx InMMI gefunden                        19
    8 OK      MoMoH      MBx InCCH angelegt                        20
    9 OK      MoMoH      MBx InCCH gefunden                        20
    1 OK      CCH_$MSrv  TCS initialisiert                          1
```

Start MoMo-Logger

Start MoMo-Handler

MoMoH startet einen CCH um Kommandos zu empfangen

```
    3 OK      CCH_$MSrv  Debug = OFF                          1
    4 OK      CCH_$MSrv  Protokoll        = TCP              1
    5 OK      CCH_$MSrv  TCPKomPar.SMBx   = $InCCH           1
    6 OK      CCH_$MSrv  TCPKomPar.RMBx   = $InMoMoH         1
    7 OK      CCH_$MSrv  Servername       = any              1
    8 OK      CCH_$MSrv  Servicename      = momoh            1
    9 OK      CCH_$MSrv  TCPKomPar.State  = p                1
   10 OK      CCH_$MSrv  TCPKomPar.Urgent = 1                1
   13 OK      CCH_$MSrv  Empfangbare MSG-Laenge: 300         1
   14 OK      CCH_$MSrv  Initialisierung abgeschl.           1
   34 OK      CCH_$MSrv  TCP - Connection (passiv)           1
   10 OK      MoMoH      LoadKah MSrv                       27
    1 OK      Control    Initialisierung TKS                 2
    6 OK      Control    Lookup SsControl                    2
   14 OK      Control    Lookup MBxMMi                       6
   99 OK      Control    Initialisierung abgeschl.           9
   99 OK      Control    Control_Automat                    10
  100 OK      Control    INIT_STATE                         11
10100 OK      Control    Input from :dat:Kin_Def.dat        12
    4 OK      Control    anzfhg                             14
   55 OK      Control    unit_id                            14
   50 OK      Control    unit_type                          14
10199 OK      Control    Close file                         14
10200 OK      Control    Input from :dat:Control.dat        15
    5 OK      Control    atz_aeh                            16
10299 OK      Control    Close file                         16
10300 OK      Control    Input from :dat:ablage.dat         17
    2 OK      Control    Abl_periode                        33
10304 OK      Control    :mes:tiska.mes                     33
10399 OK      Control    Close file                         33
  102 OK      Control    Einlesen d Parameter abgeschl.     35
  104 OK      Control    BZ: Initialisierung abgeschlosse   35
  190 OK      Control    Init-->BASIC_STATE                 35
  200 OK      Control    BASIC_STATE                        40
  204 OK      Control    BZ: Initialisierung abgeschlosse   40
    0 OK      Control    BZ: Loc-Message erhalten           40
 3000 OK      Control    MoMo-Mel_Init-Task                 40
    1 OK      Control    ApplCyc_TCS                        40
 3004 OK      Control    Send(Kom_Basic)                    40
 3009 OK      Control    MoMo-Mel_Init-Ende                 40
    0 OK      Control    BZ: Loc-Message erhalten           40
 3000 OK      Control    MoMo-Mel_Init-Task                 40
 3009 OK      Control    MoMo-Mel_Init-Ende                 40
30001 OK      SetPoint   Rahmen-Init TCS                     0
30002 OK      SetPoint   Lookup TSs_SetPoint (own Ss)        0
30003 OK      SetPoint   Lookup TSs_Control                  0
30005 OK      SetPoint   DataIn Interface found              0
30007 OK      SetPoint   DataOut Interface found             0
30009 OK      SetPoint   Input from :dat:Appl.dat            0
30601 OK      SetPoint   fopen(:dat:Appl.dat) OK             1
30602 OK      SetPoint   Input: unit_id = 10                 1
30606 OK      SetPoint   Input: LP = 10  10  1               1
30010 OK      SetPoint   File closed                         2
30011 OK      SetPoint   Anmeldungs-Kom_Init                 2
30048 OK      SetPoint   Frame Initialisierung abgeschl.     2
30091 OK      SetPoint   --> Init_State                      2
30100 OK      SetPoint   Init_State                          2
31100 OK      SetPoint   Call of ApplTsk_Init                2
```

Start der MoMo-Control-Unit

Einlesen der Parameter

MoMo-CU wechselt von Init in Basic State

Start einer Anwendungstask

Einlesen der Parameter

SetPoint meldet sich bei der CU an

```
10001 OK      SetPoint   Input from :dat:Appl.dat                  2
              SetPoint Looptime: 10  10  1
              WINKEL anzfhg: 4
              SetPoint MaxSpeed        : 0.100 0.100 20.000 0.100
              SetPoint MaxAcceleration : 0.100 0.100 20.000 0.100
10009 OK      SetPoint   Close file                               13
30104 OK      SetPoint   IZ: Initialisierung abgeschlosse         13
    0 OK      Control    BZ: Loc-Message erhalten                 62
 3000 OK      Control    MoMo-Mel_Init-Task                       62
    3 OK      Control    SetPoint_TCS                             62
 3004 OK      Control    Send(Kom_Basic)                          62
 3009 OK      Control    MoMo-Mel_Init-Ende                       62
30106 OK      SetPoint   IZ: Control-Msg received                 13
30114 OK      SetPoint   cmd: Kom_Basic                           13
30092 OK      SetPoint   --> Basic_State                          13
30200 OK      SetPoint   Basic_State                              13
30202 OK      SetPoint   BZ: State ack to Cntr 3                  13
31200 OK      SetPoint   Call of ApplTsk_Basic_Init               13
30204 OK      SetPoint   BZ: Initialisierung abgeschlosse         13
    3 OK      Control    BZ: Loc-Message erhalten                 76
 3110 OK      Control    MoMo-Mel_BasicZust                       76
 3119 OK      Control    MoMo-Mel_BasicZust-Ende                  76
    7 OK      $MMi       Initialisierung TKS                       1
   48 OK      $MMi       Initialisierung abgeschl.                 2
 2117 TIME    $MMi       MMi-Kom: Ready                          315
 2222 OK      $MMi       MMi-Kommando an MoMoH                   317
   11 OK      MoMoH      Rec at InMoMoH-MBx                     2050
    2 OK      $MMi       MoMoH_Kommando erhalten                 317
    0 OK      Control    BZ: Loc-Message erhalten                859
 3140 OK      Control    MoMo-Kom Ready                           859
 3142 OK      Control    Send(Kom_Ready)                          859
 3143 OK      Control    Direkter Zustandswechsel                 859
 3149 OK      Control    MoMo-Kom Ready-Ende                      859
  206 OK      Control    BZ: Anw_Zust_Close(BASIC_STATE)          859
30206 OK      SetPoint   BZ: Control_Kommando erhalten            808
30210 OK      SetPoint   Basic-->Ready_State                      808
33200 OK      SetPoint   Call of ApplTsk_Basic_Fin                808
30093 OK      SetPoint   --> Ready_State                          808
30300 OK      SetPoint   Ready_State                              808
30302 OK      SetPoint   RZ: State ack to Cntr 3                  808
31300 OK      SetPoint   Call of ApplTsk_Ready_Init               808
    1 OK      SetPoint   Winkel_ActPos: hCycle geholt             808
    2 OK      SetPoint   Ready_Init: IstPosition geholt           808
              SetPoint: Ist-Winkel = 0.000000 0.000000 0.000000
30304 OK      SetPoint   RZ: Initialisierung abgeschlosse         808
  300 OK      Control    Ready_State                              866
  304 OK      Control    RZ: Initialisierung abgeschlosse         867
    1 OK      Control    RZ: Loc-Message erhalten                 867
 3150 OK      Control    MoMo-Mel ReadyZust                       867
 3159 OK      Control    MoMo-Mel_ReadyZust-Ende                  867
 2118 OK      $MMi       MMi-Kom: Activ                           538
 2222 OK      $MMi       MMi-Kommando an MoMoH                    540
   11 OK      MoMoH      Rec at InMoMoH-MBx                      2273
    2 OK      $MMi       MoMoH_Kommando erhalten                  540
    0 OK      Control    RZ: Loc-Message erhalten                1082
 3206 OK      Control    Direkter Zustandswechsel                1082
  307 OK      Control    RZ: Kom_Activ                           1082
30306 OK      SetPoint   RZ: Control_Kommando erhalten           1024
```

CU schickt SetPoint Kom_Basic

SetPoint im Basic State

Start lokales MMI

Kom_Ready von MMI erhalten

Kom_Ready an SetPoint

Kom_Ready von CU erhalten

Kom_Activ vom MMI erhalten

Kom_Active von CU erhalten

```
30094 OK      SetPoint   --> Activ_State                      1024
  400 OK      Control    Activ_State                          1089
  404 OK      Control    AZ: Initialisierung abgeschlosse     1089
    1 OK      Control    AZ: Loc-Message erhalten             1089
   11 TIME    SetPoint   Cycle nimmt Sollwerte nicht ab       1054
33400 OK      SetPoint   Call of ApplTsk_Activ_Fin            1055
30093 OK      SetPoint   --> Ready_State                      1055
30300 OK      SetPoint   Ready_State                          1055
30302 OK      SetPoint   RZ: State ack to CU 3                1055
31300 OK      SetPoint   Call of ApplTsk_Ready_Init           1055
```

Fehler in Anwendungstask

Übergang in Ready State

Info an CU

Literaturverzeichnis

[Arnold & Gosling, 96] K. Arnold, J. Gosling *The Java Programming Language*; Addision-Wesley, 1997

[Berg, 97] K. Berg *Component-Based Software Development: No Silver Bullet, But More Than a Vision*; Object Expert March/April 97, SIGS Publications, 1997, 49-51

[Blair & Lea, 92] G. S. Blair, R. Lea *The Impact of Distribution on the Object-Oriented Approach to Software Development*; IEE/BCS Software Engineering Journal March 92, Vol. 7, Number 2, 1992

[Booch, 94] G. Booch *Designing an Application Framework*; Dr.Dobb's Journal 19, No.2, Feb. 1994, 24

[Breitwieser & Weber, 96] H. Breitwieser, W. Weber *MONSUN A Distributed Manipulator Control System Utilizing Network Technology*; Kopacek,P. (Ed.) Human-Oriented Design of Advanced Robotics Systems, Pergamon, 1996, 159-167

[Cotter & Potel, 95] S. Cotter, M. Potel *Inside Taligent Technology*; Addison-Wesley, 1995

[Erhardt, 97] T. Erhardt *Service-MMI-Client in Java für eine verteilte Realzeitanwendung*; Diplomarbeit am Institut für Angewandte Informatik, Forschungszentrum Karlsruhe und Berufsakademie Karlsruhe, 1997

[Gallmeister, 95] B.O. Gallmeister *POSIX.4: Programming for the Real World*; O'Reilly & Associates, 1995

[Gamma *et al.*, 94] E. Gamma, R. Helm, R. Johnson, J. Vlissides *Design Patterns*; Addision-Wesley, 1994

[Garlan *et al.*, 95] D. Garlan, R. Allen, J. Ockerbloom *Architectural Mismatch: Why Reuse Is So Hard*; IEEE Software May, 1995, 17-26

[Habert *et al.*, 90] S. Habert, L. Mosseri, V. Abrosimov *COOL: Kernel Support for Object-Oriented Environments*; Proceedings of OOPSLA'90, Ottawa, Canada, ACM Press, 1990

[Höfele, 97] C. Höfele *Debugging-MMI-Client in Java für eine verteilte Realzeitanwendung*; Diplomarbeit am Institut für Angewandte Informatik, Forschungszentrum Karlsruhe und Berufsakademie Karlsruhe, 1997

[Holler, 95] E. Holler *Operationssysteme für die minimal invasive Chirurgie*; Forschungszentrum Karlsruhe Wissenschaftliche Berichte FZKA 5670, 1995, 115-117

[InfoDuden, 88] *Duden Informatik*; Dudenverlag, 1988

[Javateam, 97] *Java White Papers*;

`http://java.sun.com/docs/white/index.html`, 1997

[Kaufmann, 97] H. Kaufmann *Erstellung eines Servers für MMI-Anwendungen in Java*; Studienarbeit am Institut für Angewandte Informatik, Forschungszentrum Karlsruhe und Institut für Biomedizinische Technik, Universität Karlsruhe, 1997

[Kühnapfel, 90] U. Kühnapfel *KISMET - 3D-Grafik zur Planung, Programmierung und Überwachung von Telerobotik-Applikationen*; VDI-Berichte Nr. 861.3, VDI-Verlag Düsseldorf, 1990, 71-86

[Lea & Weightman, 91] R. Lea, J. Weightman *COOL: An Object Support Environment Co-existing with Unix*; Proceedings of the AFUU'91 Unix Convention, Paris, France, March 1991

[Li, 1995] G. Li *Some Engineering Aspects of Real-Time*; Technical Report, Architecture Projects Management, APM.1222.02, 1995

[Sommerville, 95] I. Sommerville *Software Engineering*; Addision Wesley, 1995

[Taligent, 97] *Leveraging Object-Oriented Frameworks — A Technology Primer from Taligent*;

`http://www.taligent.com/`, 1997

[Tanenbaum, 95] A. S. Tanenbaum *Distributed Operating Systems*; Prentice Hall, 1995

[Tanenbaum, 97] A. S. Tanenbaum *Computernetzwerke*; Prentice Hall, 1997

[Vickery, 1993] C. Vickery *Real-Time and Systems Programming for PCs*; McGraw-Hill, 1993

[Voges *et al.*, 95] U. Voges, P. Dautzenberg, U. Kühnapfel, B. Neisius, M. Schmitt, R. Trapp, T. Vollmer *Experimenteller Telemanipulator für die minimal invasive Chirurgie*; Forschungszentrum Karlsruhe Wissenschaftliche Berichte FZKA 5670, 1995, 106-111

[Ward & Mellor, 86] P. T. Ward, S. J. Mellor *Structured Development for Real-Time Systems*; 3 volumes, Yourdon Press, 1986