NTB Interstaatliche Hochschule für Technik Buchs

Konzept und Implementierung eines Interfaces zu EEROS-Komponenten

MSE Vertiefungsprojekt 2

Herbstsemester 2014

Student: Adam Bajric
Referent: Prof. Dr. Andreas Zogg

Inhaltsverzeichnis

1. Einblick in EEROS

Bei der Programmierung industrieller Roboter müssen oft wiederkehrende Probleme gelöst werden. Jeder Roboter braucht eine Regelung, die eine korrekte Bewegung gewährleistet, ein Sicherheitssystem, das das System überwacht, kritische Fehler vermeidet und ungültige Zustände verhindert und eine Ablaufsteuerung, welche vordefinierte Programme ausführt und somit die eigentliche Aufgabe löst. Ausserdem muss die Software immer auf die Hardware zugreifen können. EEROS ist ein objektorientiertes Software-Framework in C++, das einem das Programmieren solcher Roboter erleichtert, ohne die Kontrolle über das System einzuschränken.

In Abbildung 1.1 sind die vier Subsysteme von EEROS dargestellt.

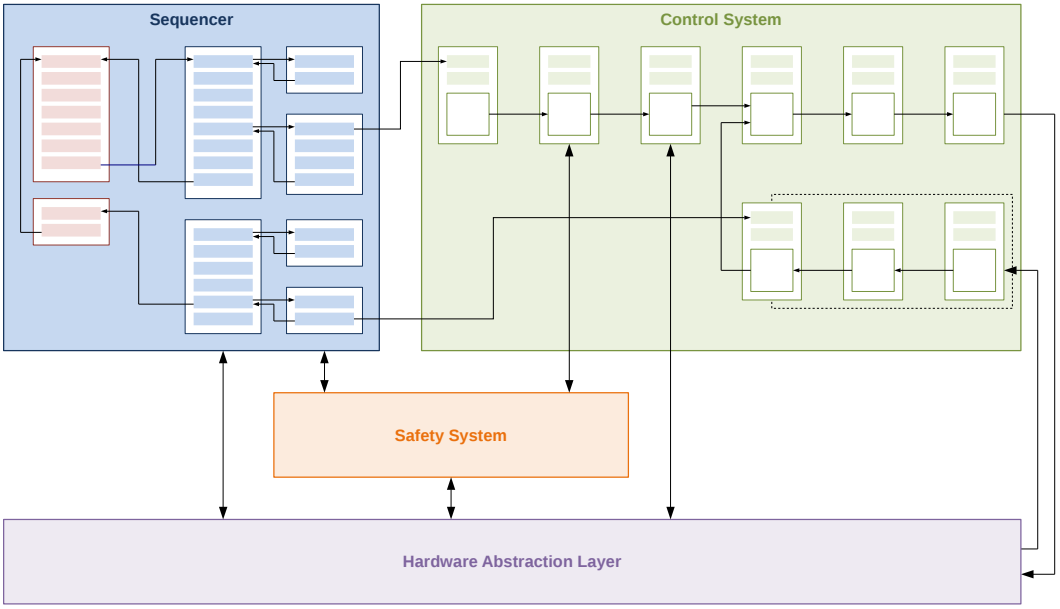


Abbildung 1.1.: EEROS Subsysteme

1.1. Control System

Das Control System kümmert sich um die Regelung des Roboters. Die Reglerstruktur wird, ähnlich wie in Simulink, aus den drei Komponenten *Funktionsblock*, *Signal* und *Zeitdomäne* zusammengesetzt. Abbildung 1.2 zeigt ein Beispiel eines Control Systems.

Die *Funktionsblöcke* generieren oder verarbeiten *Signale*. Die Outputsignale eines Blocks können als Inputsignale eines anderen Blocks verwendet werden. Beispiele dieser Blöcke sind Summierer, Integrierer, PID-Regler, usw., aber auch kompliziertere Funktionsblöcke, wie zum Beispiel ein Bahnplaner, können umgesetzt werden.

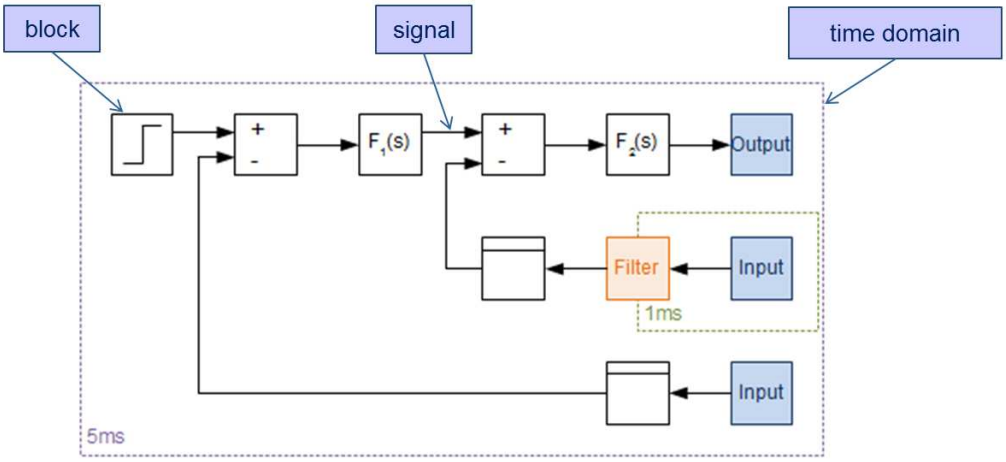


Abbildung 1.2.: Beispiel eines EEROS Control Systems

Alle Blöcke in EEROS erweitern die abstrakte Basis-Klasse `Block` und implementieren die Methode `run`. In dieser Methode wird die eigentliche Funktion des Blocks implementiert. Grundsätzlich sieht eine solche Implementierung wie folgt aus: die Eingänge werden gelesen, Daten werden verarbeitet und die Ausgänge werden gesetzt.

Die Funktionsblöcke können aber um weitere Methoden ergänzt sein. Abbildung 1.3 zeigt einen Switch-Block. Dieser Block leitet den Signalwert eines bestimmten Inputs zum Output weiter. Mit der Methode `switchToInput` kann der Input ausgewählt werden, welcher weitergeleitet wird.

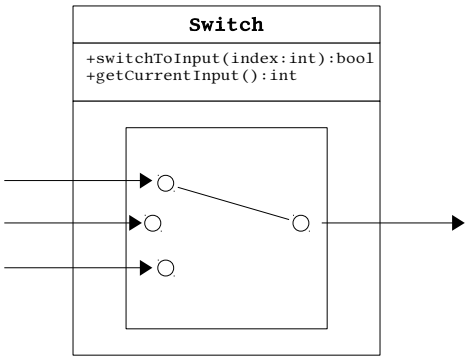


Abbildung 1.3.: EEROS Switch Block

Die *Timedomains* geben den Takt vor. Alle Blöcke, die sich innerhalb einer Timedomain befinden, werden in zeitlich äquidistanten Abständen ausgeführt. Dabei ist die Reihenfolge der Ausführung sehr wichtig. Signale werden von einer Timedomain in eine andere durch Übergangsblöcke übergeben, welche sich in beiden Timedomänen befinden und deshalb auch zwei `run` Methoden haben. Diese Blöcke filtern oder interpolieren die Signale. In Abbildung 1.2 gibt es zwei Timedomains mit einem Takt von 5 ms bzw. 1 ms.

1.2. Sequenzen

Die Abläufe des Roboters werden in sogenannten Sequenzen implementiert. Beispiele dafür sind einfache Dinge wie *geradeaus Fahren* und *rechts/links Drehen*. Sequenzen sind Folgen von Anweisungen, die, wie es die Bezeichnung Sequenz zum Ausdruck bringt, sequenziell ausgeführt werden müssen. Sequenzen können Untersequenzen aufrufen oder können selbst Teil von übergeordneten Sequenzen sein. So lassen sich komplexere Abläufe, wie das Zusammenbauen eines Autos, in einfachere unterteilen.

In Abbildung 1.4 wurden drei konkrete Sequenzen dargestellt. Wird die `GrabSequence` ausgeführt, schliesst der Roboter den Greifer, bei `MeasureDistanceSequence` wird eine Distanz gemessen und die `ParkSequence` platziert den Roboter auf einem numerierten Parkfeld.

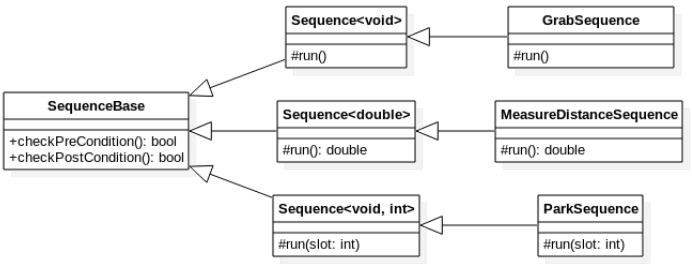


Abbildung 1.4.: Beispiel von EEROS-Sequenzen

Die Sequenzen werden vom *Sequencer* ausgeführt. Bevor eine Sequenz ausgeführt wird, wird über die Methode `checkPreCondition` überprüft, ob die Sequenz ausgeführt werden darf. Falls dies nicht der Fall ist, wird entweder gewartet oder es gibt eine Exception. Analog wird nach dem Beenden der Sequenz mit der Methode `checkPostCondition` geprüft, ob die Sequenz korrekt abgearbeitet wurde und ob mit der nächsten Sequenz weitergemacht werden darf.

1.3. Safety System

Bestimmte Zustände dürfen nicht eintreten. Beispielsweise darf der Roboter nicht aus seinem sicheren Arbeitsraum ausbrechen oder mit sich selbst kolidieren. Das Safety System überwacht den Betrieb des Roboters und verhindert solche Zustände. Wird der Bereich der akzeptierten Zustände verlassen, sorgt das Safety System dafür, dass eine Massnahme, beispielsweise Strom ausschalten und Bremsen anziehen, ergriffen wird. Das Safety System agiert als unabhängige Oberaufsicht.

1.4. Hardware Abstraction Layer

Damit der Hardwarezugriff vereinfacht und vereinheitlicht ist, gibt es in EEROS einen Hardware Abstraction Layer. Ports zur Hardware werden in EEROS als Signale dargestellt. Für die meisten Anwendungen (Encoder, PWM, GPIO, usw.) reichen normale Signale, die `double` oder `bool` Werte repräsentieren.

2. Ausgangslage und Aufgabenstellung

EEROS richtet sich primär an zwei Benutzergruppen: Hersteller, welche ihre Roboter mit einer gewissen Grundfunktionalität anbieten wollen, und die Benutzer des Roboters, welche ein Problem mit dem erworbenen Roboter lösen möchten. Die Anwendungsfälle der beiden Gruppen sind gegenseitig ergänzend. Die Hersteller müssen die Funktionstüchtigkeit des Roboters bereitstellen, sie können aber nicht ein spezifisches Programm für jede Anwendung schreiben, was die Aufgabe der Benutzer ist. Und diese hingegen wollen sich nicht mit den technischen Einzelheiten befassen, sondern konkrete Probleme lösen.

EEROS will beiden Anspruchsgruppen gerecht werden. Der aktuelle Entwicklungsstand lässt es aber noch nicht optimal zu, den Kern der Robotersteuerung vor fehlerhaften Anwendungen zu schützen. In dieser Arbeit soll deshalb ein Konzept entwickelt und implementiert werden, welches die Anwendungen vom Kernsystem abkoppelt und so das Kernsystem wirksam schützen kann.

Die Schnittstelle zum Kernsystem soll dabei so konzipiert werden, dass das Debugging und Logging des Kernsystems optimal unterstützt wird. Die aktuelle EEROS Version lässt kein Debugging im gewohnten Sinne mit kontrollierten Abläufen und dem Anzeigen und Verändern der internen Zustände zu. Die aktuelle Version erlaubt es bisher nur, Ausgaben auf eine Konsole oder in eine Datei zu schreiben, so dass diese dann anschliessend z.B. mit Matlab analysiert werden können. Der Debugger selbst ist allerdings nicht Teil der Aufgabenstellung.

Das Fernziel ist, einen Debugger zu haben mit dem man Abläufe kontrolliert ausführen und interne Zustände anzeigen und verändern kann, um somit einem Einblick in das Innere der Robotersteuerung zu erhalten. Gleichzeitig braucht es für jeden Roboter ein HMI (Human-Machine-Interface), mit welchem man Parameter ändern und vordefinierte Sequenzen ausführen kann.

3. Konzept

3.1. Rahmenbedingungen

Da EEROS Open Source ist, wurde das Framework hauptsächlich für Linux entwickelt, es sollte aber auch auf dem kommerziell erhältlichen Echtzeitbetriebssystem QNX lauffähig sein. QNX ist unter anderem in der Autobranche sehr verbreitet. Für die Konzeptionierung werden diese beiden Betriebssysteme berücksichtigt.

Im Vordergrund von EEROS steht die Realisierung von echtzeitfähigen Robotersteuerungen. Die Systemgrenze ist der Rechner. In dieser Arbeit wir die Ausweitung auf verteilte Systeme nicht betrachtet, weil dies zusätzliche Komplexität mit sich bringt. Die Echtzeitfähigkeit ist bei Netzwerken nicht üblich. Probleme, wie Verzögerungen oder sogar Ausfälle, müssen gelöst sein. Ausserdem ist ein Roboter von allen Leuten, die im selben Netzwerk sind, zugänglich. Es muss sichergestellt werden, dass Unbefugte keinen Schaden anrichten können.

3.2. Grundidee

Um den sicheren Betrieb der Kernkomponenten zu gewährleisten, werden potenziell unsichere Komponenten in einen eigenen Prozess isoliert. Verhält sich ein solcher Prozess fehlerhaft, so darf er das Kernsystem nicht beeinträchtigen. Einen gewissen Schutz bietet bereits das Betriebssystem, welches sicherstellt, dass Prozesse unabhängig von anderen, welche beispielsweise blockieren oder abstürzen, laufen können.

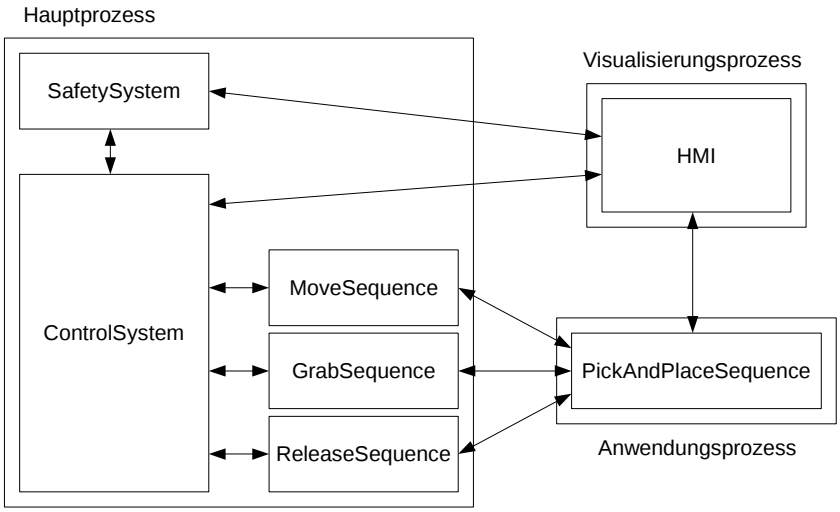


Abbildung 3.1.: Beispiel eines möglichen EEROS-Systems

Wie die Kernkomponenten in einem eigenen, von den Anwenderprozessen entkoppelten, Prozess abgetrennt werden können, zeigt das Beispiel in Abbildung 3.1. Auf einem Laufband kommen fertige Produkte, welche vom Roboter aufgenommen und in eine Schachtel auf einem anderen Laufband gelegt werden. Die Anwendung besteht aus drei Prozessen. Im Hauptprozess ist das Control- und Safety-System und einige einfache Grundsequenzen, die als sicher betrachtet werden. Die eigentliche Anwendung, die `PickAndPlaceSequence`, ist in einem separaten Prozess. Die Position der Produkte wird mit einer Kamera bestimmt, was relativ heikel ist. Falls diese Anwendung abstürzt, läuft der Hauptprozess weiter, welcher den Roboter in eine sichere Position bringen kann. Die Pick & Place Anwendung kann einfach neu gestartet werden, ohne dass sich der Roboter neu initialisieren oder sogar kalibrieren muss.

Das hat zur Folge, dass nicht alle Sequenzen direkt ausgeführt werden können und die Prozesse miteinander kommunizieren müssen.

3.3. Interaktionsmuster

Hauptprozess und Anwenderprozesse laufen nicht unabhängig voneinander, sie interagieren miteinander. Beispielsweise ruft die `PickAndPlaceSequence` im Anwenderprozess die `MoveSequence` im Hauptprozess auf. Oder vom Control System im Hauptprozess fliessen laufend Daten zum HMI im Visualisierungsprozess. Es gilt deshalb, geeignete Formen und Mittel der Interprozesskommunikation zu finden, so dass das Ziel der möglichst grossen Entkopplung erreicht werden kann.

Es gibt zwei Kommunikationsmuster, die unterstützt werden müssen, um die Anforderungen der EEROS-Komponenten an die Kommunikation zu erfüllen. Die Sequenzen müssen nur eine kleine Datenmenge übertragen werden. Jedoch gibt es einen Rückgabewert, der zurückgegeben werden muss, auch wenn er leer (void) ist, um die Ausführung zu quittieren.

Im Gegensatz dazu generieren die Funktionsblöcke des Control-Systems ständig eine relativ grosse Menge an Daten. Diese sollten bei Bedarf abrufbar sein, ohne das zeitliche Verhalten des Control-Systems zu beeinträchtigen. Ob jemand da ist, um die Daten zu lesen, oder ob die Daten lückenlos gelesen wurden, interessiert das Control System nicht.

Um diese Anforderungen zu erfüllen, werden zwei bekannte Kommunikationsmuster gewählt:

Request/Reply ist eines der grundlegendsten Kommunikationsmuster, um zwei Kommunikationspartner, im weiteren Knoten genannt, miteinander kommunizieren zu lassen. Sie wird vor allem bei Client/Server Anwendugen und Remote Pocedure Calls eingesetzt. Knoten A sendet eine Anfrage (Request) über einen Kanal an Knoten B. Dieser verarbeitet die Anfrage und sendet eine Antwort (Reply) über denselben Kanal zurück. Das ist eine sehr einfache Methode, welche aber viele Aufgaben lösen kann. Grundsätzlich ist die Kommunikation synchron und blockierend, was sehr gut zu unserer Problemstellung passt. Das Muster ähnelt einem Methodenaufruf.

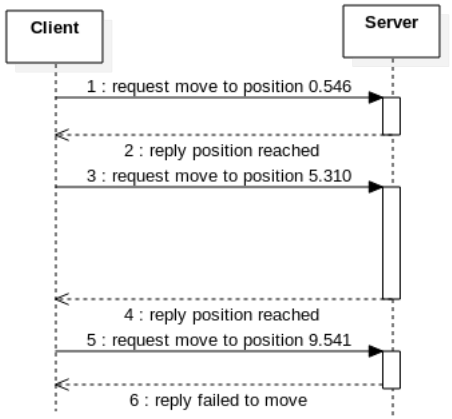


Abbildung 3.2.: Beispiel einer Request/Reply Kommunikation

Publish/Subscribe ist ein beliebtes Muster der Kommunikation, da es lose Koppelung erlaubt. Der Publisher, wie der Name schon sagt, veröffentlicht seine Daten einem Broker (oder in einen Puffer im Shared Memory), welcher dann die Subscriber über die Daten informiert. Natürlich können dabei Daten verloren gehen, wenn beispielsweise der Publisher Daten ins Leere sendet, weil kein Subscriber da ist. Dieses Muster garantiert nicht, dass die Daten vom Publisher immer bei den Subscribern ankommen.

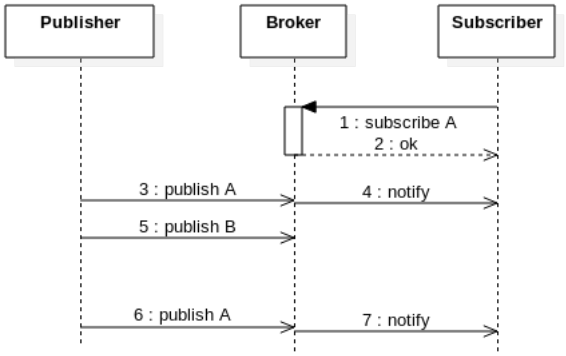


Abbildung 3.3.: Beispiel einer Publish/Subscribe Kommunikation

Unix-Derivate stellen verschiedene Interprozesskommunikationsmittel zur Verfügung. Eine umfassende Zusammenstellung findet sich im Anhang. An dieser Stelle wird nur auf die wichtigsten eingegangen.

Das Request/Reply Kommunikationsmuster wird von dem von QNX benutzten IPC Verfahren (Synchronous Message Passing) unterstützt. Dieses Verfahren lässt sich gut mit den Anforderungen, welche EEROS stellt, vereinen. Leider gibt es unter Linux nichts Vergleichbares. Android Binder ermöglicht zwar synchrone Kommunikation, ist aber zu komplex und hat viele Features, die EEROS nicht tangieren. Es wurde entschieden eine ähnliche Implementation in Linux zu realisieren, weil das QNX Message Passing im Grunde recht einfach ist.

ZeroMQ wird nicht verwendet, weil in anderen Projekten damit schlechte Erfahrungen gemacht wurden. ZeroMQ wurde entwickelt, um die Netzwerkkommunikation

einfach zu machen. Zum Beispiel merkt der Client nicht, dass die Verbindung zum Server getrennt wurde. Der Client blockiert, bis die Verbindung wieder steht, um das sich ZeroMQ kümmert. Das ist zwar gut gemeint, aber in der Praxis meistens unerwünscht.

Beim Publish/Subscribe Kommunikationsmuster lässt sich nicht so einfach ein IPC Verfahren finden. Deshalb wird eine eigene Implementierung dieses Muster auf der Basis von Shared Memory ausgearbeitet. Im Kapitel 3.5 wird das Konzept dieser Implementierung beschrieben.

3.4. Interprozesskommunikation

Bisher liefen alle EEROS-Komponenten (Control System, Safety System und Sequenzen) im gleichen Prozess. Wenn eine Funktionalität einer Komponente benötigt wurde, konnte einfach die entsprechende Methode aufgerufen werden. In Abbildung 3.4 ist das Beispiel der `PickAndPlaceSeq` welche die Methoden des Bahnplaners nutzt.

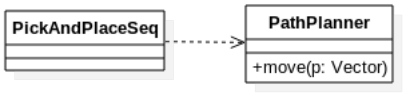


Abbildung 3.4.: `PickAndPlaceSeq` nutzt die Methoden vom `PathPlanner`

Das geht nicht mehr, da sich die Bahnplaner-Sequenz (oder Objekt) in einem anderen Prozess befindet. Um das Objekt dennoch zu gebrauchen, muss dem Prozess mitgeteilt werden, welche Methode es ausführen soll. Dies nennt sich RPC (Remote Procedure Call) oder RMI (Remote Method Invocation), je nachdem, ob nur eine Funktion oder eine Methode aufgerufen wird.

Damit die `MoveSequence` nicht wissen muss, ob es die Methode direkt aufrufen darf oder ob sie über IPC ausgeführt wird, muss sie immer über das Interface gehen, welches von der `PathPlanner` Klasse implementiert wird, wie in Abbildung 3.5 gezeigt. Das ermöglicht das Erstellen einer zweiten Implementation des `MoveInterface`, welche Invoker genannt wird.

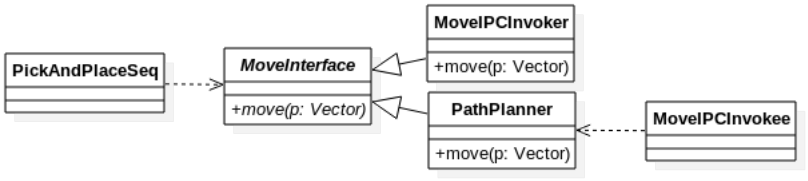


Abbildung 3.5.: `PickAndPlaceSeq` nutzt die Methoden vom `MoveInterface`

Bei der Konstruktion des `PickAndPlaceSeq`-Objekts muss eine Referenz auf den Bahnplaner oder auf ein `MoveIPCInvoker`-Objekt übergeben werden, je nachdem ob der Bahnplaner im gleichen oder einem anderen Prozess ist. Das Gegenstück zum Invoker ist der Invokee, welcher den eigentlichen Methodenaufruf ausführt und den Rückgabewert zurückschickt. Abbildung 3.6 zeigt die Objekte, welche bei der Interprozesskommunikation involviert sind.

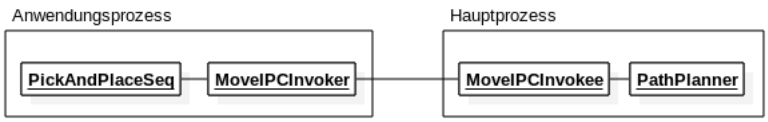


Abbildung 3.6.: Methodenaufrufe gehen über IPC-Invoker und -Invokee

Alle EEROS-Komponenten müssen ein solches Interface anbieten, damit Invoker und Invokees erstellt werden können. Im Interface dürfen nur Methoden vorkommen, da das Schreiben von Felder nicht detektiert werden kann. Falls man Felder benötigt, müssen Getter- und Setter-Methoden erzeugt werden, um darauf zuzugreifen.

3.5. Logging der Signale

Das Control System besteht aus Blöcken (wie z.B. Summierer, Integrierer, Filter, usw.), die hintereinander ausgeführt werden. Die Ausgabe wird in Signal-Objekte gespeichert und mehrere dieser Signal-Objekte können als Input für den nächsten Block verwendet werden. Jeden Taktzyklus ändern sich die Signale. In Abbildung 3.7 ist nochmals das Control System von Kapitel 1 abgebildet.

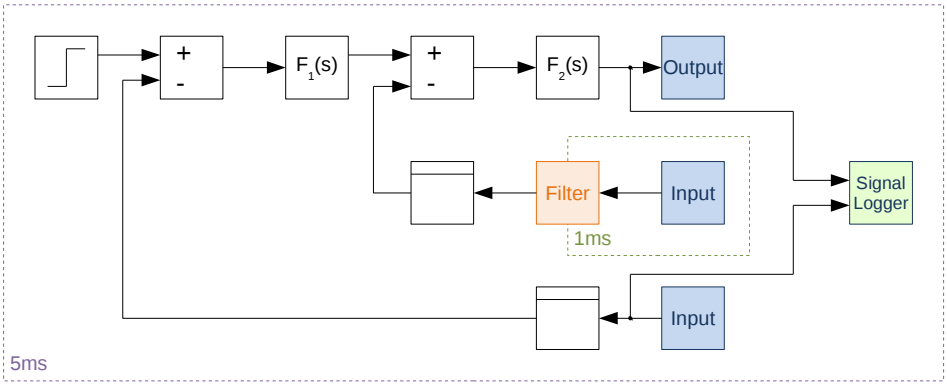


Abbildung 3.7.: Beispiel eines EEROS Control Systems

Das Control System ist in Anlehnung zu Simulink entworfen worden. Um bei Simulink ein Signal zu betrachten, verbindet der Benutzer das Signal mit einem Scope-Block. Analog dazu werden in EEROS die Signale mit dem `SignalLogger` Block verbunden. Dies geschieht, indem ein Subscriber beim `SignalLogger` (Publisher) ein Signal abonniert.

Das Abonnieren geschieht mit der Methode `subscribe`, wie in Abbildung 3.8 dargestellt. Das Aufrufen der Methode wird mit dem bereits beschriebenen Request/Reply Verfahren ausgeführt.

Der `SignalLogger` ist der letzte Block in der Timedomain, der ausgeführt wird. Die Timedomain läuft mit einem konstanten Takt. In jedem Taktzyklus veröffentlicht der `SignalLogger` alle abonnierten Signale, welche der Subscriber dann abholen kann.

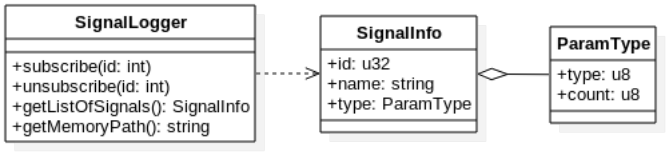


Abbildung 3.8.: Inteface des `SignalLogger` Blocks

Der im Laufe der Zeit erzeugte Datenstrom (`A1`, `B1`, `C1`, usw.) wächst ständig an. Da die Daten etwa mit 1 MB/s anfallen, würde die Speicherung des gesamten Datenstroms schnell zu gross. Da die Subscriber an aktuellen Daten interessiert sind, genügt es, einen Ausschnitt von neuesten Datenwerten, und ein Stück zurück in die Vergangenheit, zur Verfügung zu stellen. Dieser Ausschnitt ist ein Fenster, welches schön gleichmässig über den Datenstrom gleitet und die für die Subscriber abrufbaren Daten beinhaltet.

Abbildung 3.9 zeit dieses Fenster nach vier Taktzyklen. In den ersten drei Zyklen wurden die Signale `A`, `B` und `C` veröffentlicht. Beim vierten Zyklus ist das Signal `D` dazugekommen, welches von einem Subscriber abonniert wurde. Zu jedem Signalwert gehört der Wert selbst, die Länge, ein Zeitstempel und eine ID.

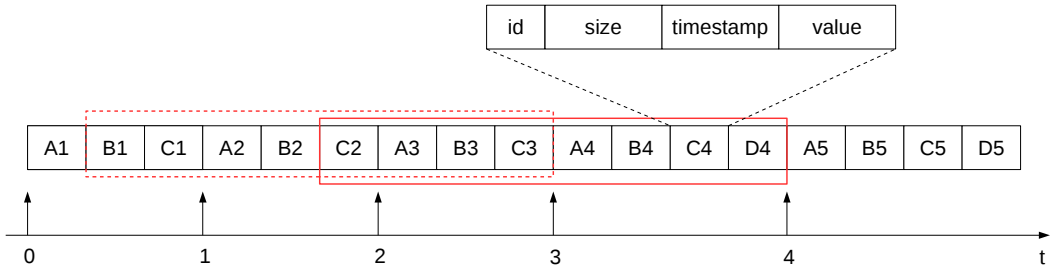


Abbildung 3.9.: Sliding Window bei Publish/Subscribe

Beim Realisieren muss darauf geachtet werden, dass sich der Publisher in einem Realtime-Thread befindet. Der Subscriber darf den Publisher nicht behindern können. Um dies zu verhindern, wird die Benachrichtigung vom Publisher zum Subscriber weggelassen. Der Subscriber muss durch Polling herausfinden, ob neue Signale verfügbar sind. Da die Signale in einem regelmässigen Takt generiert werden, ist dies aber nicht notwendig, weil vorausgesagt werden kann, wann neue Signale generiert werden.

Der Subscriber beobachtet das Fenster in fixen zeitlichen Abständen. Anhand der letzten und aktuellen Fensterposition sieht er, welche Signale neu dazugekommen sind. Die Signale, welche er nicht abonniert hat, werden verworfen. Es wurde Entschieden alle Signale im gleichen Fenster zu veröffentlichen. Eine andere Variante wäre, für jedes Signal ein eigenes Fenster zu erstellen, damit man sich die Metainformationen sparen kann. Weil dies aber nicht so viel ausmacht, wird die einfachere Variante implementiert.

Publisher und Subscriber laufen zwar beide periodisch, aber asynchron zu einander. Der Takt des Subscribers kann viel kleiner gewählt werden, um den Prozessor nicht mit vielen Kontextwechseln auszulasten. Die Grösse des Fensters muss aber so eingestellt sein, dass keine Daten verloren gehen.

3.6. Service Discovery

Auf beiden Seiten der Kommunikation braucht es genau die gleiche Interfacedefinition, das heisst, wenn am Interface was geändert wird, müssen beide Prozesse beendet und alles neu kompiliert werden. Das ist eigentlich nicht so schlimm, weil das Interface normalerweise nicht so oft ändert. Jedoch wäre es sehr umständlich, wenn ein Debugger bei jeder Änderung neu kompiliert werden muss. Das sollte eigentlich nie nötig sein.

Dem Debugger muss es aber möglich sein für jeden EEROS-Prozess herausfinden zu können, welche Funktionen zur Verfügung gestellt werden, wie sie heissen und wie die Signatur dieser Funktionen aussieht. Jeder Prozess hat ein `InformationInterface`, welches in Abbildung 3.10 dargestellt ist, über das der Debugger die nötigen Information kriegt.

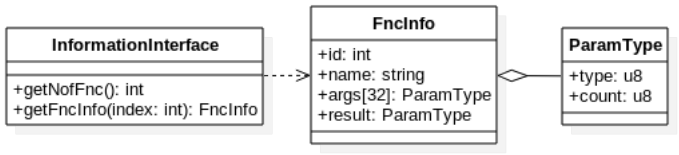


Abbildung 3.10.: Interface für allgemeine Informationen

Die Funktionen der `SignalLogger` Klasse können über die gleichen Deskriptoren erfragt werden und somit kann auch herusgefunden werden, welche Signale im Control System vorhanden sind.

3.7. Echtzeit

Eines der Ziele dieser Arbeit ist Roboterherstellern und Anwendern, welche EEROS verwenden, zu ermöglichen eine fehlertolerante Robotersteuerung zu realisiern. Wenn man eine Schnittstelle nach aussen zur Verfügung stellt, muss sichergestellt sein, dass Fehlverhalten der Nutzer dieser Schnittstelle keinen negativen Einfluss im Inneren haben. Es geht nicht nur darum, ein System vor einem kompletten Absturz zu schützen, sondern auch um Verzögerungen zu verhindern, welche die Echtzeitfähigkeit des Systems beeinträchtigen.

Die erste Schnittstelle, die überprüft wird, ist das Request/Response Message Passing. In Abbildung 3.11 wird nochmal das Beispiel mit dem Bahnplaner aufgegriffen. Die Kommunikation zwischen dem funktionskritischen Bahnplaner und der potenziell fehlerhaften Pick & Place Sequenz läuft über Proxyobjekte. Die kritischen Punkte sind das Aufrufen der `startMove` Methode und das Zurückschicken der Antwort an die Pick & Place Sequenz, weil sie den Bahnplaner involvieren.

Die `startMove` Methode wurde nicht vom Anwender geschrieben und kann grundsätzlich als sicher betrachtet werden. Falls aber zu viele `startMove` Aufrufe gemacht werden, könnte der Bahnplaner überlastet werden. Dieses Problem kann einfach gelöst werden, indem `startMove` sofort `false` zurückgibt, wenn die Bahn noch nicht abgefahren ist.

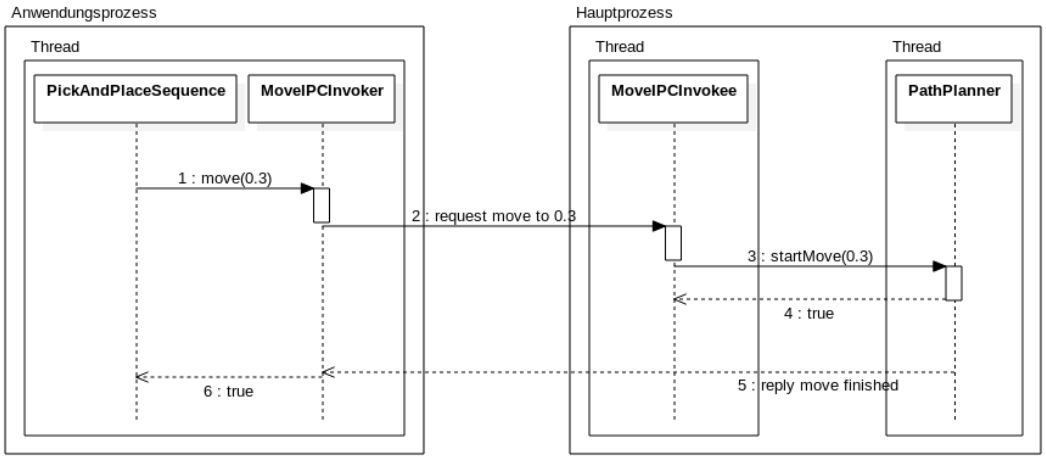


Abbildung 3.11.: Beispiel einer Request/Reply Kommunikation

Das zurückschicken der Antwort ist unproblematisch, weil der Vorgang nicht blockiert. Selbst wenn der Anwendungsprozess nicht mehr da ist, um die Antwort anzunehmen, ignoriert dies der Bahnplaner einfach.

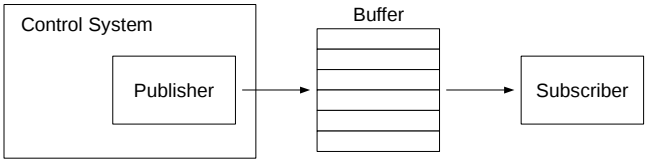


Abbildung 3.12.: Publish/Subscribe Kommunikation

Die zweite Schnittstelle, die analysiert wird, ist der Publish/Subscribe Logger. Abbildung 3.12 zeigt die involvierten Parteien. Der einzige Kontaktpunkt von Publisher und Subscriber ist das Shared Memory, welches beide miteinander teilen. Damit der Subscriber immer einen konsistenten Zustand sieht, muss der Zugriff auf diese gemeinsam genutzte Ressource synchronisiert werden. Dies geschieht grundsätzlich mit Sperren.

Während dem der Subscriber liest, darf der Publisher nicht schreiben. Der Subscriber sperrt den Puffer für den Publisher, bis er alle Daten gelesen hat. Ein langsamer Subscriber, der die Lesesperre nicht schnell genug freigibt, könnte so den Publisher blockieren. Dies kann aber umgangen werden, indem der Publisher in einen Bereich schreibt, welcher vom Subscriber nicht gelesen werden darf. Nachdem alle Daten geschrieben wurden, wird das Fenster nach vorne geschoben.

Der Subscriber muss aber auf die Fensterposition zugreifen, welche vom Publisher gesetzt wird. Dieser Zugriff muss immer noch synchronisiert werden. Bei der Implementierung darf keine Synchronisationsmethode gewählt werden, welche den Publisher blockieren kann, um die Echtzeitfähigkeit zu gewährleisten. Sperren, wie zum Beispiel Mutex oder Semaphore, können nicht verwendet werden, weil ein fehlerhafter Subscriber den Publisher aussperren könnte. In C++ gibt es die Möglichkeit einen Datentyp zu definieren, bei dem jeder Zugriff atomar verläuft (z.B. `std::atomic<int>` für Integer). Somit können keine inkonsistenten Zustände entstehen.

4. Implementierung

4.1. Message Passing Kernelmodul

Um die low-level IPC Kommunikation umzusetzen, wurde ein Linux-Kernelmodul erstellt (mpipc), welches einen Teil der Funktionalität von den QNX Messages implementiert. Beim Laden des Kernelmoduls wird ein Char-Devicefile unter `/dev/mp` erstellt. Dieses Device kümmert sich um die Server (z.B. Bahnplaner). Abbildung 4.1 zeigt, wie sich der Server beim Kernelmodul anmeldet.

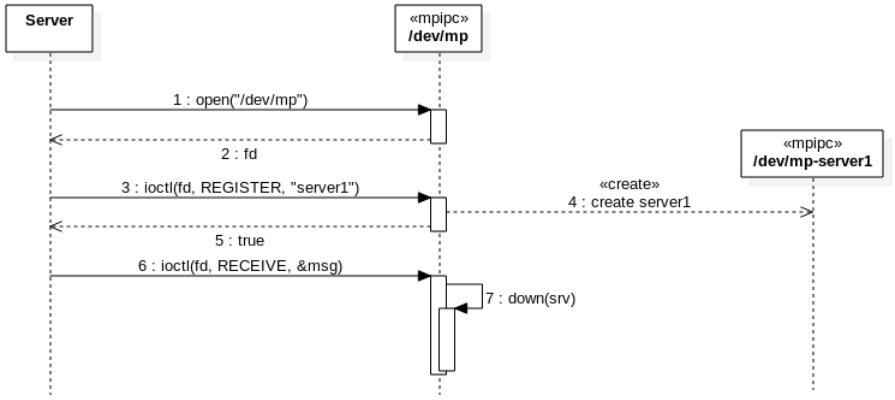


Abbildung 4.1.: Erstellen des Servers

Über einen `ioctl`-Systemcall wird vom Server das Kommando `REGISTER` und der Name des Servers geschickt. Ein neues Devicefile wird erstellt mit dem Namen `mp-server1`. Die Clients können über dieses Device mit dem Server kommunizieren. Falls man lieber alle Server in einem eigenen Unterordner haben will, anstatt direkt unter `/dev`, kann dies mit udev-Regeln realisiert werden.

Wenn der Server bereit zum Empfangen von Nachrichten ist, führt er das Kommando `RECEIVE` aus. Falls keine Messages vorhanden sind, was hier der Fall ist, wird auf dem Semaphore die `down` Operation (P-Operation) ausgeführt. Der Server blockiert.

Bevor wir zur Clientseite kommen, werden die verwendeten Datenstrukturen vorgestellt. Abbildung 4.2 zeigt die Datenstrukturen und wie sie miteinander verbunden sind.

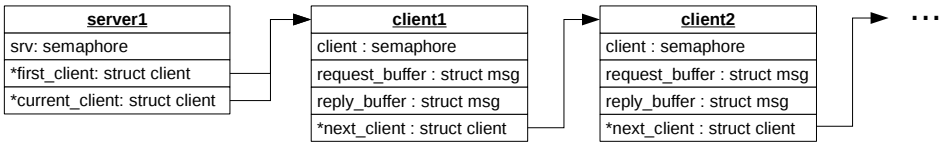


Abbildung 4.2.: Datenstrukturen im Kernel

Für jeden Server und Client werden Objekte erstellt, welche relevante Informationen für die Server bzw. Clients speichern. Das `srv`-Semaphore beim Server wurde bereits erwähnt. Jeder Client hat ebenfalls einen eigenen Semaphore und je einen Message Puffer für die Requests und die Replies. Da der Server alle Clients kennen muss, werden diese in einer Linked-List verbunden.

Abbildung 4.3 zeigt, wie die Verbindung zwischen Client und Server hergestellt wird. Das Devicefile `/dev/mp-server1` wird geöffnet, der Client legt eine Open-Message in seinen Puffer und führt auf dem Server-Semaphore die `up` Operation (V-Operation) aus. Der Request wurde abgeschickt und der Client muss nur noch warten, bis eine Antwort kommt. Dies macht er indem er die `down` Operation auf seinem Semaphore ausführt.

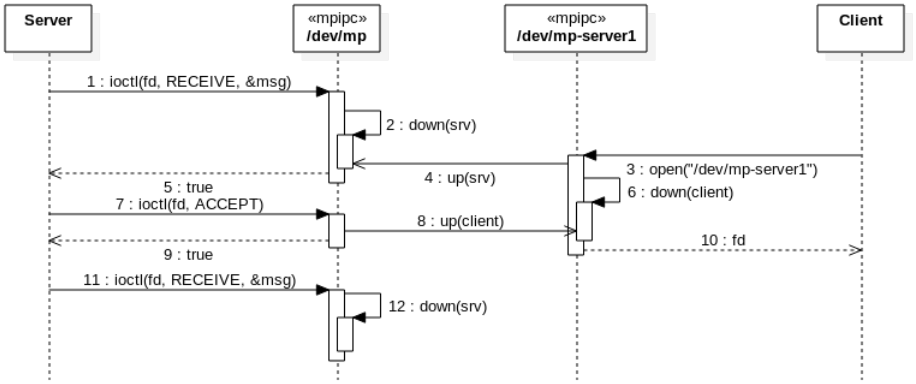


Abbildung 4.3.: Verbindungsherstellung zwischen Client und Server

Der Server wacht im Kernelkontext wieder auf. Er durchläuft die Linked-List und gibt den ersten Request, den er findet, zum Userspace zurück. Die Message, welche vom Client kommt, ist eine Verbindungsanfrage. Der Server kann entscheiden, ob er die Verbindung akzeptieren will. Ist dies der Fall, führt er das `ACCEPT` Kommando aus. Auf dem Client-Semaphore wird die `down` Operation ausgeführt und der Client erhält einen gültigen Filedescriptor. Im selben Muster werden Requests und Replies ausgetauscht, wie in Abbildung 4.4 dargestellt.

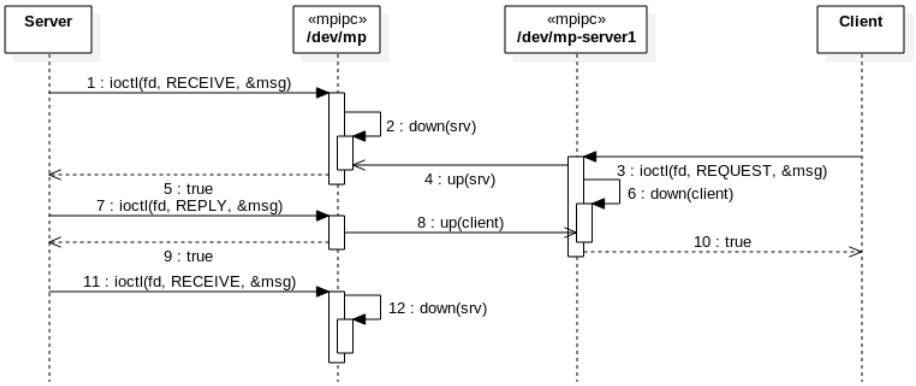


Abbildung 4.4.: Request/Reply Kommunikation zwischen Client und Server

4.2. Message Passing EEROS Client/Server

Das erste was benötigt wird, um die IPC Kommunikation zu zulassen, ist das Interface. Ausschnitt 4.1 legt das Interface des Bahnplaners dar. Einfachheitshalber werden Implementationsdetails, wie virtuelle Dekonstruktoren, weggelassen. Der hier gezeigte Bahnplaner stellt nur eine `move` Methode für eine Achse zur Verfügung.

Listing 4.1: Interface des Bahnplaners

```
class PathPlannerInterface {
public:
    virtual bool move(double position) = 0;
    static constexpr int moveID = 17;
}
```

Als nächstes wird ein Proxyobjekt für den Client benötigt, welches in Ausschnitt 4.2 dargestellt ist. Das Interface des Bahnplaners wird implementiert, indem die Parameter, in diesem Fall nur der Parameter `position`, in einem Request an den Server gesendet werden. Die Antwort vom Server wird in die Variable `result` geschrieben. Der Client kann dann diese Methode direkt aufrufen.

Listing 4.2: Klassendefinition des Proxyobjekts

```
class MoveIPCInvoker : public PathPlannerInterface {
public:
    virtual bool move(double position) {
        bool result;
        ipc.call(moveID, position, result);
        return result;
    }
};
```

Auf der anderen Seite muss man den Server (`eeros::ipc::Server`) starten. Diese Klasse verfügt über eine `addMethod` Methode, wie in Ausschnitt 4.3 gezeigt. Mit dieser Methode lassen sich wiederum andere Methoden unter einem Namen und einer ID registrieren. Es werden Referenzen auf beliebige Objekte und deren Methoden akzeptiert.

Listing 4.3: Server-Methode um die Bahnplaner-Methode zu registrieren

```
template < typename TClass, typename TReturn, typename ... TArgs >
void addMethod( int id,
                const char *name,
                TClass *object,
                TReturn(TClass::*method)(TArgs ...) );
```

Die Bahnplaner Methode `move` kann von überall her dem Server bekanntgegeben werden. Am einfachsten ist das aber im Konstruktor der Bahnplaner Klasse, wie in Ausschnitt 4.4 gezeigt. Auf der Serverseite muss nichts mehr gemacht werden. Wenn Requests von Client kommen, werden sie von der Server Klasse angenommen und verarbeitet. Der Server kümmert sich um den eigentlichen Aufruf der `move` Methode.

Listing 4.4: Konstruktor des Bahnplaners

```
PathPlanner(eeros::ipc::Server &server) {
    server.addMethod(moveID, "move", this, &PathPlanner::move);
}
```

4.3. Signal Logging im Shared Memory

Um die Signale zu loggen, wird ein `SignalLogger` Block im Control System erstellt. Alle Signale werden mit diesem Block verbunden. Diese Verbindung kann nicht durch einen einfachen Pointer oder Referenz gelöst werden, weil die `Signal` Klasse generisch ist und für jeden Signaltyp eine andere Grösse hat. Der `SignalLogger` Block weiss nicht wie gross das Signal ist, und kann es deshalb nicht ins Shared Memory kopieren.

Eine Lösung wäre, das Kopieren in eine Methode der `Signal` Klasse selbst auszulagern. Der Logger würde die dann die `copyToMemory` Methode von jedem Signal ausführen. Das Problem dabei ist, dass ein Teil der Funktionalität des Loggers in die `Signal` Klasse verlagert wird, was gegen die Prinzipien der objektorientierten Programmierung verstösst.

In C++ gibt es Möglichkeiten dieses Problem sauber zu lösen. Im Ausschnitt 4.5 wird dies vereinfacht dargestellt.

Listing 4.5: Konstruktor des Bahnplaners

```
1 template < typename T >
2 void addSignal(Signal<T> &s) {
3     availableSignals[s.getId()] = SignalRef{ [&s] (void *m) -> int {
4         *reinterpret_cast<T*>(m) = s.getValue();
5         return sizeof(T);
6     }};
7 }
```

Es wird keine Referenz auf das Signal gespeichert, sondern eine Lambda-Funktion, die dasselbe wie die bereits beschriebene `copyToMemory` Methode macht. In einer Template-Funktion `addSignal`, in welcher der Typ des Signals bekannt ist, wird die Lambda-Funktion definiert und einem Signal zugewiesen. Um das Signal zu kopieren, führt der Logger einfach die Lambda-Funktion aus.

Auf der anderen Seite, liest der Subscriber die Signale durch Polling aus dem Shared Memory. Wie beim Schreiben, besteht auch beim Lesen das selbe Problem. Die Daten liegen im Memory und müssen in ein Signal kopiert werden. Der Subscriber kann aber keine Referenz auf ein generisches Objekt halten. Das Problem lässt sich auf die gleiche Weise lösen, wie es beim Logger gemacht wurde.

4.4. Serialisierung von Methodenaufrufen

Der vom Client kommende Request kann als Byte-Array betrachtet werden, welches die Parameter für den Methodenaufruf enthält. Diese Parameter müssen irgendwie aus dem Byte-Array und in die Parameterliste des Methodenaufrufs gelangen. Beim Logging musste ein Objekt von unbekanntem Typ kopiert werden, hier muss aber eine Methode ausgeführt werden, bei welcher die Signatur erst zur Laufzeit bekannt ist.

In der C++ Standard Template Library (STL), gibt es eine Klasse die *Tuple* unterstützt. Damit lassen sich die Parameter im Byte-Array abbilden. Ausschnitt 4.6 zeigt zum Beispiel, wie drei Parameter mit `std::tuple` extrahiert werden können.

Listing 4.6: Extraktion von Parametern mit `std::tuple`

```
using ParamType  = std::tuple<Vector3,double,double>;
ParamType *param = reinterpret_cast<ParamType*>(byteArray);
Vector3 position = std::get<0>(*param);
double maxVel    = std::get<1>(*param);
double maxAcc    = std::get<2>(*param);
```

Natürlich können die Variablen `position`, `maxVel` und `maxAcc` nicht verwendet werden, weil sie erst zur Laufzeit bekannt wären. Es muss daher eine Lösung gefunden werden, welche mit der `param` Variable auskommt.

Beim Request wird die ID der Methode mitgesendet, über die die Signatur der Methode herausgefunden werden kann. Es fehlt also nur noch eine Funktion, welche einen Tuple auf einen Methodenaufruf anwendet. Die Funktion ist in Ausschnitt 4.7 abgebildet.

Listing 4.7: Funktion um Parameter auf Methodenaufruf anzuwenden

```
template < typename TClass, typename TReturn, typename ... TArgs >
TReturn apply(  TClass *object,
                TReturn(TClass::*method)(TArgs...),
                const std::tuple<TArgs...> &args )
{
  return apply_method<sizeof...(TArgs)>::execute(object, method, args);
}
```

Die `apply` Funktion instanziert die `apply_method` Template-Klasse (Ausschnitt 4.8) mit der Anzahl der Parameter (`N`) und führt die statische `execute` Funktion aus. Diese Funktion ruft sich selbst rekursiv auf mit `N=N-1`. Für jeden Parameter wird diese Funktion einmal ausgeführt, und jedes Mal wird der letzte Parameter im Tuple `args_tuple` in die Parameterliste `args_mapped` verschoben. Bei `N=0` sind alle Parameter ausgepackt und der eigentliche Methodenaufruf kann stattfinden. Diese rekursiven Funktionsaufrufe passieren während dem Kompilieren. Zur Laufzeit verhalten sie sich wie ein hartkodierter Methodenaufruf. Die Spezialisierung von `apply_method` mit `N=0` ist nicht dargestellt.

Listing 4.8: Hilfs-Klasse um Parameter auf Methodenaufruf anzuwenden

```
template < int N >
struct apply_method {
    template <  typename TClass,
                typename TReturn,
                typename ... TArgsMethod,
                typename ... TArgsTuple,
                typename ... TArgs >
    static inline
    TReturn execute(    TClass *object,
                        TReturn(TClass::*method)(TArgsMethod...),
                        const std::tuple<TArgsTuple...>& args_tuple,
                        TArgs ... args_mapped )
    {
        return apply_method<N-1>::execute(  object,
                                            method,
                                            args_tuple,
                                            std::get<N-1>(args_tuple),
                                            args_mapped... );
    }
};
```

5. Stand der Arbeit

In dieser Arbeit wurde ein IPC Interface für EEROS realisiert. Es ermöglicht die Kommunikation zwischen EEROS-Komponenten, welche sich in anderen Prozessen befinden könnten. Das umgesetzte Konzept konnte erfolgreich in Beispielanwendungen getestet werden. Zu diesem Zeitpunkt sehen die Resultate vielversprechend aus. Es sind aber mehr Tests notwendig, um herauszufinden, ob sich das Konzept in der Praxis etablieren kann.

Aus zeitlichen Gründen konnte das entwickelte Interface nicht in allen EEROS-Komponenten eingebunden werden. Zum Beispiel gibt es in EEROS einen Message Logger, mit dem sich Log-Nachrichten, wie z.B. „Initialisierung gestartet“, aufzeichnen lassen. Diese Messages könnten direkt mit den Signalen im Shared Memory geloggt werden. Auch beim Safety System fehlt das Interface. Jedoch handelt es sich bei den fehlenden Elementen nur um die Integration und nicht um fehlende Features des Interfaces.

Für die eigentliche Kommunikation wurde ein Linux-Kernelmodul erstellt. Es ermöglicht die Kommunikation von Messages im Request/Reply Prinzip. Der Inhalt der Messages kann beliebig sein. Die Länge einer Message ist auf 4 kB begrenzt. Diese Begrenzung ist willkürlich und wurde eingebaut, um den Code einfach zu halten. Im Allgemeinen, wurden bei der Effizienz schnell Kompromisse gemacht, um die Implementierung der Funktionalität möglichst rasch voranzutreiben. Wenn man dieses Kernelmodul in richtigen Projekten einsetzen will, sollte es vorher zuerst überarbeitet werden. Das Kernelmodul ist komplett unabhängig von EEROS. Auch andere Anwendungen könnten es für die Interprozesskommunikation nutzten.

Signale werden nach dem Publish/Subscribe Verfahren ins Shared Memory geloggt. Das Abonnieren der Signale läuft über die Request/Reply Schnittstelle. Aktuell muss der Name des Shared Memory Bereichs von Publisher zu Subscriber übergeben werden, was das Einrichten komplizierter macht. Der Shared Memory Bereich könnte vom Subscriber direkt über das Kernelmodul gemappt werden. Bei der Konfiguration gibt es ausserdem noch Parameter, welche im Moment noch vom Programmierer bestimmt werden müssen. Die Logger Komponente kennt jedoch alle Eigenschaften des Systems und könnte grundsätzlich die Parameter (z.B. Grösse des Schiebefensters) automatisch bestimmen.

6. Ausblick

Um das in dieser Arbeit realisierte Interface in einer EEROS-Komponente zu implementieren, muss relativ viel Code geschrieben werden. Dieser Code ist nicht sehr anspruchsvoll und könnte automatisch generiert werden, um dem Programmierer Tipparbeit zu ersparen und damit er sich auf das wesentliche konzentrieren kann. Evtl. kann sogar das Interface direkt aus dem C++ Headerfile geparst werden (z.B. mit *clang*).

Beim Kernelmodul lässt sich viel an der Effizienz verbessern. Die Messages werden zweimal kopiert, einmal vom Client in einen Kernelpuffer und das zweite Mal vom Kernelpuffer zum Server. Dies kann auf eine Kopie reduziert werden, indem der Kernel direkt vom Client zum Server kopiert. Falls dies immer noch nicht schnell genug geht, könnte der Client-Puffer direkt in den Speicherbereich des Servers gemappt werden. Somit fallen keine Kopien an und es können auch grössere Datenmengen sehr effizient kommuniziert werden.

Es gibt viele Features welche von QNX angeboten werden, die aber nicht vom in dieser Arbeit entwickelten Kernelmodul unterstützt werden. Zum Beispiel wird strikt das Request/Reply Muster implementiert. In gewissen Fällen wird keine Antwort von Server benötigt. Der Client möchte den Server nur über ein Ereignis informieren. In QNX werden die Nachrichten, auf die keine Antwort verlangt werden, Pulse genannt. Ausserdem unterstützt QNX Multipart-Messages und nur teilweise konsumierte Nachrichten.

In einer MAS Masterarbeit wurde ein REST Interface in Python für die Visualisierung von Roboterdaten erstellt. Damit diese Visualisierung verwendet werden kann, muss die Message Passing Interprozesskommunikation in Python realisiert werden. Die Schnittstelle zum Kernelmodul ist in C geschrieben, was sich gut in Python integrieren lässt. Es werden aber nur Signale visualisiert. Über das in dieser Arbeit entwickelte System lässt sich die Struktur des gesamten Control Systems aufdecken. Diese Struktur könnte wie in Simulink visualisiert werden, um das Debugging und das Optimieren des Systems zu erleichtern.

Literaturverzeichnis

[1] David A Rusling, *The Linux Kernel*. (Stand Januar 2015)
`http://www.tldp.org/LDP/tlk/`

[2] QNX Software Systems, *QNX 6.6 Documentation*. (Stand Januar 2015)
`http://www.qnx.com/developers/docs/660/index.jsp`

[3] Wikipedia, *Unix Domain Sockets*. (Stand Januar 2015)
`http://en.wikipedia.org/wiki/Unix_domain_socket`

[4] Wikipedia, *Synchronous Interprocess Messaging Project for LINUX*. (Stand Januar 2015)
`http://en.wikipedia.org/wiki/SIMPL`

[5] Embedded Linux Wiki, *Android Binder*. (Stand Januar 2015)
`http://elinux.org/Android_Binder`

[6] Linux Weekly News, *Kdbus meets linux-kernel*. (Stand Januar 2015)
`http://lwn.net/Articles/619068/`

[7] Greg Kroah-Hartman, *Kdbus Details*. (Stand Januar 2015)
`http://kroah.com/log/blog/2014/01/15/kdbus-details/`

A. Grundlegende IPC Arten

A.1. Unix / POSIX IPC

In den folgenden Abschnitten werden die IPC Möglichkeiten von Linux und QNX zusammengefasst.

Signale sind eine sehr primitive Art der Interprozess-Kommunikation. Ein Prozess kann ein Signal an einen anderen Prozess senden, was einen Funktionsaufruf im zweiten Prozess verursacht. Die Anzahl der Signale ist beschränkt auf die Wortgrösse des Prozessors (32 Signale für 32-Bit Prozessoren und 64 Signale für 64-Bit Prozessoren).

Pipes und FIFOs sind virtuelle Dateien über die mehrere Prozesse miteinader kommunizieren können, indem ein Prozess in die Datei schreibt und der Andere daraus liest. Die Datei wird nicht auf die Festplatte geschrieben, sondern existiert nur in einem Buffer, der vom Kernel verwaltet wird. Der Unterschied zwischen einer Pipe und einem FIFO ist, dass eine Pipe nur von verwandten Prozessen benutzt werden kann und bei einer FIFO können beliebige Prozesse miteinander Kommunizieren. Die FIFO wird deshalb von einer Gerätedatei im Dateisystem repräsentiert.

Message Queues sind ähnlich wie Pipes und FIFOs mit dem Unterschied, dass ganze Nachrichten und nicht Bytes übertragen werden. Es wird immer eine ganze Nachricht gesendet oder empfangen. Es gibt keine partielle Nachrichtenübertragung. Wie Pipes und FIFOs sind Message Queues asynchron.

Shared Memory kann für sehr schnelle und effektive IPC Kommunikation genutzt werden. Mehrere Prozesse teilen sich einen Speicherbereich, auf den alle direkt Zugriff haben. Das Problem bei Shared Memory ist die Synchronisierung des Zugriffs auf den Speicher, was z.B. mit einem anonymen Semaphore gelöst werden kann.

Sockets sind bekannt aus der Netzwerkprogrammierung. Unter Unix Systemen können Sockets für die Interprozess-Kommunikation verwendet werden. Der Vorteil von diesen sogenannten Unix Domain Sockets ist, dass der Netzwerkprotokoll Overhead wegfällt. Die zwei Übertragungsarten *Stream* und *Datagram* entsprechen TCP und UDP bei Netzwerksockets, welche verbindungsorientierte, zuverlässige Bytestreams oder verbindungslose, unzuverlässige Pakete sind. Unix Domain Sockets sind jedoch immer zuverlässig. Die dritte Übertragungsart *Sequencial Packet* hat kein Netzwerkgegenstück. Sequencial Packets sind verbindungsorientiert und wie der Name schon sagt, werden Pakete übermittelt.

A.2. QNX Message Passing

QNX ist ein Microkernel-Betriebssystem. In einer solchen Architektur implementiert der Kernel nur die notwendigsten Funktionen, wie zum Beispiel Memorymanagement und Prozessmanagement. Alle anderen Komponenten des Betriebssystem (Filesysteme, Netzwerk, ...) laufen in seperaten Prozessen, die miteinander über IPC kommunizieren. Das macht die Interprozesskommunikation zur wichtigsten Komponente des Betriebssystems, da sie die einzelnen Komponenten miteinader verbindet. Selbst die Systemaufrufe wie `open`, `read`, `write`, usw. werden über IPC implementiert.

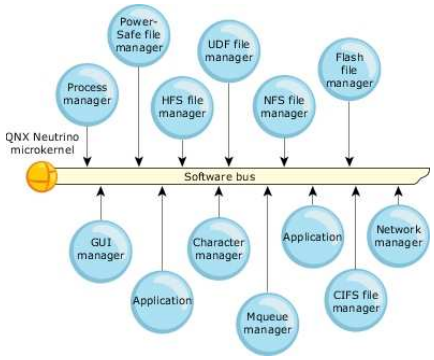
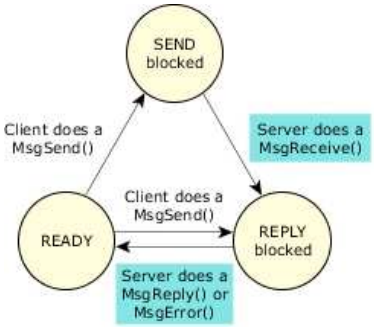
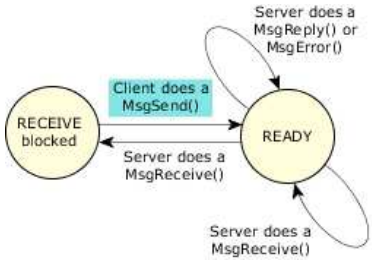


Abbildung A.1.: QNX Neutrino Architektur

Kommuniziert wird, indem Prozesse synchron miteinander Nachrichten austauschen im Client/Server-Prinzip. Der Client sendet eine Nachricht an den Server und blockiert gleich danach im `send-blocked` State. Der Server empfängt diese Nachricht was den Client in den `reply-blocked` State wechseln lässt. Der Server kann nun die Nachricht verarbeiten und eine Antwort zum Client zurückschicken, was den Client in den `ready` State versetzt.



(a) Client



(b) Server

Abbildung A.2.: Prozess-Status State-Machine

Die Nachrichten werden direkt in den Speierbereich vom anderen Prozess kopiert. Es kommt also nur zu einer Kopie, was sehr effizient ist. Dennoch kan das bei grösseren Datenmenge unerwünscht sein. QNX ermöglicht deshalb das Übermitteln von mehrteiligen Nachrichten. Im ersten Teil können Informationen über die Daten übermittelt werden, die dem Server erlauben zu entscheiden, ob er die restlichen Teile auch empfangen will. Das erlaubt es auch das Minimum von einer Kopie wirklich einzuhalten, da die Daten vor dem Senden nicht zuerst in eine Struktur kopiert

werden müssen, sondern dort bleiben können wo sie sind, weil nur ein Zeiger kopiert wird.

Für manche Nachrichten braucht der Sender keine Antwort, zum Beispiel, um ein Event mitzuteilen. QNX nennt solche Nachrichten Pulse, jedoch können nur der Typ und ein 32-Bit Integer übermittelt werden. Eine weitere Variante solcher Events ist ein Softwareinterrupt.

Leider gibt es unter Linux standardmässig nichts ähnliches zu den QNX-Messages. Im Internet findet man zwei Implementierungen, die aber beide recht alt und wahrscheinlich nicht mehr supported werden. `SIMPL` ist ein Projekt, das versucht hat synchrone Nachrichtenübertragung im QNX-stil zu implmentieren (über Pipes und Semaphoren). Das zweite Projekt heisst `srripc` und ist ein Kernelmodul, welches möglichst QNX-API kompatibel gehalten wurde, um das Portieren von Code zu erleichtern.

A.3. ZeroMQ

ZeroMQ ist eine intelligente Socket Library, um Nachrichten über TCP, UDP, aber auch Unix Domain Sockets zu verschicken. Es werden viele Kommunikationsmuster unterstützt (Request/Reply, Publish/Subscribe, parallel Pipelines und andere). ZeroMQ versucht dem Programmierer so viel Arbeit wie möglich abzunehmen. Zum Beispiel kann bei ZeroMQ der Client vor dem Server gestartet werden, ZeroMQ kümert sich darum, dass die Verbindung trotzdem zustande kommt.

A.4. Android Binder

Binder ist ein Linux Kernelmodul, welches bei Android verwendet wird, um Remote Procedure Calls auszuführen. Binder wurde entwickelt, weil es im Linux Kernel keine effizienten IPC Mechanismen gibt mit welchen man unnötige Kopien vermeiden kann und um die Lebensdauer der übertragenen Objekte zu verwalten. Binder ist sehr komplex und macht mehr als nur IPC. Es kümmert sich um Thread-Management, Memory-Management und mehr. Ausserdem wurde die Implementierung bemängelt, was auch ein Grund war wieso Binder lange Zeit im Staging Bereich des Linux-Kernels war.

A.5. D-Bus

D-Bus ist im Unterschied zu Binder asynchron. Nachrichten werden in einer Art Mailbox gespeichert und asynchron verarbeitet. D-Bus ist sicherer als Binder, jedoch nicht so effizient. Eine Version für Linux ist in Entwicklung welche die Effizienzprobleme gelöst hat und es können über 1 GB an Daten übermittelt werden. D-Bus ist sehr komplex, unterstützt aber sehr viele Funktionen.