ABSTRACT

Title of Document:	DEVELOPMENT OF A REUSABLE TOP-LEVEL CONTROL ARCHITECTURE FOR A ROBOTIC MANIPULATOR
	Nicholas John D'Amore Master of Science, 2010
Directed By:	Associate Professor David L. Akin Department of Aerospace Engineering

The capabilities of a robotic system are strongly constrained by the capabilities of its control software. The development of this software represents a substantial fraction of the development effort of the overall system, due in part to the difficulty of reusing software written for previous robotic applications. A reusable software control architecture therefore has enormous potential to expedite the development and reduce the cost of this development process. This thesis presents a component-based reusable architecture for the top-level control of a robotic manipulator, developed within the Open Robot Control Software (Orocos) framework. This framework enables the development of software components that are applicable to a variety of robotic manipulators. The software is implemented on an existing manipulator platform as a demonstration of basic functionality. Simulations are conducted to verify adaptability to other kinematic arrangements.

DEVELOPMENT OF A REUSABLE TOP-LEVEL CONTROL ARCHITECTURE FOR A ROBOTIC MANIPULATOR

By

Nicholas John D'Amore

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2010

Advisory Committee:
Associate Professor David L. Akin, Chair
Assistant Professor Raymond Sedwick
Professor Norman Wereley

Acknowledgements

I would like to thank Professor David Akin for his support and guidance as thesis advisor throughout this endeavor. Thanks are also owed to Professor J. Sean Humbert for making available his lab's Vicon system, and to his graduate student Greg Gremillion for preparing the Vicon system and performing the data collection.

Thank you to Andrew Ellsberry, whose contemporaneous thesis work on electronic hardware for robotic systems enabled the present study to move beyond simulation and onto an actual robot. For assembling, disassembling and reassembling the arm countless times; for producing the end effector that joined Ranger and Faro; for calibrating Faro and recording the test data; and for your patience through all the lower-level software bugs that could only be diagnosed by plugging in the arm and seeing what it did. Your endless hours of hardware preparation, repairs, and supervision are greatly appreciated.

Thank you to Kate McBryan for generously taking the time to read and comment on an earlier draft of this thesis. Thank you to Barrett Dillow for suggesting enhancements to the client command interface. And thank you to Heather Bradshaw, whose ready-made, handcrafted reflective markers enabled rapid preparation of Ranger for the Vicon tests.

Last but not least, I thank my family—Mom, Dad, and Alex—for your endless support and for your understanding during the busy final months of this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

When a new robot is developed, its software system is often custom built with little or no reuse of previously developed software [1],[2]. Moreover, the development of such software is often seen as simply a hurdle to be overcome on the way to a higher-level research goal for which the software is required [1]. Within the past decade, however, several open source projects have devoted considerable attention to the development of reusable robotic software as a goal in and of itself. By devoting extensive attention to a reusable software framework rather than a specific application, these projects have the potential to greatly reduce the time and effort needed for application development. By leveraging the results of these open source efforts, the present work aims to implement a software system for the top-level control of a robotic manipulator having sufficient flexibility to enable a wide range of advanced research.

1.2 Requirements

The following requirements drove the development of this system:

- The system must be capable of point-to-point motion to a desired goal specified in Cartesian space, performing all necessary forward and inverse kinematic calculations.
- The system must provide the capability to execute motion specified as a sequence of waypoints.

- The system must be adaptable, with modest effort, to a variety of serial-link robotic manipulators to enable reuse of the developed software.
- The system must provide a simple command interface for external software to send basic commands to the system and receive updates from the system via a protocol that will not unduly constrain the development of the external software.
- The system must be extensible to encompass more advanced capabilities such as visual servoing, path planning, collision avoidance, and nonlinear control laws.

1.3 Previous Work in Software Reusability

Nesnas [3] divides the existing approaches to writing reusable software into two categories. The first is a component-based approach, in which functional software units are written with the intention of being directly reused as building blocks for multiple applications, with suitably defined and abstracted ways of passing information between these blocks. The second style of approach is object-oriented, in which the reusable elements are generic abstract base classes from which one derives specialized classes for each robotic system's unique requirements, thus enforcing a level of standardization on the specialized classes that, it is hoped, will promote reusability. Nesnas [4] attributes the historically limited success of the component-based approach to excessive generality, leaving undone too much of the software engineering needed to produce the domain-specific solutions needed by robotics researchers. To historical object-oriented approaches, Nesnas attributes exactly the opposite problem, observing that domain-specificity of the software led to only limited levels of reuse that was largely confined to the particular domain in which it was originally developed. Thus, a significant challenge in this area is to find the appropriate balance of generality and specificity, providing enough capability to be genuinely and directly useful in practice without becoming deeply tied to a specific application.

Fitzpatrick, Metta, and Natale [1] observe that, particularly with regard to domain specificity, robotic software development bears a resemblance to natural evolution. All software has a domain in which it can be used. It develops within that domain, possibly expanding to nearby domains as it changes and grows. If this domain is too small, "genetic isolation" will quickly drive it to obsolescence as new technologies bring about new requirements that the old software is unable to fulfill. Thus, reusability is the key to survival in this Darwinian competition, with the fittest software exhibiting reusability across both time (i.e., from past to present and future applications) and place (i.e., between different researchers at different labs). The authors go on to extol the merits of a modular approach in this regard, where they define modularity as the absence of unnecessary dependencies. The authors' argument here presents strongly in favor of a component-based approach, pointing out its flexibility to adapt to change as new components cleanly step in to replace obsolete ones without catastrophic effects rippling throughout the remainder of the code. However, it must be noted that component-based and object-oriented reusability strategies are not at odds with one another. Either strategy may be employed separately; or both may be employed together, with useful objects used and reused internally by components and even passed between components as an exchange of information. Moreover, the object-oriented approach has the potential to enable a significant degree of modularity in its own right, if abstract classes are carefully used to standardize interfaces such that a new child class may be readily inserted to replace an obsolete one. Realizing this, however, would appear to require greater care to avoid dependencies that the component-based approach avoids by its very nature. This raises an important additional consideration.

That a piece of software can in theory be reused does not necessarily mean that it will be reused in practice. Software may see greater reuse if it promotes an architectural style that benefits the development process in ways other than mere reusability of code. Eve Coste-Maniere and Reid Simmons [5] observe that robotic systems are becoming increasingly complex, and that the right choice of architecture can go a long way toward managing that complexity. The appropriate choice of architecture, they claim, can ease the specification, execution, and validation of robotic systems. Specification refers to the management of interactions, both between the system and its environment and between individual elements within the system. An architecture can aid in this area by providing a structured decomposition of the system into smaller components and abstracting the flow of data between those components, thus simplifying the interactions that must occur. Execution refers to the run-time behavior of the system, including its real-time abilities, scheduling of tasks, and appropriate and reliable behavior. This may include resolving conflicting behaviors between tasks and invoking exception handlers whenever appropriate. The third dimension of development, validation, refers to testing and verification of the system. In this area, it may be desirable for an architecture to permit testing individual system components independently of the rest of the system, which may not be written yet. Viewing internal variables and visualizing data are useful capabilities here.

The following subsections describe specific projects that have aimed to produce reusable software for robotic systems. In addition to their architectural benefits, particular emphasis is placed on capabilities for performing the manipulator kinematics calculations required for the present application. Closed-source projects such as Microsoft Robotics Developer Studio (MRDS) were not considered for the present application. Although MRDS has some useful features, the closed-source model necessarily introduces additional constraints on the roboticist's capabilities, and leaves him or her dependent on the original developer for ongoing maintenance of the platform.

1.3.1 Orocos

The Open Robot Control Software (Orocos) project combines both the component-based and object-oriented reusability strategies. The project has yielded four C++ libraries, two geared toward each strategy. The Real-Time Toolkit (RTT) and Orocos Component Library (OCL) establish a component-based infrastructure and a library of ready-to-use components, providing the high level management of interactions within an application. Components exchange information primarily through abstracted Data-Flow Ports, an anonymous publish-subscribe system in which a component does not know where its inputs are originating or where its outputs are being utilized. These Data-Flow Ports may be either buffered, in which case messages are stored in a single queue per link, or unbuffered, in which case only the most recent message is available to be read by any subscribers. This decomposition and isolation of the components enables easier specification of the interactions taking place within the software and easier validation of individual components, because this organization is conducive to generating test inputs and examining the resulting outputs. A system of synchronous methods and asynchronous commands is also defined for explicitly requesting information or behavior of a component when the continuously updating Data-Flow Ports are not appropriate. Through the use of the Common Object Request Broker Architecture (CORBA), one may even interconnect components running in different processes on different computers. The remaining two libraries of the Orocos Project, the Kinematics and Dynamics Library (KDL) and Bayesian Filtering Library (BFL), instead employ the object-based reusability philosophy by providing a library of generically useful and reusable classes for use within an Orocos component (or in a non-Orocos-based application). By embracing both the component-based and object-oriented reusability methodologies, Project Orocos encourages application writers to take advantage of the best that each has to offer.

The Orocos libraries are all released under either the GNU General Public License (GPL) or the GNU Lesser General Public License (LGPL), with terms that explicitly allow for any application or component to remain the property of the creator and to be distributed under any license that the creator sees fit.[1] Moreover, the project's website shows indications of continuous maintenance and development, with active discussion in its online forum, releases of updated versions of its libraries,

[1] Modifications to the Orocos libraries themselves may be distributed only under the appropriate GNU public license.

and preparations for the release of a version 2.0 of the RTT aimed at making the system faster and simpler. Orocos supports standard GNU Linux, the Xenomai and LXRT real-time Linux extensions, Mac OS X, as well as Microsoft Windows.

1.3.2 CLARAty

The Coupled Layer Architecture for Robotic Autonomy (CLARAty) is a reusable robotic software framework collaboratively developed by the Jet Propulsion Laboratory, NASA Ames Research Center, Carnegie Mellon, and the University of Minnesota. The primary application for the work was on research rovers for NASA's Mars Technology Program. The overarching idea behind CLARAty is to decompose robotic software into a decision layer and a functional layer, and to provide a collection of standardized and reusable classes for use within these layers [4]. It therefore falls under the object-oriented approach to reusability.

CLARAty aims for considerably more generality in its modeling of kinematics than Orocos. CLARAty includes a Mechanism_Model class to generically model any articulated mechanism, whether a robotic manipulator or rover mobility system (e.g., rocker-bogie). It is comprised of a tree structure of rigid bodies of class ME_Body, with all attachments governed by an ME_Joint. Each body may have only one parent within the tree, but may have multiple children. It is therefore possible to model multiple manipulators connected to a common base within a single Mechanism_Model. A structure is also defined for implementing linear constraints between joint values, enabling the modeling of simple parallel-link kinematic arrangements, a capability not present in Orocos. The kinematic structure may be specified within the C++ code itself or loaded from an xml file. A generic forward kinematics system allows the querying of the relative transformation between any two frames in the mechanism. A generic inverse kinematics algorithm, although envisioned, has not yet been implemented. Thus, although the Mechanism_Model is applicable to a greater variety of mechanisms than anything in Orocos, its utility is limited.

A small fraction of the CLARAty code was publicly released in June 2007. Further public releases were scheduled to occur in 2008 providing complete capability for driving, terrain sensing, and path planning for a rover. These public releases, however, have not yet occurred. Additionally, although the CLARAty project's publications describe a decomposition into functional and decision layers with objects accessible at various levels of abstraction, there is relatively little to force users to comply with this or any other architectural design, leaving it to the skills and judgment of the programmer to follow sound coding practices. A portion of the software is publicly available, however the license under which it is released prohibits commercial use and is incompatible with the GNU General Public License, thus reducing its likelihood of building up a large support community.

1.3.3 The Chimera Methodology

The Chimera architecture has its origins in a real-time operating system of the same name developed at Carnegie Mellon University (CMU) starting in the late 1980s [6]. To maximize portability, Chimera was written to run on the Motorola 68020, a popular general-purpose processor of the time. Programs were written in the C programming language, with standard Unix libraries ported to the Chimera platform. The system was expandable to meet performance requirements through the addition of more CPUs from the same family. By the mid-1990s, this real-time architecture had evolved to support a component-based reusable software paradigm bearing a strong resemblance to the Orocos framework established in the RTT [7]. The authors envisioned a programming environment which would one day allow users to download software modules that had already been written and fully tested. Rather than reading a journal paper describing the algorithm and spending days or weeks themselves implementing and testing it, the user would be able to integrate this prepackaged module into a new application in a matter of hours.

The Chimera software saw use on CMU's Reconfigurable Modular Manipulator System (RMMS). The RMMS project developed a system that was modular not only in its software, but in the physical hardware as well. Interchangeable, general-purpose link and joint modules allowed the assembly of a variety of special purpose manipulators. The hardware modules, which contained DC motors and custom electronics, could be quickly coupled by hand using locking collars that required no tools. Chimera software components could be assembled via a graphical tool, dragging and dropping the components onto a canvas to form the desired application. Relevant configuration information, such as Denavit-Hartenberg parameters, was specified via a data file for each component.

Chimera appears to have been somewhat successful in its reusability goal. In addition to Carnegie Mellon University, the Chimera real-time operating system has seen use at the Jet Propulsion Laboratory, California Institute of Technology, Concordia University, Air Force Institute of Technology, and University of Alberta [8]. The present author was not, however, able to find any indication that Chimera has seen substantial usage since the 1990s. This may be attributable in part to the architecture's dependence on the Motorola 680x0 family of processors, which has declined in usage since that time, especially in the market of desktop computers. Moreover, unlike the other robot software packages discussed in this section, Chimera is an operating system unto itself. Although the effort was undertaken to port useful Unix C libraries to this operating system to enhance code reusability, it nonetheless requires setting aside other operating systems (such as Linux, Mac OS, or Windows) with which the user may be more familiar and for which ongoing development and maintenance will more assuredly continue.

1.3.4 Yet Another Robot Platform (YARP)

YARP is a robot platform focusing on the communication between networked elements in a robotic system, enabling a component-based software architecture [1]. The focus is on flexible methods of communication, enabling easy swapping among its family of connection types. The communicating components may even be on different computers running different operating systems. Its simple communication protocols also readily support interaction with non-YARP software. The project also aims to enable flexible interfaces with hardware devices, such as motors and cameras. YARP has its origins in humanoid robotics, focusing primarily on behaviors involving sensors and actuators and ensuring that the coupling between these behaviors is sufficiently loose to enable a system to evolve to meet new requirements. YARP aims to be highly portable, employing no external libraries beyond ACE (which has itself been ported to Windows, Mac OS X, Linux, VxWorks, and other operating systems). It does not provide any built-in capabilities directly applicable to manipulator kinematics.

1.3.5 The Player Project

The Player Project focuses instead on mobile robot applications and claims to be “probably the most widely used robot control interface in the world” [9]. In its typical implementation, a server runs onboard the robot and loads the appropriate drivers for the robot’s hardware. Standardized interfaces are provided for interacting with different classes of devices. This provides an abstraction of the robot, which a client program may then control without concerning itself with the underlying hardware. Player is therefore primarily a hardware abstraction layer; and while the client/server model does decouple the higher-level control software from the details of the hardware, it does not promote modularity within the control software itself. Player aims not to constrain such higher-level architectural decisions. While this may be useful in the relatively predictable domain of simple mobile robots, it does not seem especially well suited to the present application of robotic manipulation.

1.3.6 Robot Operating System (ROS)

ROS is an open source robot software platform designed with the primary goal of enabling software reuse. It is intended to be a thin architecture, providing sufficiently few constraints as to be integrable with software written for other platforms, such as Orocos and Player [10]. Like Orocos, it employs both the component-based and object-oriented reusability strategies. ROS provides the infrastructure for writing software components, called nodes, and for exchanging information between these nodes via an anonymous publish-subscribe model[2]. Message transport between nodes occurs via TCP/IP or UDP/IP, thus allowing the nodes to be easily distributed over multiple computers. This networking style allows for individual components (which are separate processes) to be shut down, modified, recompiled, and re-initiated without interrupting the function of the rest of the application. Similar to Orocos, the ROS developers recognized that this form of continuous data streaming is not appropriate for all interactions, and thus implemented a message-response form for command-style interactions.

The ROS project has built up a large collection of software packages for various purposes (serial port communication, stereo vision processing, motion planning, etc.) and devices (SICK laser scanners, Nintendo Wiimote, a variety of cameras, etc.). Some packages are specific to individual robotic systems such as Stanford's PR2 manipulator, while others are generic. ROS draws heavily from other open source robotics projects for useful functionality, incorporating code from Orocos, OpenCV, Player, and other freely available software libraries. Because it includes as one of its packages the Orocos Kinematics and Dynamics Library (KDL), ROS could be said to have all of the same manipulator kinematics capabilities as Orocos itself. While this is not a particularly noteworthy achievement since the KDL was written to be portable to non-Orocos applications, in carrying the KDL as a package ROS may be providing a beneficial service to its users by pulling the best that other software packages have to offer into a single collection. This has the potential to grow to daunting size. ROS organizes the approximately one thousand software modules listed on its website into approximately 130 groups, called stacks.

[2] Published messages are buffered in a separate queue for each connected subscriber.

While this significantly reduces the sense of chaos experienced by someone approaching ROS for the first time, one is still left with a moderately long list of only brusquely described software units.

A noteworthy feature of the ROS style is what its creators call a tools-based design philosophy [11]. ROS provides a utility for converting data streams to text, which can then be used in conjunction with Unix tools such as `grep` to inform the user when a message meeting certain criteria has been transmitted. A virtual oscilloscope is available for plotting numerical variables as they change over time. Another tool makes available debug information for the streaming links between components, including the publishing rate of messages and the type of the messages, with the ability to publish messages from that utility[3]. Another utility allows the recording and playback (onto the application network) of data streaming between nodes, with a tool for offline visualization and inspection of these messages. A utility called `rxgraph` graphically displays the running network of node interconnections.

Real-time capability is not a driving goal of the ROS system, but a module is available for real-time communication between nodes. For more serious hard real-time purposes, ROS has been used in conjunction with the Orocos RTT [12]. This approach had an Orocos real-time application communicating information to a non-real-time ROS application on which ROS's various visualization tools could be used. The success of this integration suggests that, for a real-time application, the Orocos Real Time Toolkit may be best used to form the core of the application, with ROS serving more appropriately as a beneficial add-on than as the fundamental framework.

[3] Orocos has a somewhat similar utility called the TaskBrowser, described in Chapter 2. The TaskBrowser does not make information available as to publishing rate, however.

For applications in which real-time performance is not as critical, however, ROS may be a very suitable choice by itself. ROS currently supports the Linux and Mac OS X operating systems, and the Python and C++ programming languages. It is released under a BSD license, allowing for both commercial and noncommercial use.

1.3.7 Summary and Architecture Selection

Table 1.1 summarizes the relative merits of each of the software platforms considered. This is intended only as an evaluation of the suitability of each software package for the present application, and not an attempt to assess overall merit for a general robotic system. Project Chimera was described above primarily as a historical reference, and is not considered here due to its apparent lack of development over the past decade and its dependence on the Motorola 68K processor. In keeping with the decision to consider only open source software, CLARAty is considered only on the basis of its publicly released code. The capabilities along the left side of the table are derived from the requirements presented in Section 1.2, and are listed roughly in order of decreasing importance. "Architecture" refers to the specification and enforcement of a design methodology to maximize modularity and ease development. "R. T. Ready" refers to the capability for writing hard real-time applications. "F.K. (Pos, Vel)" refers to the capability to perform forward kinematic calculations for a general serial link kinematic chain, separately indicating both position (i.e., from joint angles to Cartesian pose) and velocity (i.e., from joint rates to Cartesian translational and angular velocities). "Traj. Gen." refers to the ability to generate a sequence of set-points between an initial pose and a desired pose while respecting velocity and acceleration limits, in both Cartesian and joint space. "Devel.

Tools" refers to tools for examining the internal happenings of an application, examining the flow of data between software elements and providing a means of visualizing those data. "Ext. Interact." refers to the ready capability to interact (over Ethernet or a similarly generic standard) with external processes that were not developed within the same software framework. A fully blackened circle indicates that the capability is fully implemented and ready for basic use. An empty circle indicates that the capability is either completely absent or substantially lacking. A partially filled circle indicates an intermediate point between these two states, with an implementation that may be incomplete or ill-suited to the present application.

Orocos and ROS both present as strong contenders for use in the present application. Both offer the infrastructure for a component-based architecture. Both offer generic manipulator forward and inverse kinematics capabilities. Much of the appeal of ROS, however, comes from its integration with Orocos, which offers stronger real-time capabilities. Although hard real-time capabilities are not immediately required for the present application and are not developed in the present research, real-time extensibility is strongly desirable for potential future applications. For this reason, Orocos was selected to form the basis of the present application, with ROS viewed as a potential future add-on if its capabilities should be desired. The Orocos libraries relevant to the present work are described in greater detail in Section 2.1.

Table 1.1 Comparison of the suitability of various software platforms for the present application.

	Orocos	CLARAty	YARP	Player	ROS
Architecture	●	◒	●	◒	●
R.T. Ready	●	◒	●	◒	◒
F.K. (Pos, Vel)	● ●	● ○	○ ○	○ ○	● ●
I.K. (Pos, Vel)	● ●	○ ○	○ ○	○ ○	● ●
Traj. Gen.	●	◒	○	○	●
Devel. Tools	◒	○	◒	◒	●
Ext. Interact.	○	○	●	◒	◒

Chapter 2: Background

This chapter provides an overview of the basic software and hardware resources used in the present work. It begins with an overview of the relevant details of Orocos. It then describes the manipulator hardware used as a test platform for the present research, along with the corresponding software drivers. It concludes with a brief description of the measurement systems used to collect data for assessing system performance.

2.1 Overview of Orocos

Dissatisfied with their experiences using commercial software for advanced robotics research, a group of roboticists in December 2000 conceived of an open source, modular, reusable software library for robotic control. A proposal was submitted to the European Union, which sponsored the project for two years beginning in September 2001. The Katholieke Universiteit Leuven in Belgium, Laboratory for Analysis and Architecture of Systems Toulouse in France, and Royal Institute of Technology in Sweden were selected for this initial stage of development of the Open Robot Control Software (Orocos) [13]. Since that time, the Orocos project has resulted in four separate C++ libraries. The Real-Time Toolkit (RTT) provides the basic underlying framework of Orocos, establishing a generic infrastructure intended to support the widest possible variety of robotic systems. The Kinematics and Dynamics Library (KDL) provides capabilities for calculations relevant to serial-link kinematic chains. The Orocos Component Library (OCL) provides a selection of ready-to-use software units written within the framework established by the RTT. Lastly, the Bayesian Filtering Library (BFL) provides capabilities for information processing and estimation. The following subsections describe the RTT, OCL, and KDL. The BFL was not used in the present work.

2.1.1 Real Time Toolkit (RTT)

The Orocos framework established in the RTT is built around the concept of the component as the basic functional unit of an application. The level of granularity is at the discretion of the component builder, and a single component may represent as small or as large a unit of functionality as is appropriate for the application. The framework provides for lock-free, thread-safe interaction between these components, both synchronously and asynchronously. All components derive from the provided RTT::TaskContext class, and have a standardized interface for configuration, data flow, and execution flow. The interfaces of multiple components may then be connected to form an application consisting of a network of peer components. This subsection provides only a brief overview of the Orocos component-based design approach. Additional information with a focus on the details of implementation can be found in reference [14].

Components are equipped with five optional forms of public interface with their peers: Events, Attributes and Properties, Methods, Commands, and Data-Flow Ports. Events allow changes in the system to be broadcast to whatever other components may be listening for that event. Attributes and Properties specify configuration parameters that the component uses during its operation. Methods are essentially publicly available synchronous calls making a request of the component that can be fulfilled or rejected immediately, such as a requested calculation.

Commands are similar to Methods, but are used in situations in which the request cannot be completed immediately, as when the component is instructed to reach a goal. Commands are therefore executed asynchronously in the thread of the component receiving the command, and the caller is provided with the means to inquire as to whether the Command has been completed. Lastly, Data-Flow Ports represent the primary means of regular information exchange between components. These connections may be either buffered, in which case messages accumulate until read, or unbuffered, in which case a new message will overwrite any previous one.

A component has a small number of predefined states which govern its behavior. The three states of primary interest are PreOperational, Stopped, and Running. In the PreOperational and Stopped states, the component will not respond to Commands and Events. A transition from PreOperational to Stopped may be initiated by calling the component's configure function, and from Stopped to Running via the start function. Each transition has a corresponding hook function in which the component builder may specify instructions to be performed upon these transitions. The full state diagram is presented in Fig. 2.1. While in the Running state, the component's Execution Engine is responsible for managing algorithms which form the functionality of that component. This Execution Engine may be triggered either periodically or in response to the arrival of commands, events, or explicit trigger calls. When triggered, whether periodically or otherwise, the Execution Engine executes an update hook into which the component builder has written the component's primary run-time functionality. Most of the effort involved in writing a new Orocos component consists of populating these hook functions with the desired functionality.

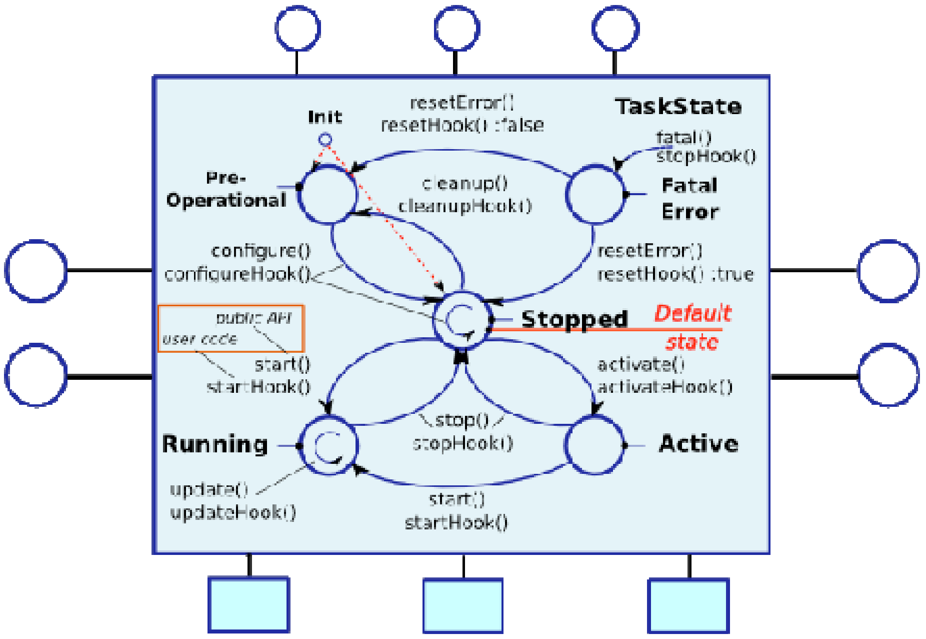


Figure 2.1 State diagram for an Orocos component. Pre-Operational, Stopped, and Running are the three basic states. From [14].

Components may also contain a finite state machine (RTT::StateMachine) representing a collection of states and corresponding actions to be taken on entry into each state, while running within the state, upon exit of that state, or in the event that no transition to another state is possible (specified as functions by the name of entry, run, exit, and handle, respectively). Conditional requirements for transition to other states are also defined. The StateMachine may run in either reactive or automatic mode. In the reactive mode, state transitions will occur only in response to an event or a request to change states. When activated, the StateMachine will default to reactive mode. The start() command will put it into automatic mode. In the automatic mode, the StateMachine will automatically transition to another state if any such transitions are legal. StateMachines are specified in an Orocos State

Descriptions file defining the four functions for each state and stipulating the requirements for transitions between states. These StateMachines may be used to tailor behavior to a more complex set of states than the simple five states available for the components themselves (PreOperational, Stopped, Running, etc.).

2.1.2 Orocos Component Library (OCL)

The OCL contains a selection of components that are written and ready for use in an application. This includes implementations of certain hardware such as firewire cameras and a laser ranging sensor, with varying degrees of reusability outside of the labs in which they were developed. Of greater interest for the present application is OCL's collection of motion control components, as well as its Task Browser and Deployment components.

The motion control components provide for trajectory generation and control, making use of some functionality from the KDL. It includes a selection of feedback controllers and trajectory generation for both joint space and Cartesian space applications. The position-trajectory generators output a sequence of waypoints along a trapezoidal trajectory obeying specified acceleration and velocity limits.

The Task Browser allows the user to browse the components within an application while it is running, providing the capability to inspect and interact with their interfaces. The user can determine the current state of the component (Stopped, Running, etc.) and request transitions between those states. He or she can call publicly available Methods or Commands. The Task Browser can also display and edit the configuration properties, as well as messages on the Data-Flow Ports. Its most glaring limitation is its relatively limited set of supported types. While simple types such as `int` and `double` can be readily examined and modified via this interface, more complicated types are not so easily accessible. The contents of a Data-Flow Port or Property with type `std::vector<double>` can be displayed, but to edit its value the user has available only a two-argument function which takes the length of the vector and a single value for all elements of that vector. Thus, a user wishing to change the value via this interface is constrained to having all of the elements be equal. While this is useful, for example, if one wishes to set an input to all zeros or to the empty vector, more complicated adjustments do not appear to be possible. Moreover, the present author has reported a bug in the Task Browser which causes a segmentation fault if the user connects to a component's Data-Flow Ports and then attempts to browse to a different component [15]. Nonetheless, the Task Browser is ultimately a very useful development and debugging tool because it immerses the user within the inner workings of the application, allowing much easier component-level access than would otherwise be available.

As with other C++ objects, Orocos components may either be compiled and linked directly into an executable binary for a given application, or built into a separate dynamically linked library which multiple programs may utilize. The OCL Deployment Component expands upon the versatility of the latter case by providing a means for automatically loading, configuring, and connecting components. With the addition of a few lines of code employing compiler macros provided with the OCL, one or more components may be compiled into a deployable library. The Deployment Component can then be instructed to import these deployable libraries, instantiate the desired components, specify the configuration properties, and establish the desired interconnections. These steps may be automated by preparing one or more xml files specifying all of this information. (Orocos will issue an error if all of the necessary configuration parameters for a component have not been specified.) In this way, the user can quickly and easily reconfigure for different applications without recompiling.

2.1.3 Kinematics and Dynamics Library (KDL)

The KDL provides capabilities for kinematic calculations involving serial-link manipulators. It defines classes for geometric primitives, including the vector and the rotation matrix. A three-dimensional transformation is represented by the KDL::Frame, which contains a vector for the displacement distance and a rotation matrix for the relative orientation. Functionality is provided for composition of transformations and for calculation of the inverse transformation. Twist and Wrench classes are also defined, and may be transformed via a Frame from one coordinate system to another.

The KDL defines a Joint class to represent each degree of freedom of the kinematic chain. There are seven types of joints: three representing rotations about each of the three principal coordinate axes, three representing translation along those axes, and a fixed joint which does not allow any movement. The Segment class represents a combination of a Joint with a Frame containing the transformation from the proximal to the distal end of the link. The KDL::Chain representing the serial kinematic chain for the robot is then built up from the appropriate number of Segments.

The KDL::Chain class representing the mechanism for which the kinematics solution is to be computed is limited to a serial chain of segments in which all of the joints may be actuated independently. What it lacks in generality, however, it makes up for in concrete implementation. In addition to providing abstract classes from which to derive a kinematics solver, the KDL also provides concrete solvers for forward and inverse kinematics which may be applied without modification to an arbitrary Chain. Support is also provided for a kinematic tree, KDL::Tree, which may have multiple endpoints of interest.

2.2 Hardware and Drivers

2.2.1 Manipulator Hardware

The Ranger Mark I manipulator was originally developed in the 1990s for use as a camera arm on the Ranger Neutral Buoyancy Vehicle (NBV) at the University of Maryland Space Systems Laboratory (SSL). It has six degrees of freedom, arranged roll-pitch-pitch-roll-pitch-roll. Fig. 2.2 shows Ranger's basic shape and depicts the coordinate frames assigned to each link. The base frame, denoted with subscript 0, remains fixed regardless of any actuation of the robot's joints and is the frame in which Cartesian commands are expressed. Table 2.1 gives the Denavit-Hartenberg

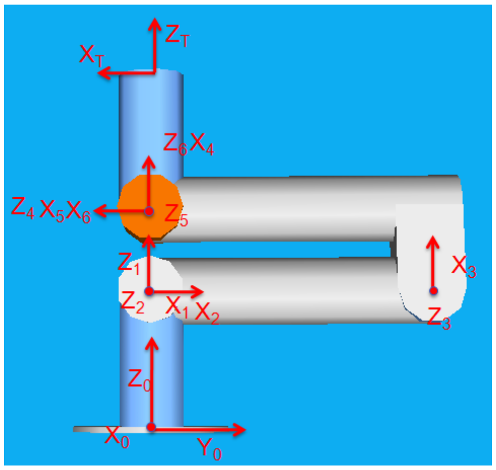


Figure 2.2 Computer rendering of Ranger Mark I with link frames drawn. From [16].

parameters[4] for Ranger, as measured by Ellsberry [16].

Table 2.1 Denavit-Hartenberg parameters for Ranger Mark I. From [16].

i	α_{i-1} (deg)	a_{i-1} (m)	d_i (m)	θ_i (deg)
1	0	0	0.2491	θ_1
2	90	0	0	θ_2
3	0	0.5589	0	θ_3
4	-90	0.1514	0.5388	θ_4
5	90	0	0	θ_5
6	90	0	0	θ_6
T	0	0	0.2666	0

In a recent overhaul, Ranger has been equipped with six SimplIQ Whistle digital servo drive controllers from Elmo Motion Control. The control hardware for this manipulator is the result of a separate research project; the rationale behind the selection of this hardware and the details of its implementation may be found in Reference 16. For the purpose of the present research, this hardware platform is simply a given. Each Whistle manages the servomotor for one of the robot's joints. The SimplIQ line of servo drive controllers is capable of motor current, velocity, and position control modes. Commands for the servo drives may be specified either by writing an onboard program in the SimplIQ drive language or, as in the present application, by sending commands from a host system via a supported communication interface. The Whistle supports both RS-232 and CANopen communication protocols, the latter of which is employed in the present work.

2.2.2 CAN bus and CANopen

This subsection describes the details of the Controller Area Network (CAN) that are relevant to the present research. The use of CAN is a requirement imposed

[4] Denavit-Hartenberg parameters are a common way of describing the kinematic arrangement of a manipulator. The parameters employed throughout this thesis conform to the convention presented in Reference 20.

by the control hardware described in subsection 2.2.1. A discussion of the merits of employing CAN in a robotic system may be found in Reference 16.

The CAN bus is a serial bus system originally developed in the 1980s to enable communication between devices without a host computer. Although originally designed for use in automobiles, the CAN bus quickly saw use in elevator systems and x-ray machines as well [17]. Nodes on this network may communicate at rates up to 1 megabit per second. The CAN standard defines a standard message structure, the CAN frame. This structure incorporates, among other things, an identification number for the message, a data field of up to 8 bytes, and a cyclic redundancy check. Arbitration of multiple nodes is provided. A message begins with a start-of-message signal followed by the message's identification number. If multiple nodes attempt to transmit at the same time, the message having the lower identification number will be given priority. This arbitration happens without delaying the highest-priority message because the zero bit is dominant in the CAN architecture—i.e., if any node is transmitting a zero, other nodes on that bus will read a zero regardless of any nodes which may be attempting to transmit a one [18]. Thus, if a node is attempting to transmit a one and yet reads a zero on the bus, it detects that it has lost the arbitration to a higher-priority message having more dominant bits at the beginning of the ID. The lower-priority node then ceases transmission and waits for the higher priority message to pass. To the other nodes on the network, there is no indication that the lower-priority node was ever transmitting. This capacity for prioritized communication is generally desirable when real-time communication is required. Although not a vital part of the control architecture developed in the present research, CAN hardware nicely complements the present architecture by providing a level of hardware standardization that enables even more extensive reuse of developed systems from one robot to the next.

The CANopen protocol provides a higher level communication protocol on top of the basic CAN specification. The CAN frame's identification number is divided into a message type identifier and a node identifier. The two message types of primary interest in the present application are Emergency Objects, which indicate an exception such as a motor undervolt, and Process Data Objects, which are used to relay commands and other real-time information to the nodes. A high-priority Synchronization Object is also available for triggering execution of tasks that are desired to begin simultaneously. Lower-priority Service Data Objects can be used for configuration of the node.

2.2.3 PCAN Interface and Driver

PEAK-System produces hardware and software for CAN applications. They offer a variety of different CAN interfaces for connecting a host computer to a CAN bus via USB, PCI, PCI Express, and a number of other methods. They also provide a Linux driver package, distributed under the GNU General Public License. This driver provides a single Application Programming Interface (API) for all supported CAN interface hardware, thus allowing for software to be written so as to be compatible with most PEAK-System CAN interfaces without modification. The API provides methods for opening and initializing the CAN interface, including specification of the desired data rate. Blocking and nonblocking methods are available for reading and writing to the CAN bus. Statistics are available as to the number of pending reads and writes in the buffer.

The research, development, and testing described in the present work relied primarily on a PEAK-System PCI card (PCAN-PCI) as the interface device between the host computer[5] and the CAN bus. The PEAK-System USB interface (PCAN-USB) was also used for limited testing, and was found to perform adequately in the present system.

2.2.4 SimplIQ Command Language

The SimplIQ Command Reference Manual [19] documents the available commands for operating a SimplIQ servo drive such as the Whistles used in the present application. These commands are grouped into a number of categories: motion, input/output, status, feedback, configuration, communication, control filters, protections, data recording, user programs, and general (miscellaneous) commands. These commands are available over both the RS-232 and CANopen interfaces, as well as within user-defined programs stored and executed onboard. The RS-232 and CANopen interfaces are available simultaneously so that, for example, one could use the RS-232 serial interface to check values and settings to debug interactions that are happening over the CANopen interface. By default, the Receive[6] Process Data Object 2 (RPDO2) message type, having message ID 0x300 + node ID, is used to send commands to the drive. The node replies with a Transmit Process Data Object 2 (TPDO2) message, having message ID 0x280 + node ID. To issue a command the

[5] A Dell Optiplex GX280 with a 3.6 GHz Intel Pentium 4 processor and 1 GB of memory, running Ubuntu 8.04 (Hardy Heron).
[6] The terms "receive" and "transmit" are defined from the node's perspective.

RPDO2 is structured with eight data bytes, the first two of which are ASCII-encoded characters indicating the desired command. The next two data bytes contain an index, and specify whether the last four bytes (which represent the numerical value being sent as an argument) form an integer or floating point number. If it is only desired to query a value rather than modify it, the data portion may be reduced to four bytes containing only the two ASCII characters and index.

The motion commands allow specification of relevant motion parameters, including maximum acceleration/deceleration values and smoothing factors. Depending upon the operational mode, commands are also available to specify a point-to-point motion (PA or PR), desired jogging velocity (JV), or desired motor current (TC). One command specifies the desired motion; and begin (BG) and stop (ST) commands are used to initiate and abort the motion. Another command (MO) turns the motor on and off. The SimplIQ commands for a simple point-to-point motion compute a trapezoidal trajectory so as to bring the motor to rest at the desired goal. This is not desirable in a situation in which waypoints along a continuous trajectory are fed to the controller on the fly, as in the present application. Separate configuration commands are provided for specifying trajectories, which one can initiate and abort using the same BG and ST commands. QP provides access to an array of position values, specified in encoder counts. A motion parameters array (MP) allows setting the time duration between waypoints, and specifies how to behave when upon reaching the end of the QP buffer. A slightly more advanced form of trajectory configuration is also available, in which both joint positions and rates are specified at each waypoint. In either event, the servo drive performs a third-order polynomial interpolation to join the waypoints with a smooth trajectory.

Feedback and status commands provide access to the feedback information that the servo drive has available. This includes the position (PX) and velocity (VX) as measured by the drive's encoder, as well as motor current (resolved into two components, ID and IQ, respectively perpendicular to and aligned with the rotor's magnetic direction). A temperature command (TI) returns the temperature of the controller module as measured by an onboard sensor, if available.

2.2.5 Faro Arm Coordinate Measurement Machine

The SSL is also in possession of a Faro Arm Platinum portable coordinate measurement machine consisting of a six-degree-of-freedom serial kinematic chain with high-precision encoders for determining joint angles. The Faro Arm, shown in Fig. 2.3, provides submillimeter-precision capabilities for determining the Cartesian coordinates of its end effector probe, which may be placed in contact with an object to be measured. An adapter is available for holding this probe in a fixed position relative to the last link of the Ranger manipulator. The Faro Arm may therefore be used to measure Ranger's three-dimensional end effector position coordinates as an independent measure of its static position accuracy.

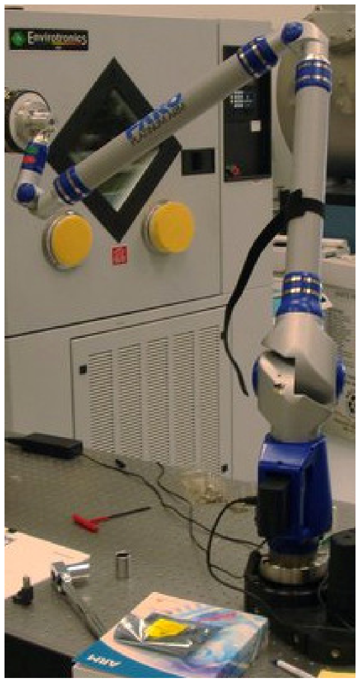


Figure 2.3 Faro Arm

2.2.6 Vicon Visual Tracking System

The University of Maryland's Autonomous Vehicle Laboratory (AVL) is equipped with a Vicon visual tracking system, which was made available to the present author. This system utilizes an arrangement of cameras surround by light-emitting diodes to track and record the motion of reflective spherical markers attached to an object, processing the two-dimensional position data recorded from each camera to yield a time-stamped history of that object's translational and rotational motion in space. AVL personnel informed the author that the system could be expected to report marker locations within an error of no more than 5 mm as calibrated during the present investigation. Because the system tracks reflective markers, it is necessary to cover any reflective surfaces which may already exist on the object to be tracked.

Chapter 3: Software Development

This chapter describes the software system that was developed in the present research. The goal here is to establish a top-level control system for a robotic manipulator, with the required kinematics and trajectory generation capabilities, while minimizing the effort required to port this software to a different robot system. By leveraging the open source efforts of Project Orocos, it is hoped that the cost of developing, debugging, and maintaining the present software system will be substantially reduced as compared to a completely custom-built system.

3.1 Overview

Fig. 3.1 shows the components which make up the present application. The command and data handling components (Ethernet Interface, Message Handler, and Logger) run in one thread, and are responsible for handling interactions with the client program and any information which needs to be logged. The kinematics and trajectory components run in a separate thread, and are responsible for producing the joint-level commands to be sent to the robot. The Robot Component is responsible for interaction over the CAN bus with the robot hardware. This component runs in its own thread at a higher frequency than any of the other components in order to relay commands and update information at the required rates.

The CMake cross-platform build automation system was used to manage the compilation of the software components in this application. This system enables the use of build configuration files that are independent of the system on which the code is compiled. This system then automatically generates makefiles for use with the GNU compiler collection in the Ubuntu Linux operating system used in the present application.

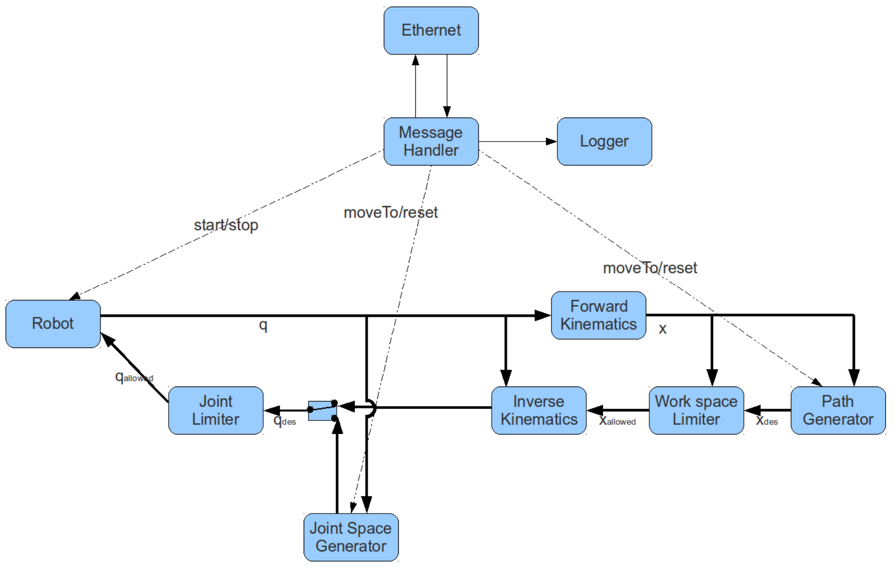


Figure 3.1 Overview of the software components and major interconnections in the present application.

During run time, messages are received via ethernet Internet Protocol from a client program. These messages are relayed to a message handling component, which processes them to identify the request that the client is making. If the message is a request for information, the Message Handler gathers the request information and sends it to the client. If the message is an instruction to be executed, the Message Handler initiates the appropriate commands and/or methods in the other components of the application, replying to the client to indicate success or failure of the instruction. A Cartesian point-to-point motion command, for example, will be relayed to the trajectory generation component. The Trajectory Generator will then begin sequentially outputting set-points lying on a straight line connecting the initial and goal positions, moving a step closer to the goal at each update cycle. After being checked by the Work Space Limiter to ensure that the set-point does not lie in a region marked as off-limits, these Cartesian set-points are passed through an inverse kinematics component which converts them to a joint space representation. This desired joint configuration then passes through the Joint Limiter, which ensures that each set-point does not violate any joint travel limits.[7] The allowed configuration then passes into the Robot component, which instructs the control hardware to execute the motion. Commands sent from the client in joint space bypass the earlier stages of this system, with the joint space trajectory generation component feeding directly[8] into the Joint Limiter.

In addition to the components which directly comprise the present application, additional software was written for validation of the system. This includes a simulation version of the Robot Component, as well as a collection of simple client programs for sending commands to the control application.

The following sections describe the developed software components and programs in greater detail, explaining the details of their interfaces and implementation as well as the design decisions that went into them. The source code may be found in Appendices B and C. Both the Cartesian and joint space trajectory generation components were taken directly from the Orocos Component Library and required no further development for the present research. The remaining components were written and developed during the present research by the present author. The forward and inverse kinematics components contain fully functional kinematic solver

[7] The Joint Limiter and Work Space Limiter individual inspect each set-point generated by the trajectory generation component. Their function and implementation are described in greater detail in Section 3.5.
[8] via the Switching Component described in Section 3.5

objects taken directly from the Orocos Kinematics and Dynamics Library (KDL) and used without modification. The development of the kinematics components for the present research consisted primarily of wrapping these objects within a component interface, assembling the kinematic chain according to KDL conventions, and (in the case of the inverse kinematic position solver) correcting undesired behavior[9]. The Ethernet component and the client programs employ communications objects developed by Stephen Roderick.

3.2 Robot Component

3.2.1 Purpose and Interface

The Robot Component (SSL::ElmoArm2) provides the interface between the control software and the robotic hardware. Its purpose is to communicate the joint-level commands generated by the control software to the servo drive controllers that implement them. The component acts as an Orocos wrapper for all the hardware-specific code that must inevitably be included somewhere within the control software. By encapsulating this functionality within a single component with a generic interface unencumbered by the details of the hardware, all of the code which would need to be modified to port this control application to another platform is cleanly isolated from the rest of the system. Thus, the effect of even a complete overhaul of the robot's internal electronics (or, equivalently, a transition to a different robot containing different electronics) would be limited to a single component.

[9] The divergence of the inverse kinematic solver near singular conditions, described in Section 3.3, could be viewed as a bug in the KDL solver, however the present author elected to correct it in the kinematic component rather than the solver object itself, though the technique used is readily transferable.

The first major design choice in preparing this component was the level of granularity at which the rest of the software would interact with the robot. A joint component representing a single joint could have been written, with as many copies instantiated in the control software as are necessary for the given robot. Because, however, performing the kinematic computations for a typical manipulator requires collectively considering all joint positions and rates, this would only serve to complicate the interface between the robot component(s) and the kinematic components. It was therefore decided that the Robot Component would be written such that a single instance of it would be responsible for all interaction with the servo drives.

The ElmoArm2 interface was also desired to be generic, ideally betraying no hint of hardware specificity that could trickle over to other components. In a manipulator control scenario, the commands to be sent to the robot typically consist of either desired joint rates, desired absolute joint positions, or desired motor currents (which closely relate to joint torques). The present work focused primarily on the situation of sending desired absolute joint positions, although support has been written for the other two modes as well. Thus, the essential elements of the interface are just two Data-Flow Ports. One port accepts as input to the component a vector of desired joint positions to be transmitted to the drive controllers. The other port provides as output from the component a vector of the current joint angles. The std::vector<double> is used as the data type on both ports for compatibility with components in the OCL (viz., the feedback controllers and Task Browser). In addition to these two ports for basic operation, the ability was desired to monitor basic status information to detect off-nominal conditions that could cause damage to the robot. The most obvious parameters to monitor for this purpose are the temperatures of the Whistle units. An additional output Data-Flow Port was therefore incorporated into the Robot Component providing a vector of measured drive temperatures. This information, however, is not utilized by any of the control components. Temperature information is only made available to a client program via the MessageHandler, which can run otherwise unaffected alongside a robot component that does not have the capability to report controller temperatures. Although this component is unavoidably tied to the Elmo SimplIQ servo drive controllers and PEAK-System CAN interface driver, an effort was made to remain as reusable as reasonably possible within those constraints. Thus, the component is configurable to accommodate any number of joints[10] arranged in any manner, requiring only that each joint be controlled by a digital servo drive from the Elmo SimplIQ line of digital servo drives[11] and that the host computer interact with those drives via a PEAK-System CAN interface.

In addition to the robot component itself, simulated robot components were developed to enable testing and validation of the other components of the application when the robot hardware is not available or not required. Initially, SSL::SimArm was developed as a very simple simulation that only copied the desired joint vector from its input port onto its output port, simulating a robot which instantaneously and

[10] Each joint's servo drive must have a unique node ID for the CAN system to function properly. The standard CAN message structure allows for 127 unique node ID numbers.
[11] Elmo's ExtrIQ line of digital servo drives, designed for use in extreme environments, conforms to the same communication protocols and command language as the SimplIQ line. ExtrIQ products may therefore also be operated with SSL::ElmoArm2.

perfectly tracked any input. A later version, SSL::SimArm_nAxes, uses an OCL trajectory generator to enforce maximum joint rates.

3.2.2 Implementation

During configuration (i.e., in the configureHook function), ElmoArm2 sends a message over the CAN bus informing all the nodes (servo drive controllers) to switch into operational mode so that they will respond to instructions. It then prepares an array of data structures that it uses to keep track of the latest information received from each joint, and queries each node for its operational mode (i.e., whether it is running a feedback loop to achieve desired position, speed, or motor current). If any joint fails to respond, the configuration will fail (i.e., configureHook returns false) and no further action will be taken.

The start-up procedure (in the startHook function) verifies that all nodes are in the correct operational mode. It requests the current position reading from each node and does not allow the component to start until all joint positions are known. Because the hardware employed in the present study does not provide absolute encoders for position measurements, zero encoder counts is assumed to correspond to a home position which is specified in the component's configuration properties. These positions are converted from encoder counts to radians via a conversion factor which is also a configuration parameter. A method is also provided for recalibrating the joint positions during run time by adjusting the zero reference position.

When operation first begins, the updateHook sets each node to operate in a cyclical mode, moving its read pointer back to the beginning of the buffer when it reaches the end. The time duration between waypoints is calculated on the basis of the period of the Robot component. The Whistles' set-point buffers are filled with several copies of the current position; and motion is initialized. The Robot Component then continues to send it new position waypoints to allow for continuous operation. Because the Whistles employ an onboard algorithm to smooth the trajectory, the Robot Component must maintain its write pointer ahead of the Whistle read pointer by several waypoints. This buffering results in a time delay (roughly 0.1 seconds as configured in the present testing) between writing the set-point to the buffer and actual motion to that set-point.

It was observed during testing that servo drives to which more than two messages were sent in rapid succession did not reliably respond to all of them. This is most likely due to a limitation of the drive controller's ability to store incoming messages. It appeared as though the first message received would be executed immediately. The second message would be stored for processing upon completion of the first. If a third message arrives while the first is still being processed, it appeared to overwrite the second message. ElmoArm2 therefore does not send more than two commands to each servo drive per cycle.

3.3 Kinematics Components

3.3.1 Purpose and Interface

The commands sent to the arm hardware via the Robot Component must be in joint space, but the higher level commands coming into the system are expressed in Cartesian space. Thus, one or more kinematics components are required to convert quantities between the two representations. Joint angles reported by the Robot Component must be converted into Cartesian positions (forward kinematics), and desired Cartesian movements must be converted into the corresponding joint angles or rates (inverse kinematics). To promote maximum reusability, none of the details of the robot geometry are written into the code itself. All kinematics components have configurable properties containing the Denavit-Hartenberg (D-H) parameters specifying the robot's geometry. The appropriate KDL::Chain is then constructed in the configureHook at run time.

In principle, both forward and inverse kinematics could be carried out within a single component, which would allow for all of the kinematic details of the robot to be managed neatly within a single component. However, as can be seen from examination of the flow of execution in Fig. 3.1, this would complicate the selection of the period at which this component operates. It would have to run once after Robot has updated its current position reading to perform the forward kinematics, and then again after the trajectory generation components to perform the inverse kinematics. This could be accomplished by triggering the kinematics component when new information arrives on a Data Port rather than running periodically; however, this still results in a scheduling arrangement that requires more careful consideration than would otherwise be necessary. It was instead decided to instantiate two separate kinematic components for this purpose. Separating these capabilities also allows for more easily modifying one without affecting the other—for example, switching from inverse position kinematics to inverse velocity kinematics—at the cost of some additional resource usage associated with instantiating two separate copies of the kinematic chain. The risk of unexpected behavior due to an erroneous discrepancy between the two Chains can be somewhat mitigated by writing an identical set of configuration properties for all components such that they may be configured from the same configuration parameters file.

3.3.2 Implementation

All of the kinematics components contain essentially the same configureHook. The KDL::Frame contains a function which can produce a Frame corresponding to a set of four D-H parameters describing the relative orientation between two coordinate frames, following the convention of [20]. The variable parameter (θ for a revolute joint, and d for a prismatic joint) is set to zero, and this Frame is followed with the appropriate KDL::Joint for either rotation or translation along the local z-axis. Thus, the combination of the variable transformation due to the Joint with the constant transformation supplied by the link is equivalent to the transformation between consecutive joint frames in Craig's convention. Because the KDL::Segment contains a joint at its proximal end and the link afterward, the first KDL::Segment is a fixed transformation and uses the fixed Joint type. Once the Chain has been built, it is used to initialize the appropriate kinematic solver(s) from the KDL.

With the exception of the inverse position kinematics component, SSL::InversePosKinematics, the start hooks of these components only resize the necessary arrays to have the appropriate size for the number of degrees of freedom of the robot (which is a configurable parameter). InversePosKinematics additionally attempts to locate a peer component named JointLimiter and, if successful, inquires as to any joint range-of-motion limits that must be respected.

During operation (updateHook), most of the work is done within the kinematic solvers provided by the KDL. The forward kinematics solver, KDL::ChainFkSolverPos_recursive, simply uses the current joint angles to compute the end effector position, starting from the base frame and recursively right-multiplying by the transformation from each link to the next along the chain. A forward-kinematic velocity solver is available within the KDL but was not required for the present implementation. Inverse position kinematics are provided by KDL::ChainIkSolverPos_NR_JL, which uses an iterative Newton-Raphson technique along with an inverse velocity kinematics solver. An implementation of inverse velocity kinematics is available in the form of a KDL solver which uses the Jacobian pseudoinverse[12]. This solver is both used within the iterative inverse position kinematics component and, although not required for the present application, packaged into a separate component for potential future use. These KDL kinematic solvers employ the KDL::JntArray rather than the std::vector<double> for representing the joint space configuration of the robot. This introduces an unfortunate inconsistency of interface between the kinematic solvers and the OCL components that drove the use of the std::vector<double> in the present application. The kinematics components therefore must internally convert back and forth between the two.

It was observed during testing in simulation that the iterative inverse kinematic solver employed may diverge from the desired goal if in a near-singular configuration. After successfully maneuvering through a sequence of waypoints, the simulated robot was commanded to a goal that was not within its reachable workspace. The resulting trajectory is shown in the left of Fig. 3.2. The robot moved

[12] The Jacobian pseudoinverse, $J^{\dagger} = J^T(JJ^T)^{-1}$, becomes the common inverse for a square Jacobian matrix but is applicable to manipulators having non-square Jacobians as well.

in an approximately straight line toward the goal until it approached full extension. The inverse kinematic solution then diverged, causing the robot to chaotically move further from the desired position. For this reason, a solution check was written into the inverse kinematics component to calculate how far[13] the inverse kinematic solution is from the desired Cartesian position. If the inverse kinematic solution is further from the goal than the initial guess that was given to the solver, the new solution will be discarded. The right side of Fig. 3.2 shows the same sequence of waypoints executed with this safety check in place. The robot approaches the goal approximately as nearly as it is able, coming to a stop when the inverse kinematics fail to produce a solution which is nearer to the goal than its current position.

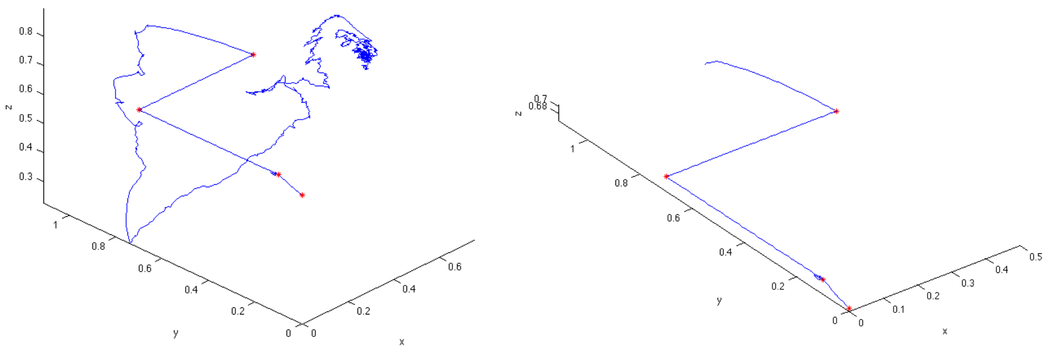


Figure 3.2 End effector trajectory in space, showing inverse kinematic divergence near full extension (left). A system to detect and manage this was implemented (right). Red asterisks denote the commanded waypoints. The final waypoint, (-0.5, 4.0, 0.7), is beyond the bounds of the graph and hence is not shown. Dimensions in meters.

[13] The distance metric used here is the Cartesian straight-line distance (in meters) plus the angular orientation difference (in radians) between the desired pose and the pose resulting from the inverse kinematics solution.

3.4 Command and Data Handling Components

3.4.1 Purpose and Interface

A system is needed for handling incoming instructions through a client interface and commanding the relevant components on the basis of those instructions. It was desired that this interface being as simple as reasonably possible, preferably enabling the client to issue commands to the manipulator without requiring the use of Orocos on the client side. For this purpose, a simple instruction language was designed for interaction with a client program. A message consists of two ASCII-encoded characters indicating a command and an additional 29 bytes for any corresponding data, for a total of 31 bytes. The 29-byte data portion of a message is sufficient to carry seven floating-point or integer numbers with one additional byte left over for other use (e.g., as an index). The full list of available command instructions is given in Appendix A. These commands provide the client program with the ability to specify a goal position (in joint space or Cartesian space), to initiate motion toward that goal, to stop motion, and to query the current end effector Cartesian position and joint angles, among other things. To allow for maximum versatility, the interpretation of these commands was decoupled from the communication mechanism. Thus, one component (SSL::MessageHandler) acts to interpret incoming messages, issue the appropriate commands, and prepare response messages to be sent to the client. Another component (SSL::EthernetInterface) acts to relay these messages between the control application and the client program, in this case via User Datagram Protocol (UDP). In this way, one could switch to a completely different communication protocol by simply writing a new component to relay messages between its Data-Flow ports and this alternate communication channel.

In addition to interpreting and distributing commands, a system is needed for logging relevant data for debugging and other analysis. Orocos is equipped with a system for logging information, but writes it all to a single orocos.log file in a manner more suitable for logging run time warnings and errors than large amounts of data. It is instead desirable for the present application to have multiple log files (for Cartesian position, joint angles, incoming commands, etc.). Because the data to be logged will vary from one component to the next, and from one application to the next, it was decided to give the component producing the log data full control over the content of each log entry, with the logging component (SSL::FileLogger) acting only as a simple mechanism for gathering these messages and writing them to the appropriate files.

3.4.2 Implementation

In contrast to other components within this application, the transmission of a new message over one of the Data-Flow Ports connecting the command and data handling (CDH) components does not render any previous messages obsolete. Whereas the kinematics components, for example, need only concern themselves with the most recently measured position of the robot, CDH components must not discard commands to be processed or information to be logged. For this reason, the Data-Flow Ports connecting one CDH component to another are buffered; these components execute a loop within their respective update hook until they have processed all awaiting messages.

SSL::MessageHandler needs to interact with a number of other components in the application in order to carry out the commands received from the client program. These other components must be set up as peers of MessageHandler so that it can access their interfaces. This can be accomplished by specifying the appropriate peers in the xml configuration file from which OCL::Deployer determines the components and configuration information to be loaded. MessageHandler::configureHook() then searches among MessageHandler's peers for several mandatory and optional peers, and retrieves pointers to the necessary Commands, Methods, and Events within those peers. For example, the Robot Component is a mandatory peer whose start and stop Methods must be available to the MessageHandler to allow for disabling and enabling control of the hardware. If this component is not found, configuration of the MessageHandler will fail. In contrast, the inverse kinematics component is optional. MessageHandler attempts to locate it during configuration and, if successful, connects to its divergence event (an event which the inverse kinematics component emits to indicate that it failed to converge to a solution). If such a component is not found, however, MessageHandler produces a warning but still allows the configuration to proceed. MessageHandler::startHook() then simply initializes some arrays and vectors to the appropriate sizes and sets variables to appropriate starting values.

MessageHandler::updateHook() checks for any warning conditions that need to be relayed to the client program. It then enters a loop which pops a message from the incoming message buffer, sends a copy of the message (converted to an ASCII string) to the logging component, and uses nested switch statements to determine which code to execute on the basis of the two command characters. After that loop, it logs any additional information that may need to be logged. The Cartesian position, joint space configuration, and desired Cartesian set point can then be logged. Counters are used to log these values only once in every several cycles to avoid generating excessive log data.

The FileLogger configuration hook call creates/opens a number of log files having file names of the form log#.dat, where # is an integer ranging from one to numLogs, a configuration parameter specifying how many log files are required. A pointer to each of these files is stored in an array. If any of these files fail to open, configureHook returns false and the configuration fails. No functionality is written into startHook() because no further action is necessary to prepare the log files. The update hook of the logging component pops a message from its incoming Data-Flow port buffer, writes it to the appropriate log file. If there is more than one message in the buffer, it loops until the buffer is empty. Writing to the hard drive in this manner requires an inherently unpredictable amount of time to complete. For this reason FileLogger runs within its own thread, separated from every other component in the application so that performance of the CDH and control components will not be impaired if FileLogger blocks. If it blocks for an extended period, its buffer can become full and messages that were intended to be logged will be lost, but the performance of the remainder of the application will be unaffected.

3.5 Trajectory Generation and Management Components

3.5.1 Purpose and Interface

Motion instructions received from the client consist of only a single goal position or a moderately distantly spaced sequence of several waypoints. The Robot

Component, however, requires a continuously updating stream of desired positions. A trajectory generation system must therefore be implemented to produce a more tightly space sequence of set points, smoothly moving the robot from the starting position to the goal position. Fortunately, the Orocos Component Library (OCL) already contains components written for precisely this purpose. OCL::CartesianGeneratorPos has an input Data-Flow Port for the current measured end effector position, and output ports for both desired position and desired velocity in Cartesian space. When instructed via a Command, it begins outputting a sequence of setpoints following a trapezoidal trajectory from the current position (measured on the input Data-Flow Port) to the goal position within configured velocity and acceleration limits. OCL::nAxesGeneratorPos does the same, but operates in joint space rather than Cartesian Space.

As a safety check on the system, components were desired to provide basic constraints to prevent motion which could damage the arm. Attempting to drive a joint beyond its allowable range of motion, for example, could cause it to rip its internal wiring (for joints without a hard stop) or overheat due to excess current (for joints with a hard stop). The ability to specify basic exclusion zones in Cartesian space into which the end effector is not allowed to move enables a basic level of protection against collision with the environment. For these reasons, SSL::JointLimiter and SSL::WsLimiterCart were written to enforce these disallowed configurations in joint space and Cartesian space, respectively.

To enable switching between joint space and Cartesian space operating modes, SSL::ControlSwitch was written so that both a Cartesian and joint space trajectory generator could be instantiated and that only one of them would command the Robot Component at any time. ControlSwitch has two input ports, taking the desired joint space set point directly from the joint space trajectory generator and the desired Cartesian set point from the inverse kinematics component. Thus, both inputs are vectors in joint space, and it is only a matter of selecting which of them to copy to its output port, which feeds into the Robot Component via the joint limiter. In addition to selecting which of its inputs is to be passed to the robot, ControlSwitch also manages the starting and stopping of both trajectory generators and the inverse kinematics so that these components do not run when they are not needed.

3.5.2 Implementation

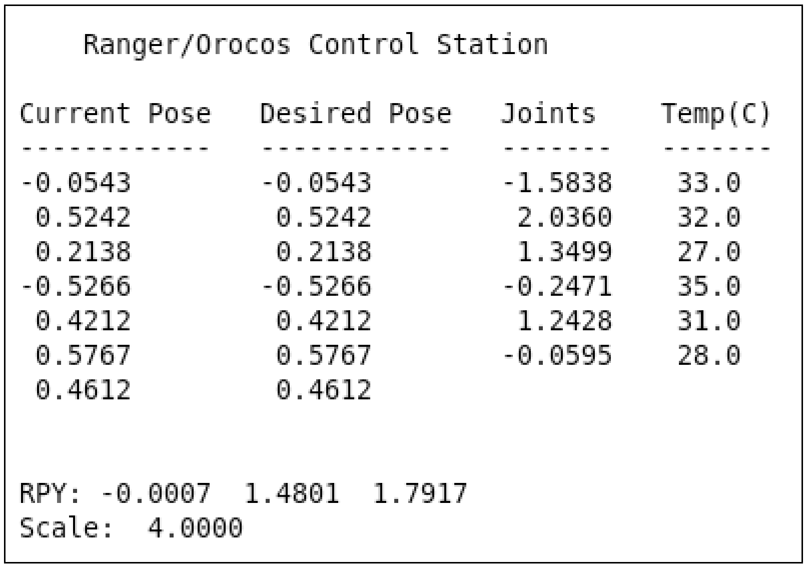
There are two basic ways in which motion limitations could be enforced. The motion-limiting component could either bring the robot to a halt if it is in violation of motion limits, or it could simply suppress the offending component of the motion. Joint range-of-motion limitations are sufficiently straightforward that the latter solution is feasible, with SSL::JointLimiter simply substituting the appropriate maximum or minimum joint value (specified in the component's configuration parameters) when the desired joint-space vector exceeds a limit on a particular joint. The hope here is that, although this will cause a deviation from the desired path, this deviation may be temporary and not warrant aborting the motion. In this case, a warning event is generated which is relayed to the client. Cartesian obstacles, on the other hand, represent more complicated limitations that cannot be neatly implemented as simple upper and lower bounds on a single workspace coordinate. They are represented in the workspace limiter's configuration parameters as box-shaped exclusion zones with principal axes aligned with the robot base frame's coordinate axes. Obstacle avoidance of this nature is really a path planning problem which would be more appropriately handled in the trajectory generation component. SSL::WsLimiterCart therefore resets the trajectory generator when an exception to its motion limitations is detected, aborting the motion and resulting in an error message to the client. WsLimiterCart will suppress any illegal waypoints before they reach the Robot Component, thus allowing further motion in a legal direction. If, however, the robot enters an exclusion zone due to imperfect tracking, WsLimiterCart will bring the robot to a stop within this exclusion zone and will not allow further Cartesian-specified motion. In joint space operations, such a workspace limit violation will still be reported to the client, however motion will not be aborted. The user may therefore guide the robot in joint space until it is clear of the exclusion zone, at which point Cartesian control may be re-enabled.

SSL::ControlSwitch utilizes a finite state machine (RTT::StateMachine) to manage its state, ensuring that only one of the two operating modes (Cartesian and joint space) is running at any given time, and that transitions between the modes are handled properly. Upon activation, the StateMachine begins in an initial state which immediately transitions to one of three states depending upon the mode setting (Disconnected, Cartesian, or Joint Space). The entry and exit functions for each of these states ensure that the appropriate trajectory generator (and, in the Cartesian state, the inverse kinematics component) is enabled and disabled. While in principle it would do no harm to leave these components running unnecessarily while not in use, it represents unnecessary computation. Furthermore, disabling trajectory generators that are not in use causes the MessageHandler to send an error to the client in the event that a command is specified in the wrong space for a given mode. For example, when operating in joint space, commanding motion to a Cartesian goal will return an error rather than performing all of the trajectory generation and inverse kinematic calculations, only to have the results quietly suppressed at the ControlSwitch. Starting and stopping these components breaks other components' access to their Methods and Commands (except for the start and stop methods, which are available even when the component is not running). While this is desirable when the components are not in use, it requires that the MessageHandler reestablish access when they are started again. ControlSwitch uses only the start and stop methods of these components, and so is not affected by this.

ControlSwitch::configureHook() locates the two trajectory generators and establishes access to their start and stop methods. The startHook() loads the StateMachine which manages the switch and sets the control mode Attribute to zero (which represents the Disconnected state). The first time updateHook() executes, it activates the state machine and sets it to automatic mode. After this, the updateHook() simply copies the appropriate input (or, in the Disconnected state, an empty vector) to its output, which passes through the joint limiting component to the robot. MessageHandler may trigger a change of state by changing the value of ControlSwitch's mode Attribute. During the next run of updateHook(), the StateMachine will then determine that a transition to one of the other states is valid, and will execute the appropriate functions. To ensure that the appropriate trajectory generator has had an opportunity to reset to the current position, ControlSwitch does not pass along the desired joint vector for two cycles after the transition.

3.6 Command Station Client

To enable user operation of the manipulator during development, a collection of software clients were written to relay user commands over ethernet to the control application. A low-level client allows for text-based entry of the commands as given in Appendix A. Because the user manually enters the command characters and data values, this provides access to the full set of supported commands but is cumbersome for basic operations. For this reason, two clients were written (one for Cartesian space and one for joint space) to enable basic motion commands via a single keypress and to display basic status information. Fig. 3.3 shows the Cartesian-space version of the client. The arrow keys and page up/page down keys are used to increment and decrement the x, y, and z coordinates of the desired end effector position. The plus and minus keys adjust the scaling factor which determines how much the arm moves in response to each key press. The space bar resets the desired position to the current position. The control software does not require continuous contact with the client program, so the user may start and stop the client as many times as desired, and switch between the available clients as needed. This latter pair of client programs will also display any warnings or errors received from the control software. These clients were written primarily to aid in development and testing of the control software; a more sophisticated user interface will have to be developed as the system sees more extensive use.



```
    Ranger/Orocos Control Station

Current Pose   Desired Pose   Joints    Temp(C)
------------   ------------   -------   -------
-0.0543        -0.0543        -1.5838    33.0
 0.5242         0.5242         2.0360    32.0
 0.2138         0.2138         1.3499    27.0
-0.5266        -0.5266        -0.2471    35.0
 0.4212         0.4212         1.2428    31.0
 0.5767         0.5767        -0.0595    28.0
 0.4612         0.4612


RPY: -0.0007  1.4801  1.7917
Scale:  4.0000
```

Figure 3.3 Cartesian client user interface. Errors and warnings (not shown) are printed on the lines below the above information. Pose values consist of translational coordinates (x,y,z) and unit quaternion.

Chapter 4: Demonstration and Testing

The following section describes an optimization technique used throughout the analyses presented in this chapter. The remainder of the chapter then describes demonstration and testing of the developed control application, both in simulation and on the hardware platform described in Section 2.2. The goal here is to assess whether the system meets the requirements set out in Section 1.2. The static positional accuracy test serves to validate the basic functionality to drive the robot to a desired Cartesian position. The trajectory tracking tests are designed to assess the performance of the system more thoroughly, investigating dynamic performance in an effort to characterize the system's capabilities. Simulations are then performed to demonstrate the adaptability of the system to a variety of kinematic configurations.

4.1 A least squares optimization technique

The Nelder-Mead optimization method employs a simplex-based iterative search for the minimum of an objective function of multiple variables [21]. The simplex is an n-dimensional geometric figure having n+1 vertices.[14] Such a collection of vertices is constructed in the objective function's input space, and the objective function is evaluated at each of these vertices. At each iteration, the algorithm performs one of four operations to select possible new vertices for the simplex. Vertices with higher objective function values are discarded in favor of new ones with lower values. The Nelder-Mead algorithm is implemented in the

[14] Thus, in the two-dimensional case the simplex is a triangle.

MATLAB function fminsearch, included in the optimization toolbox. Because this search method does not require any information about the derivatives of the objective function, it is especially well suited to nonlinear optimization problems.

When independent measurements of the end effector location are available, they may be compared to end effector positions computed using the forward kinematics model. An objective function quantifying the accuracy of the kinematics model may then be defined as the sum squared error between expected and measured end effector coordinates:

$$sse = \sum_i \left[(x_{K,i} - x_{M,i})^2 + (y_{K,i} - y_{M,i})^2 + (z_{K,i} - z_{M,i})^2 \right]$$

where ($x_{K,i}$, $y_{K,i}$, $z_{K,i}$) are the expected end effector coordinates as computed using the forward kinematics model and ($x_{M,i}$, $y_{M,i}$, $z_{M,i}$) are the independently measured coordinates for the i^{th} datum point. An analytical form of the forward kinematics, derived in Mathematica using formulas from [20], was used to compute the expected coordinates. This objective function was used along with the Nelder-Mead optimization method for all least-squares optimizations mentioned in the remainder of this chapter.

4.2 Initial Calibration

Ellsberry [16] has used the Faro Arm to measure the manipulator's dimensions and to provide encoder calibration data. For the latter purpose, he swept several links through a known quantity of encoder counts, using the Faro Arm software to determine the arclength and radius of that motion. Dividing arclength by radius gives the angle swept in radians. His data suggest 130,148 counts/radian for the first two joints and 65,929 counts/radian for the third joint. Due to hardware similarity, it is assumed that all subsequent joints are identical in this regard to the third. These calibration settings were used in the static positioning test described in the following section.

4.3 Static Positional Accuracy

4.3.1 Set Up

Ranger and the Faro Arm were mounted facing one another on an optical bench as shown in Fig. 4.1. The optical bench was assumed to be a sufficiently stiff mounting surface so as to minimize relative motion between the two arms. This arrangement allows for a substantial region of overlap between the Ranger and Faro work spaces while reducing the likelihood of damage to the Faro Arm in the event of a Ranger malfunction. Preliminary preparations for the static positioning test brought the arms slowly through the desired waypoints to verify that the motions could be executed safely without either arm approaching a joint range-of-motion limit or singularity.

The Faro Arm was calibrated via the procedure specified by the manufacturer, which involves placing the Faro end effector into a fixed mount, and moving the joints of the arm while the mount ensures that the end effector does not translate.

The Ranger arm was placed into the home position shown in Fig. 2.2 as precisely as possible through purely visual inspection. Because Ranger is not equipped with any means of measuring its absolute joint angles, any error in this starting position will have propagated through to all future poses during the data collection.

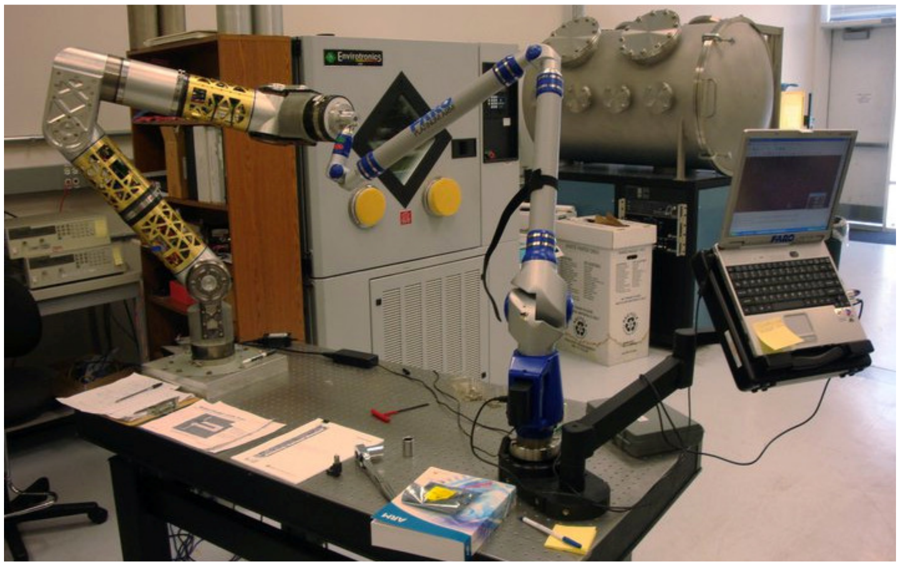


Figure 4.1 Hardware arrangement for positional accuracy testing.

4.3.2 Test Trajectory

A test trajectory was chosen to resemble the trajectory used to assess the static positional accuracy of a later-generation Ranger arm, presented in Reference 22. The test trajectory lies in a plane angled approximately 55 degrees above the horizontal, with the waypoints of this trajectory lying along two lines passing horizontally through the robot's workspace. Each line contained four waypoints spaced 20 cm apart along the line. The two lines themselves were spaced 10 cm apart, as shown in Fig. 4.2. The test trajectory consisted of 10 vertices which were visited multiple times with a constant end effector attitude in order to collect 50 datum points for analysis. Reference 22 demonstrated a static positional accuracy of 22.8 mm for the manipulator tested. Because a similar testing procedure is used presently, this performance figure of merit represents a reasonable baseline for comparison between the two systems.[15]

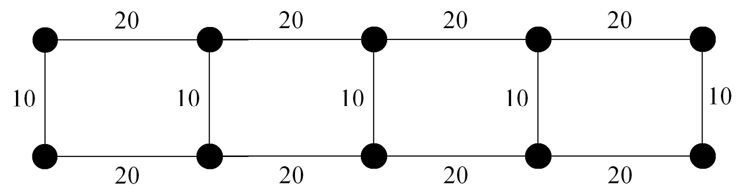


Figure 4.2 Test path for static positional accuracy. The horizontal direction in this figure coincides with the x-axis of Ranger's base frame. The vertical direction in this figure is oriented at a slope of $\sqrt{2}$ above Ranger's x-y plane. Dimensions in centimeters.

4.3.3 Analysis

A least-squares match was performed to determine the transformation between the Faro Arm coordinate frame and the Ranger base frame. All Faro measurements were then converted into the Ranger base frame, in which all subsequent analysis was performed. Due to an offset between the resolution point of the Faro Arm measurement probe and the resolution point of the final Ranger link, the forward kinematics for the Ranger arm were recomputed with this additional transformation in place. The results described in the following subsection are therefore the error between the predicted position of the Faro tip on the basis of Ranger's kinematic model, and the actual location of the Faro tip as determined by

[15] The cited test conformed approximately but not strictly to the standardized testing procedure established in Reference 23. This ANSI standard calls for an eccentric load attached to the end effector, a requirement not followed in the present investigation and similarly ignored in the cited test. The present investigation further ignored the requirement to begin testing from a cold start condition due to required preparation time. Ambient temperature and humidity were not noted. The order in which the points were visited was not fully randomized. The present study also held the end effector at a level orientation rather than perpendicular to the test plane due to difficulties presented by the placement of the Faro arm.

the Faro measurement system[16]. The distance between each pair of predicted and measured locations was then calculated as

$$d_i = \sqrt{\left(x_{p,i} - x_{m,i}\right)^2 + \left(y_{p,i} - y_{m,i}\right)^2 + \left(z_{p,i} - z_{m,i}\right)^2}$$

where *i* is the index for that pair of values, the subscript *p* indicates the predicted coordinate, and the subscript *m* indicates the measured coordinate. The mean positional accuracy and its standard deviation are then calculated as

$$\bar{d}_{PA} = \frac{1}{N}\sum_{i=1}^{N} d_i \quad \text{and} \quad S_{PA} = \sqrt{\frac{\sum_{i=1}^{N}\left(d_i - \bar{d}_{PA}\right)^2}{N-1}}$$

where N = 50 is the total number of datum points collected. These formulas are taken from Reference 23.

4.3.4 Results

The mean positional accuracy was found to be 5.1 mm with a standard deviation of 1.8 mm. This represents a more than acceptable level of accuracy, at approximately one-fourth the error of the later-generation Ranger arm assessed in Reference 22. These results demonstrate the basic validity of the kinematic calculations performed in the control application, as well as the capability of the system to perform point-to-point motion. Fig. 4.3 shows a plot of the test data, indicating the end effector position predicted by the kinematics as well as the

[16] The accuracy values presented as the results of this analysis therefore quantify only the error in the forward kinematics model—i.e., the error between where the control application believed the Ranger end effector to be and where it actually was, not the error between the *commanded* and actual positions. In all cases, however, the end effector position reported by the control application agreed with the commanded position to within several micrometers in each principal axis direction. This small discrepancy is most likely the result of the error tolerance specified in the inverse kinematic solver object as 10^{-6} meters. A smaller tolerance could be employed if greater accuracy is desired and sufficient computation time is available.

independent measurement from the Faro arm. As a general trend, the Faro data lie toward the outside of the test trajectory (in the x-direction) as compared to the kinematic predictions, hinting at a possible systematic error in the kinematics model. The following section explores the possibility of correcting for any such bias.

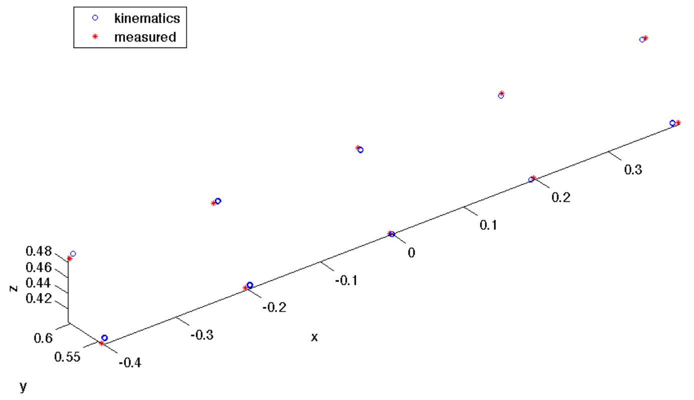


Figure 4.3 Static positional test data. Dimensions in meters.

4.4 Exploration of a Possible Refinement Approach

Although the performance determined in the previous section is more than satisfactory, Ranger's performance during that test was undoubtedly degraded by error in the joint angles. Because Ranger has no means of measuring its absolute joint positions, the joint angle values used in the control software suffer from a constant offset due to error in the presumed starting position. This section describes efforts undertaken to identify this offset for the static positioning testing session of the previous section[17] through numerical optimization. Consideration was also given to the possibility of error in the Denavit-Hartenberg parameters describing the robot's physical geometry as well (link lengths and offsets). Although the test employed the

[17] These angular offsets will in general vary from one testing session to the next because no system is yet implemented for holding the physical arm precisely in position between runs.

D-H parameters measured by Ellsberry to a high precision via the Faro arm, the offset of the Faro positioning tip was estimated using only a tape measure. Refinement of these parameters has the potential to improve Cartesian positional accuracy of the system. Although this is presently an off-line optimization technique requiring the use of MATLAB, it provides a demonstration of concept which could lead to the development of a calibration technique for the manipulator.

4.4.1 Analysis

The static positioning test described in the previous section yielded a data set containing a set of joint angles corresponding to various robot positions, and an independent measurement via the Faro Arm of the end effector positions for each of those joint configurations. This information was also collected at additional points not within the standard test plane in order to provide further data for the present parameter refinement effort.

Because the goal is to identify the robot kinematics in a form that can be utilized in the present control application, the parameters that are available to be varied are the 21 Denavit-Hartenberg parameters required to describe the mechanical linkages of a six-degree-of-freedom manipulator.[18] Uncertainty in the starting positions of the joints introduces a further six unknowns in the form of constant offsets to the joint angles. A further six parameters could be introduced to accommodate the unknown transformation between the Faro coordinate frame and the robot's base frame. This approach, however, quickly results in an objective function

[18] There are additional effects that introduce error into the kinematic model, including the effect of link flexibility. These effects, however, are not supported by the present kinematics components. The goal here is to refine the parameters used by these components.

having a great many input parameters. In the interest of producing a tractable optimization problem, only the five nonzero length parameters describing the robot geometry were treated as free variables. An additional parameter was added to account for the unknown offset between the Faro tip and the axis of Ranger's last link. These are combined with the six unknown joint offsets to yield an optimization problem in twelve variables. Rather than attempting to optimize all twelve parameters simultaneously, the problem is initially divided into two subproblems: first optimizing only the joint offsets, and then optimizing only the length parameters. As described in the following subsection, one of the twelve parameters was fixed in order to achieve convergence to a set of values which the present author deemed reasonable.

4.4.2 Results

Although the angle of the offset between the Faro tip and Ranger's tip was not measured precisely, optimization efforts allowing this to vary as a parameter produced results which were not consistent with the author's visual estimation of the value during testing. This is likely an especially difficult parameter to optimize because the offset between the two tips is so small that its effect is not particularly pronounced compared to the other parameters. The author therefore fixed this angle at 1.3 radians (approximately 75°) from the negative x-axis of the end effector frame. Optimization of only the five remaining joint angles yielded offsets of less than one degree, an easily believable magnitude given that the arm was initially aligned only by visual inspection. With these adjustments to the joint angles, the mean positional accuracy for the test trajectory of the previous section is reduced to 3.1 mm with a standard deviation of 1.4 mm, an improvement of nearly 40% over the uncorrected results.

Optimization of the D-H length parameters (with the angular offsets held constant at their separately optimized values) yielded the parameters given in Table 4.1. The optimized parameters agree to within a few millimeters with the original parameters measured by Ellsberry. This optimization again results in a modest improvement of the mean positional error, which becomes 1.9 mm with a standard deviation of 0.7 mm with these modified parameters.

Table 4.1 Separately optimized Denavit-Hartenberg parameters for Ranger Mark I. The large offset on θ_6 is due to the fact that the angular orientation of the Faro-Ranger tip offset was not measured and was determined only through optimization. The parameter a_6 was introduced to account for the Faro-Ranger tip offset. The offsets given for the θ parameters will not persist to any future runs. Optimized values are shown in bold.

i	α_{i-1} (rad)	a_{i-1} (m)	d_i (m)	θ_i (rad)
1	0	0	**0.2477**	θ_1 **– 0.001197**
2	$\pi/2$	0	0	θ_2 **+ 0.001839**
3	0	**0.5616**	0	θ_3 **+ 0.004736**
4	$-\pi/2$	**0.1489**	**0.5366**	θ_4 **– 0.003007**
5	$\pi/2$	0	0	θ_5 **– 0.01497**
6	$\pi/2$	0	0	$\theta_6 + 1.3$
T	0	**-0.003229**	**0.3118**	0

Simultaneous optimization of all eleven free parameters, using the results of the previous two subproblems as the initial guess, actually produces a slight increase in positional error.[19] Table 4.2 shows the fully optimized parameters, which yield a mean positional error of 2.0 mm with a standard deviation of 0.5 mm. The values again come to within a few millimeters of Ellsberry's measurements.

[19] Recall that the optimization was performed on a superset of the data used to evaluate mean positional accuracy, which is dictated by a desire to use a comparable test path to the later-generation Ranger evaluation.

Table 4.2 Simultaneously optimized Denavit-Hartenberg parameters for Ranger Mark I.

i	α_{i-1} (rad)	a_{i-1} (m)	d_i (m)	θ_i (rad)
1	0	0	**0.2505**	θ_1 **– 0.002627**
2	$\pi/2$	0	0	θ_2 **+ 0.0008630**
3	0	**0.5599**	0	θ_3 **+ 0.01267**
4	$-\pi/2$	**0.1525**	**0.5335**	θ_4 **– 0.006912**
5	$\pi/2$	0	0	θ_5 **– 0.006189**
6	$\pi/2$	0	0	$\theta_6 + 1.3$
T	0	**-0.00438**	**0.3122**	0

It is difficult to draw any meaningful conclusions as to the accuracy of the results of these optimization runs. The first five joint offsets represent a reasonable consistency between the separate and simultaneous optimization runs, suggesting that there is likely some validity to the corrected values. The optimization of the length parameters is a bit more questionable. The D-H parameters were already known reasonably precisely at the outset of this investigation. At the level of precision at which this effort attempts to optimize the values, the dominant source of error may no longer be D-H parameters themselves but instead the nonrigidity of the links, representing a breakdown of a central assumption of the kinematic model. These optimized parameters may therefore represent an effective average set of D-H parameters over the range of poses visited in the present investigation. In any event, these results suggest that the 5 mm accuracy determined in Section 4.3 is not a hard limit of the system's capability.

4.5 Trajectory Tracking Performance

While the static positional accuracy test of Section 4.3 confirms that the robot can eventually be driven to within a reasonable tolerance of a desired position, the path via which the robot approaches that goal is also of interest. In its Cartesian operating mode, the robot nominally follows a straight-line trapezoidal trajectory to the goal. This section describes testing undertaken to assess the accuracy with which the system tracks this trajectory under the present hardware implementation. Circular and sinusoidal trajectory tracking is also assessed. Although the present client interface does not enable the user to command trajectories that are not piecewise linear, the system's performance in this regard is of interest for potential future applications. It must be emphasized that tracking performance is heavily a function of the Elmo control hardware and the means through which the Robot Component interacts with them. The present tests therefore provide only a lower bound as to the system's potential capabilities. Possible techniques for improving performance are discussed in subsection 5.2.1.

4.5.1 Test Set-up

Trajectory tracking performance was assessed using the Vicon motion tracking system described in subsection 2.2.6. In order to mitigate undesired triggering of the Vicon system by surfaces other than the reflective markers, Ranger's reflective surfaces were covered with paper and painter's tape, as shown in Fig. 4.4. Reflective spherical markers were affixed with hot glue. The optical bench was covered with a

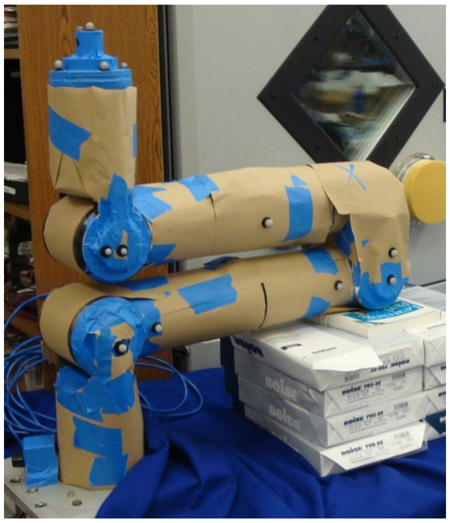


Figure 4.4 Ranger with reflective surfaces covered and markers attached in preparation for Vicon testing.

sheet of felt to avoid reflections off the tabletop. Most of the Faro Arm's outer casing is not especially reflective; however, some regions of exposed metal had to be similarly covered.

4.5.2 Test procedure

The optical bench with the Ranger and Faro arms was situated within the Vicon sensing volume. With the Faro arm attached, Ranger was successively commanded to visit six waypoints shown in Fig. 4.5 in alphabetical order, maintaining the end effector at a constant and level attitude. The path forms a rectangle parallel to the x-z plane of Ranger's base frame, at y = 58 cm. Point A is located at (0, 58, 40) cm. At each of these test points, the end effector position was recorded using both the Faro arm and the Vicon system. This provides a basis for assessing the accuracy of the Vicon measurements and identifies the waypoints within the Vicon coordinate system, enabling assessment of the accuracy of the shape of the path as distinct from any absolute positioning issues. With the Faro arm detached, Ranger was then commanded to perform a multipoint trajectory through the four vertices of this test path, nominally proceeding from B to C in 2 seconds, from C to E in 5 seconds, from E to F in 2 seconds, and from F to B in 5 seconds.[20] This trajectory was specified and initiated

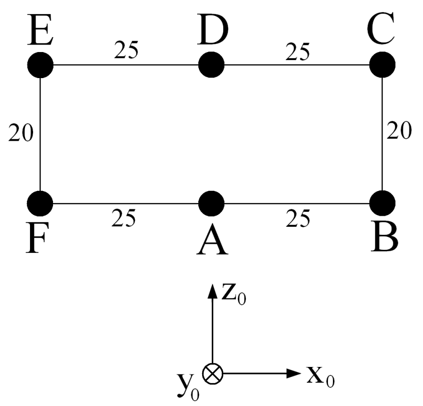


Figure 4.5 Rectilinear tracking test path. Dimensions in centimeters.

[20] The commanded motion is calculated to form a trapezoidal trajectory having the desired travel time within acceleration limits. The top speed is therefore slightly in excess of 10 cm/s.

entirely via the client interface, and was executed twelve times with the Vicon system recording the motion.

Circular and sinusoidal trajectories were then similarly executed and recorded. Because the present control application does not yet provide a means for the client to command such trajectories, the trajectory generation component was replaced successively with a Cartesian circle generator (SSL::CircleGen) and a Cartesian sinusoid generator (SSL::SinusoidGen). The source code for these components is located in shapegens.cpp and presented alongside the rest of the software in Appendices B and C. Although these trajectories were therefore preprogrammed, that preprogramming occurred only for the Cartesian path. The inverse kinematics were still computed during run time for each waypoint as it became active. From a performance standpoint, this is therefore no different than if these trajectory generation components were receiving the waypoints from an outside source in real time. The paths were executed at various speeds to assess the effect on tracking performance.

4.5.3 Analysis

The data initially recorded by the Vicon system consist of two-dimensional measurements from each camera, which must then be converted to three-dimensional position data in post-processing using Vicon-specific software. This processing was performed by Autonomous Vehicle Laboratory personnel and provided to the author for analysis. The rectilinear tracking test and some of the circular tracking tests were recorded by the Vicon system at a rate of 350 Hz. This resulted in excessively large log files that, it was found during subsequent processing, cause the Vicon software to crash. For this reason, these data are not available for analysis. This precludes the possibility of quantifying tracking error for the straight-line path employed for point-to-point motions commanded via the client interface. It also precludes comparison of Faro Arm measurements to Vicon measurements for purposes of estimating the accuracy of the Vicon system.

Because the purpose of this test is to assess tracking of a desired path and not absolute positional accuracy, which was the subject of Section 4.3, the present analysis concerns itself only with the shape and dimensions of the test paths and not with their absolute positions and orientations within space. The centers of the circular and sinusoidal test paths were identified by taking the arithmetic mean of the coordinates for the slowest run of that particular type of path. The axis or plane of the motion was then identified via a singular value decomposition of a matrix containing the recentered position data. The commanded motion was logged by the control application using a separate clock from the one used by the Vicon computer, with no means implemented of synchronizing the two. A least squares match was therefore employed to align the commanded and recorded motions in precise phase with one another for comparison. This does not allow for quantification of the time delay between generation and execution of the commands; however, identifying that quantity is not the purpose of the present test.

Tracking error at each point was calculated as the Cartesian straight-line distance between the recorded and desired positions via the same formula presented in subsection 4.3.3 for static positional accuracy. The arithmetic mean of this value over eight cycles of the motion was computed as the figure of merit for tracking accuracy. This includes a small dynamic response as the motion begins from rest, and the Whistle controllers attempt to catch up with the commanded velocity. All test paths began and ended at the same point.

4.5.4 Results

Table 4.3 shows the mean tracking error for the trials conducted with a circular test path 20 cm in radius, situated parallel to the x-z plane of Ranger's base frame. At low to moderate speeds, tracking error is approximately four millimeters. For the high-speed trajectory having a period of 5 seconds, the tracking error greatly increases to more than 4 cm. Fig. 4.6 shows the actual and commanded paths for slowest (T = 20 sec) and fastest (T = 5 sec) runs of the circular test path. As can be seen in this figure, the executed path for T = 20 seconds is quite nearly circular, but a bit smaller in radius than was commanded.[21] The commanded paths shown in the figure were recalculated with the logged joint angle commands to demonstrate that the degradation of performance at high speeds was not caused by the control application. Rather, it appears that the Whistle units were unable to effect motion at the commanded rates. This was almost certainly due to a configuration parameter within the Whistle units which enforced a speed limit of approximately 0.9 rad/sec, while the test trajectory resulted in commanded rates of approximately 1.5 rad/sec on both joints 4 and 6. It is likely that this issue could be surmounted by reconfiguration of the Whistles; however, servo level performance is not the focus of this thesis. Reference 16 explores the capabilities of this hardware platform in greater detail. For

[21] Indeed, if compared to a circle of radius 19.6 cm, this path would exhibit only 1.2 mm mean tracking error. The other circular test runs similarly conformed more nearly to a circle of that radius. It could not be determined from the available data whether this might be a scaling issue within the Vicon system.

the present purposes, it is simply observed that the control application is not the limiting constraint on the ability of the overall system to track these motions.

Table 4.3 Mean tracking error $\bar{d}$ for a circular test path of 20 cm radius parallel to the x-z plane. T is the period of commanded motion, f is the frequency of commanded motion, v is the nominal tangential speed, and S is the sample standard deviation on $\bar{d}$.

T (sec)	f (Hz)	v (mm/s)	$\bar{d}$ (mm)	S (mm)
5	0.200	251	42	20
7	0.143	180	4.7	2.3
10	0.100	126	4.0	0.9
15	0.0667	84	3.9	0.9
20	0.0500	63	3.8	0.9

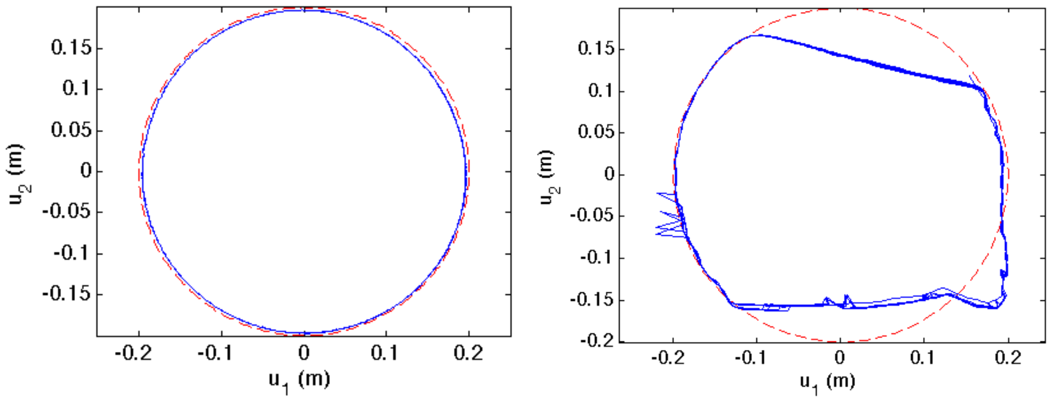


Figure 4.6 Circular test path with period T = 20 seconds (left) and T = 5 seconds (right), shown in the plane of motion. The actual path is shown as a solid blue curve, with the commanded path (recomputed from the commanded joint angles to show any error in the inverse kinematics solution) as a dashed red curve.

Sinusoidal test paths were performed in both the x and z directions of the Ranger base frame with a peak-to-peak amplitude of 20 cm, centered on the point (0, 62, 52) cm. Table 4.4 shows the results from the x-direction maneuvers. Table 4.5 shows the results for the z-direction maneuvers. Both directions were tested at frequencies between 0.5 and 0.1 Hz. Due to its greater manipulability in the z versus the x direction in this configuration, z-direction testing exhibited superior performance, and was additionally performed at 1 Hz. For low to moderate speeds, both test sets demonstrated mean tracking errors of a few millimeters. Fig. 4.7 shows the measured and commanded paths for two of the x-direction sinusoidal trials. It was again verified via recomputation of the forward kinematics on the commanded joint angles that the great majority of the error for the 0.5 Hz sinusoids was the result of tracking error on the Whistles and not the control application.

Table 4.4 Mean tracking error $\overline{d}$ for a sinusoidal motion in the x direction. T is the period of commanded motion, f is the frequency of commanded motion, v_{max} is the nominal maximum speed, and S is the sample standard deviation on $\overline{d}$.

T (sec)	f (Hz)	v_{max} (mm/s)	$\overline{d}$ (mm)	S (mm)
2	0.500	314	37	15
3	0.333	209	21	11
5	0.200	126	2.9	1.9
8	0.125	79	2.6	1.3
10	0.100	63	2.6	1.2
15	0.0667	42	2.6	1.2

Table 4.5 Mean tracking error $\overline{d}$ for a sinusoidal motion in the z direction. T is the period of commanded motion, f is the frequency of commanded motion, v_{max} is the nominal maximum speed, and S is the sample standard deviation on $\overline{d}$.

T (sec)	f (Hz)	v_{max} (mm/s)	$\overline{d}$ (mm)	S (mm)
1	1.00	628	23	11
2	0.500	314	3.4	1.1
3	0.333	209	3.8	0.8
5	0.200	126	3.7	0.9
8	0.125	79	3.7	1.1
10	0.100	63	3.7	0.9

It is likely that the Whistle units could demonstrate higher bandwidth for a smaller amplitude motion (See Reference 16). Smaller motions were not attempted in the present study due to limited testing time with the Vicon system and uncertainty as to the precision to which the Vicon system could reliably detect marker locations. In either event, this test provides a lower bound on the available dynamic performance of the combined hardware-software system presently employed to control Ranger.

Although the specific straight-line motion commands available via the client interface could not be analyzed due to data processing issues, such motions are not likely to result in greatly different performance from those analyzed here.

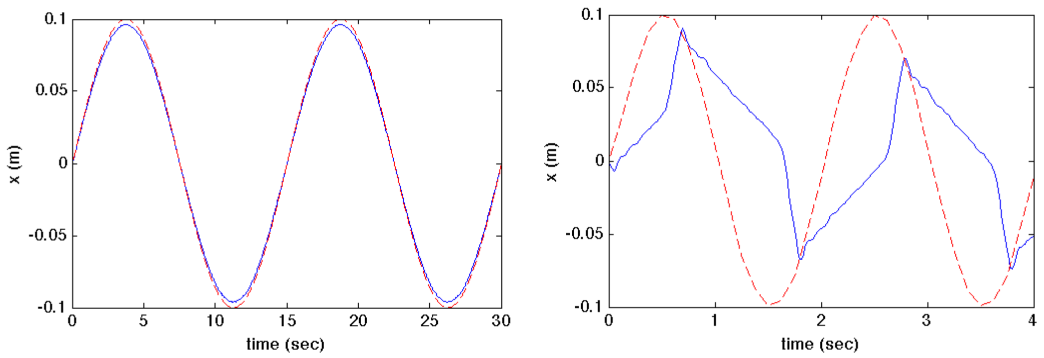


Figure 4.7 Sinusoidal test path in the x direction with period T = 15 seconds (left) and T = 2 seconds (right). The actual path is shown as a solid blue curve, with the commanded path (recomputed from the commanded joint angles to show any error in the inverse kinematics solution) as a dashed red curve. Although a least squares alignment was used to put the commanded and actual paths in phase for the purpose of computing the mean tracking error, the plot at right shows the two curves approximately aligned on the basis of the time at which motion began.

4.6 Simulation of more diverse kinematics

The preceding evaluations validate the basic functionality required from the control application, but do little to demonstrate the true adaptability of the system to a variety of manipulators. Different control hardware can be accommodated by writing a new Robot component.[22] The basic ability to substitute one robot component for another is a clear capability of the Orocos framework, and is demonstrated by the fact that the author developed much of the present software using SSL::SimArm and SSL::SimArm_nAxes. What is lacking to this point is therefore a demonstration of the ability to accommodate different manipulators having a variety of kinematic

[22] If the new control hardware is not capable of directly accepting position commands and running a servo loop, the necessary capabilities can be implemented as additional components in the control application, taking as input the desired joint angles and outputting suitable torques or joint rates to achieve them. It would not, however, require substantial modification to the existing software components.

configurations. The six-degree-of-freedom, all-revolute nature of Ranger is a common configuration, but is not by any means the only one. The proceeding simulations were therefore undertaken to demonstrate the versatility of the system to operate on robots having prismatic joints and redundant degrees of freedom.

4.6.1 Mixture of prismatic and revolute joints

Table 4.6 shows the Denavit-Hartenberg parameters for a hypothetical robot consisting of a three-degree-of-freedom, pitch-pitch-roll arm mounted to a mobile, three-degree-of-freedom planar platform. Fig. 4.8 shows a diagram of the robot.

Table 4.6 Denavit-Hartenberg parameters for a hypothetical robot containing prismatic joints.

i	α_{i-1} (rad)	a_{i-1} (m)	d_i (m)	θ_i (rad)
1	$\pi/2$	0	d_1	$\pi/2$
2	$\pi/2$	0	d_2	$\pi/2$
3	$\pi/2$	0	0	θ_3
4	$\pi/2$	0	0	θ_4
5	0	0.5	0	θ_5
6	$\pi/2$	0	0.5	θ_6
T	0	0	0	0

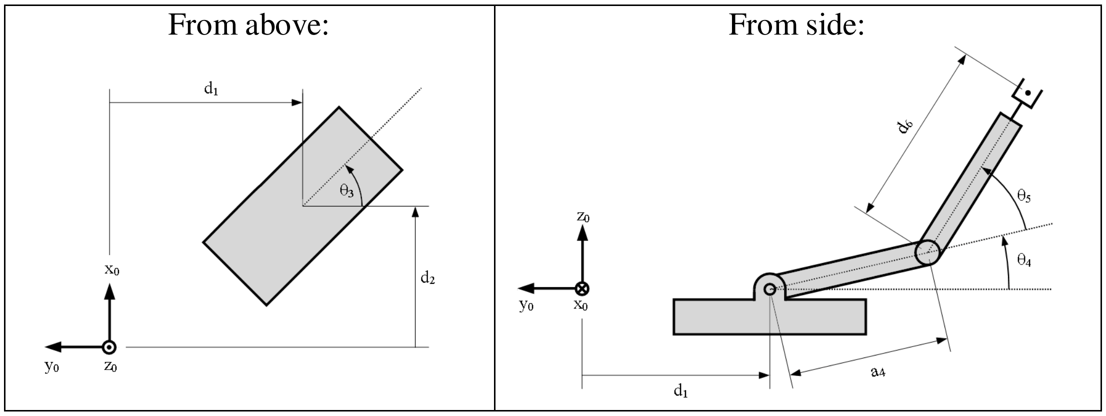


Figure 4.8 3DOF mobile platform with 3DOF arm. θ_6, not shown, is end effector roll.

The simulated robot was commanded through a rectangular trajectory consisting of waypoints (0.5,0.2,0.1), (0.5,0.2,0.4), (0.5,-0.2,0.4), and (0.5,-0.2,0.1) meters, returning to the starting point after reaching the fourth waypoint. The trajectory was executed at a nominal maximum velocity of 0.05 m/s and maximum acceleration of 0.10 m/s^2. Mean kinematic error was computed as the average distance between the end effector and the straight line connecting the previous to the following waypoint.

Fig. 4.9 shows the executed trajectory in joint space. The joint values vary smoothly and continuously over time, with a mean kinematic error of 0.18 mm. The joint space trajectory is fairly simple, with the two prismatic joints doing the x-y translational work, and joints 4 and 5 maintaining a linearly negative relationship to keep the end effector level while achieving the desired z coordinate. The maximum error, 2.02 mm, occurred approximately 23 seconds into the trajectory, during the downward leg of the motion.

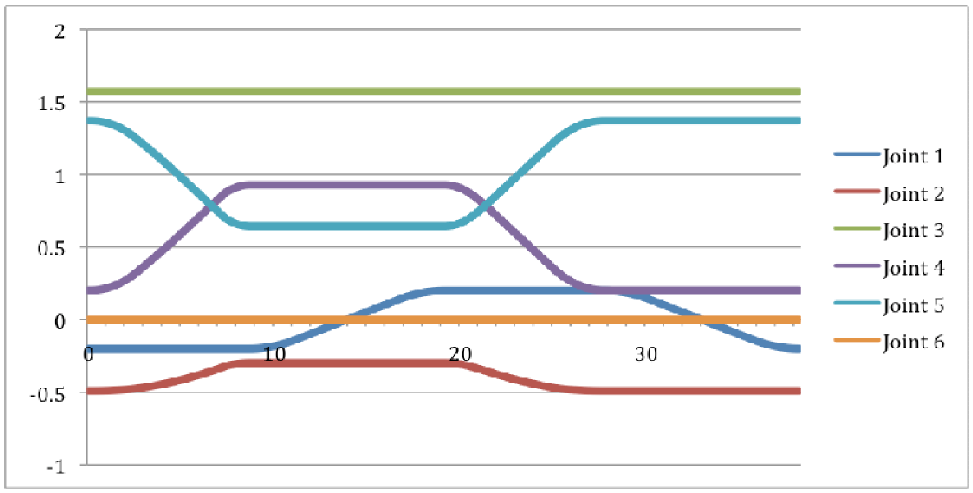


Figure 4.9 Simulated trajectory in joint space. Joints 1 and 2 are prismatic, with joint values given in meters. Joints 3 through 6 are revolute, with values given in radians.

4.6.2 Eight degrees of freedom

The software as written is capable of accommodating redundant manipulators, with the caveat that the client communication protocol established in the

MessageHandler component (see Appendix A) was not written to accommodate joint space commands in greater than six dimensions. If needed, it would be a reasonably simple matter to enlarge the standard message structure to accommodate the additional values. Because the present simulation is done in Cartesian space, however, no software modification is necessary. Table 4.7 shows the Denavit-Hartenberg parameters for a later-generation, eight-degree-of-freedom Ranger manipulator, also in use at the Space Systems Laboratory. Fig. 4.10 shows a diagram of the robot. This arm was commanded in simulation through the same sequence of waypoints as in the preceding simulation.

Table 4.7 Denavit-Hartenberg parameters for Ranger Mark II, from [24]. An offset to the tool frame, d_T = 0.1 m, was inserted by the present author to represent a generic tool.

i	α_{i-1} (rad)	a_{i-1} (m)	d_i (m)	θ_i (rad)
1	0	0	0.1524	θ_1
2	$\pi/2$	0	0	θ_2
3	$-\pi/2$	0	0.5389	θ_3
4	$\pi/2$	0	0	θ_4
5	$-\pi/2$	0.1524	0.5117	θ_5
6	$\pi/4$	0	0	θ_6
7	$\pi/2$	0	0	θ_7
8	$-\pi/2$	0	0	θ_8
T	0	0	0.1000	0

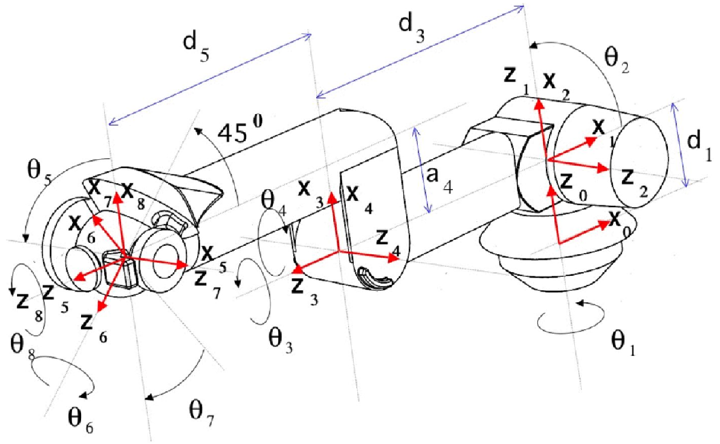


Figure 4.10 Ranger Mark II, from [24].

Fig. 4.11 shows the executed path in joint space. All eight degrees of freedom were employed in this motion, and all joint motions were smooth and continuous. Note that, although the robot returns to its original Cartesian pose at the end of the trajectory, its joint space configuration is different. This is typical of a Jacobian-pseudoinverse-based inverse kinematics approach, which minimizes the joint rates required for a given motion and thus will produce different joint space solutions when the goal is approached from different directions.

The mean kinematic error was on the order of one micrometer, the precision to which the forward kinematic results were logged. Maximum kinematic error was 8 micrometers and occurred at approximately 23 seconds into the maneuver, at around the same point in the trajectory for which the prismatic case produced its maximum path error. The improved manipulability afforded by the redundant degrees of freedom may be responsible for the decreased error in this test versus the simulation of the previous section. Obviously, this extremely high precision motion would be difficult to achieve on an actual robot due to real-world uncertainties such as joint flexibility and angular uncertainty, even if the robot's nominal dimensions were known to high precision. This test demonstrates, however, that the present system is capable of managing redundant degrees of freedom without significant software-level issues.

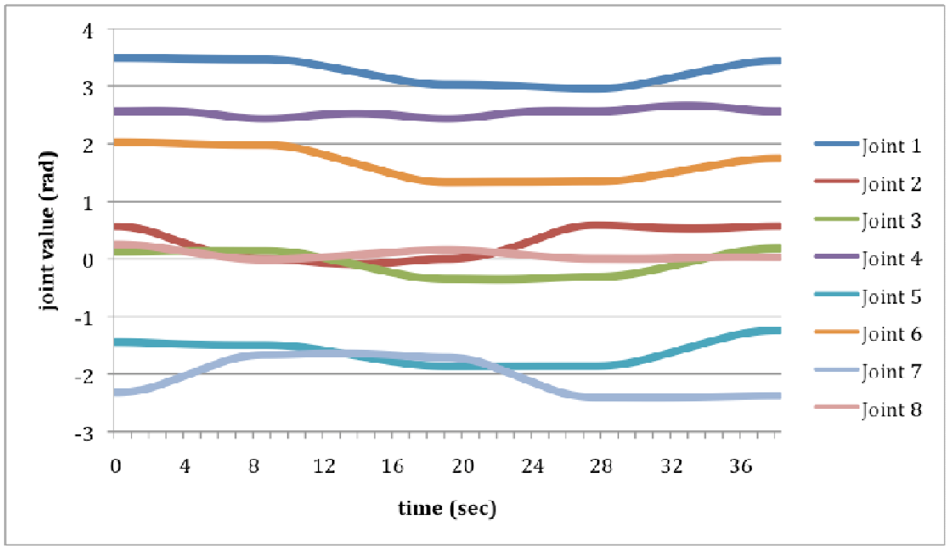


Figure 4.11 Simulated trajectory in joint space

Chapter 5: Conclusions and Future Work

5.1 Summary and Conclusions

The present research developed and implemented a software architecture for the top level control of a robotic manipulator. This architecture enables reuse of code between different robotics projects, as well as easy evolution and extension of the code by employing both component-based and object-oriented reusability strategies, and by leveraging the results of open-source projects freely available online. As a demonstration of its basic validity and capability, the system has been demonstrated on the Ranger Mark I manipulator. It demonstrated adequate performance in terms of both static positional accuracy and trajectory tracking. Further simulation has shown that the system is capable of accommodating both prismatic and revolute joints, and both redundant and nonredundant serial-link manipulators.

5.2 Future Work

5.2.1 Refinement of Performance

Further work could be done to refine the system to improve the performance of the present implementation of this architecture. Although Orocos supports real-time extensions to the Linux kernel, the present development and testing occurred on a non-real-time Ubuntu Linux system. By upgrading to a real-time system, it would become possible to assign priorities to each thread, ensuring that the control-critical components can be executed in a timely manner when they need to run. The PEAK-

System driver may be compiled with support for the Xenomai real-time environment. Since the current arrangement has the Whistles buffering incoming waypoints and timing the spacing of waypoints onboard, some amount of fluctuation in the arrival of incoming commands is tolerable and will not affect system performance. In some of the potential future applications described below, however, this would adversely affect system performance.

In the interactions with the Whistles as well, there is further room for improvement. Interaction between the host computer and Whistles is currently the limiting constraint on execution rate of the control components. Although the investigations in Section 4.5 showed the Whistle tracking performance to be the ultimate limiter of performance for the motions attempted, the update rate of the control application could become the limiting constraint if smaller amplitude motions were attempted or if the Whistles were reconfigured for improved performance. The limitations on update rate are likely a consequence of use of the non-real-time version of the PEAK-System driver[23] and of data rate limitations. In the present tests, the CAN bus was operated at 500 kilobits per second.[24] Though the present hardware was not tested at higher speeds, the CAN bus standard supports a maximum rate of 1,000 kilobits per second. Communication efficiency might also be improved by mapping the communication objects such that messages transmitted by the host

[23] The non-real-time driver may be delaying transmission of outgoing messages and processing of incoming ones. The present author observed that Graphical User Interface (GUI) events on the desktop operating system appeared to correlate with warnings from the Robot Component that responses were not received in a timely manner. A real-time implementation would allow for prioritizing the control application more highly than other, less important processes which may be interfering with it. Disabling the host computer's GUI entirely, whether switching to a real-time implementation or not, would likely reduce such events.

[24] At the update rate of 51 Hz at which the tests of Chapter 4 were performed, this would result in a theoretical bus utilization of approximately 60% of capacity.

computer to the Whistles have a higher priority than replies from the Whistles to the host computer. This would help ensure that the Whistles receive their instructions as quickly as possible at the beginning of each cycle. As currently configured, it is possible that this dissemination of instructions to the later Whistles on the list is delayed by reply messages from Whistles at the beginning of the list that have already finished executing that instruction. In addition, for position-based trajectory specification, a high speed communication technique is available in which two trajectory set-points are specified within a single CAN message, which is mapped to write directly to the trajectory buffer without the need to be processed by an interpreter as is the case with other commands.[25] This would reduce both bus traffic and Whistle processing time, likely allowing for the set-points to be spaced more closely in time, thus reducing the motion delay due to buffering. Improved synchronization of the motion of the various drives could be improved by configuring the Whistles to respond to a particular group ID number, allowing a single message on the CAN bus to instruct all of them to begin motion. Sending the begin-motion commands separately to each Whistle as is currently done theoretically produces a mismatch of at least 480 microseconds between the first and last joint at a data rate of 500 kilobits per second.[26]

In addition to the SimplIQ Command Language, the SimplIQ line of servo drives also supports the DSP 402 protocol for motion controllers defined by the CAN

[25] This technique is not available for other control modes in which only desired joint rates or motor currents are communicated.

[26] This technique would also work in the joint-rate-only control mode because desired joint rates are similarly issued via separate specify-motion/begin-motion commands. Precisely synchronizing motor current commands, which take effect immediately when processed, would require use of the synchronous trigger communication object.

in Automation (CiA) nonprofit organization. This alternate protocol merits further investigation to determine whether it could be used to improve system performance, as well as for the potential development of a robot component which is compatible with an even greater variety of servo drives than the present SSL::ElmoArm2 component.

5.2.2 Teleoperation

A more natural method for a human user to operate the manipulator might be the issuance of Cartesian twist[27] (rather than pose) commands via joysticks (rather than a keyboard). Although this could be done by replacing the trajectory generator with a component to increment the position command according to the desired twist, sending joint rate commands directly to the servo drives would have the advantage of eliminating the time delay due to trajectory buffering. The basic capabilities required for this are already present. As mentioned in Chapter 3, the necessary inverse kinematic solver has already been written, and the Robot Component supports joint-rate commands. With these components in place, the only further requirement would be the replacement of the trajectory generation component with a component which generates a desired twist on the basis of user input which is received in some manner—perhaps through the same UDP interface provided for other user commands, though a more nearly real-time protocol may be desirable.

[27] Translational and rotational rates

5.2.3 Path Planning

The trajectory generator currently employed, taken straight from the Orocos Component Library, does nothing more than connect the current and desired positions via a straight-line path. More sophisticated path planning capabilities may be useful. Such a planner could attempt to find a path to a goal position while avoiding obstacles, singularities, and self-intersecting poses. If the approach is simple, requiring relatively little planning time, the functionality might be written into a single component to replace the OCL trajectory generator currently employed. A more sophisticated approach, in which path planning time is more substantial, might be better implemented with separate planning and execution components. The higher level planning component(s) could run in a separate thread, perhaps triggered by incoming commands, while the executive trajectory generator would run periodically along with the other control components as in the present application.

Path planning with obstacle avoidance is an active research field unto itself, and has been described as "among the most difficult problems in computer science" [25]. The best known solutions which are guaranteed to find a solution if one exists (and to fail in finite time if one does not) grow at least exponentially with the number of degrees of freedom. Therefore, except for in simple cases, one normally employs techniques which cannot be formally guaranteed to succeed. Craig [20] divides the basic approaches which have emerged into two different styles. In the first, a connected graph is used to represent motions which the robot may follow to travel from one pose to another. This graph is then analyzed using some search algorithm to find a path that connects the initial and goal poses. The second approach is to construct artificial potential fields that repulse the robot from disallowed poses and attract it to the goal pose. The robot is then made to move as though it were a point particle under the influence of these fields. Spong, Hutchinson, and Vidyasagar [25] describe an approach to this latter technique, which is usually implemented in Cartesian space because determining the joint space distance to an obstacle is often difficult. Such potential field techniques are desirable in that waypoints may be generated on the fly, with the next step in the motion depending only on the fictitious forces experienced in the current configuration. Lozano-Perez [26] describes an approach instead employing the former technique, expending some up-front computational effort to construct a library of legal joint space configurations to be searched. This technique is more difficult to implement in a generically reusable component, but is more robust for path planning in cluttered environments.

Marani, et al., [27] present a singularity avoidance technique which is particularly appealing for the present application. It introduces a correction term to the desired direction of motion on the basis of the measure of manipulability, preventing the arm from moving into regions in which its manipulability drops below a particular constant. This technique is particularly relevant to the present goal of writing reusable software, because it requires no advanced knowledge of the singular configurations of the robot. Implementing this technique generically, however, would require (in addition to other values which the Orocos KDL can readily produce) the calculation of the derivative of the manipulator Jacobian with respect to the robot's joint angles, ideally on the basis of only the Denavit-Hartenberg parameters so as not to require additional configuration information beyond that already used to describe the robot geometry. This technique is applicable to both redundant and nonredundant manipulators, though altering the desired path in this way is often unnecessary in the kinematically redundant case. Nicholas Scott [28] presents a singularity avoidance technique which utilizes the self-motion of a kinematically redundant arm to avoid singularities and joint limits without deviating from the desired trajectory, adding a correctional term to the Jacobian-pseudoinverse-based solution which resides in the Jacobian's null space. This technique could be implemented in the present application by inserting an additional component immediately following the presently employed inverse kinematics component.

5.2.4 Nonlinear Control

Although the present arrangement has the Whistles implementing a local feedback loop for each joint, the control application presented here forms only an open loop for the top-level control of the system. The joint-rate and especially the motor current control modes available from the Whistles and partially implemented in the present software could enable the use of a higher-level closed control loop employing a more advanced nonlinear controller at the robot level rather than separate linear controllers at each of the joints individually. Spong, Hutchinson, and Vidyasagar [29] describe inverse dynamics and passivity based control laws, presenting robust and adaptive versions of each. By employing knowledge of the robot's dynamics, including the varying effects of gravity and inertia[28], such controllers may be able to provide superior tracking performance throughout the

[28] The moment arm to the collective center of mass of the links beyond a given joint in the kinematic chain varies as a function of the joint angles. At full extension, more torque is generally required to achieve a given joint acceleration than when the arm is in a more compact configuration.

entire workspace. Moving the control loop to the host computer comes at a cost, however, in that the Whistles have a default position-loop sampling rate of 2.75 kHz, far greater than would be achievable via the host computer.

5.2.5 Future Applications

Because of the success of this project, the present system will continue to see use on the Mark I Ranger manipulator, soon to be mounted to the Robotic Assist Vehicle for Extraterrestrial Navigation (RAVEN) earth analogue. RAVEN is a three-wheeled lunar astronaut assistance rover designed by aerospace engineering seniors in the University of Maryland's 2009-2010 ENAE 484 capstone design course. The earth analogue, shown in Fig. 5.1, was designed to accommodate the Mark I Ranger manipulator. On this vehicle, Ranger will be used to demonstrate the utility of a mobile robotic platform, and to test potential concepts of operation for future planetary exploration applications.

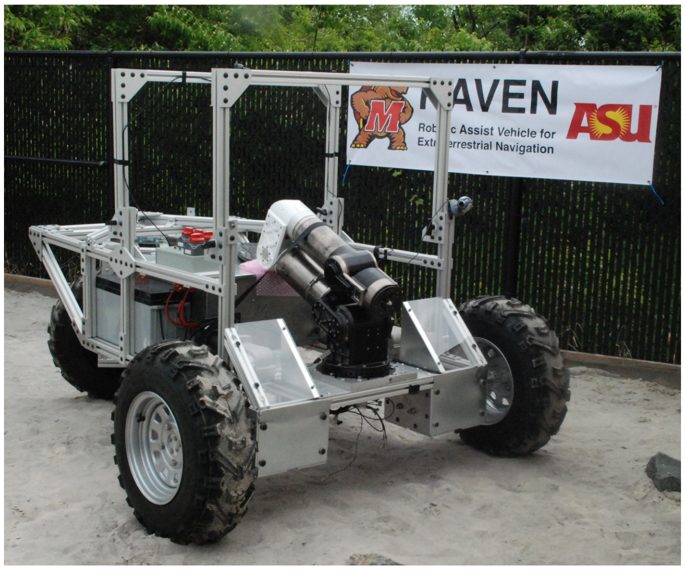


Figure 5.1 The RAVEN astronaut assistance rover. In this image, a nonfunctional Ranger manipulator occupies the location in the front of the vehicle where Ranger Mark I will be mounted. From [30].

The present architecture will also be implemented on the Subsea Arctic Manipulator for Underwater Retrieval and Autonomous Interventions (SAMURAI). SAMURAI is a six-degree-of-freedom manipulator designed to be used on an autonomous underwater vehicle for sampling operations. SAMURAI is in the process of being equipped with Whistle servo controllers to enable reuse of the present software without modification. This architecture will enable SAMURAI to be made operational more quickly than would otherwise be possible, and will serve as the baseline control system for that arm.

Appendix A: Client Command Interface

The following commands are available to the client for manipulator operations. Each entry begins with two characters used to identify the command. The following boxes indicate the data values to be provided with that command. The seven larger boxes represent 4-byte values[29], while the final smaller box represents one byte[30], for a total of 29 data bytes.

The control application replies to a command with a message of the same 31-byte size having the two leading command characters reversed (e.g., "GB" in response to a BG begin-motion command). In addition to this, the control application may also transmit errors and warnings having leading characters ER and WN, respectively. These messages consist entirely of ASCII-format characters forming a null-terminated string that specifies the nature of the error or warning

BG	0	0	0	0	0	0	0	0

Begin motion to the Cartesian goal pose: The trajectory generator will begin outputting waypoints moving the robot from its current pose to the goal pose (specified with GL) via a trapezoidal trajectory. This command will fail if no goal has been specified or if a previous motion is still in progress.

CM	M	0	0	0	0	0	0	0

Switch command mode: Instructs the CDH switching component to switch the control mode to the setting specified by the value M. Command mode 0 is off, and prevents any commands from reaching the Robot Component. Command mode 1 is joint space control. Command mode 2 is Cartesian control.

[29] Floating point numbers, arranged with the least significant byte first.
[30] Unsigned single-byte integer

GL

x	y	z	Q1	Q2	Q3	Q4	0

Specify Cartesian-space goal: Specifies the goal end effector pose for the next Cartesian motion as a position (x,y,z) in meters and a unit quaternion (Q1,Q2,Q3,Q4). The trajectory generator will move toward this goal as quickly as possible given its configured speed and acceleration limits.

GT

x	y	z	roll	pitch	yaw	T	0

Specify Cartesian-space goal with minimum duration of motion: Specifies the goal end effector pose for the next Cartesian motion as a position (x,y,z) in meters and an orientation described by angles roll, pitch, and yaw (in radians) according to the Z-Y-X Euler angle convention (Yaw about Z, Pitch about Y, Roll about X). The trajectory generator will calculate the duration of motion to be either the specified time T, in seconds, or the minimum duration allowable due to configured speed limits, whichever is larger.

jB

0	0	0	0	0	0	0	0

Begin motion to the joint space goal: The trajectory generator will begin outputting waypoints moving the robot from its current pose to the goal pose (specified with jG) via a trapezoidal trajectory. This command will fail if no goal has been specified or if a previous motion is still in progress.

jC

0	0	0	0	0	0	0	0

Clear all specified joint space via points: Clear all via points previously specified with the jV command.

jE

0	0	0	0	0	0	0	0

Execute the joint space multipoint trajectory: Execute the sequence of waypoints, which has been specified previously using the jV command. Motion will continue until it reaches an index for which a via point has not been specified or the system fails to reach a via point within tolerance.

jG

Th1	Th2	Th3	Th4	Th5	Th6	0	0

Specify goal for the next joint space motion: Sets the goal for the next joint space motion to the given joint angle values (Th1,…,Th6) in radians. When executed with the command jB, motion will proceed as quickly as possible under the configured joint rate limitations.

jP

0	0	0	0	0	0	0	0

Query joint space configuration: The control application will reply with the current joint angles.

jV

Th1	Th2	Th3	Th4	Th5	Th6	T	i

Specify a joint space via point for multipoint trajectory: Specifies the ith via point in a joint space multipoint trajectory with minimum duration of motion T in seconds. The first via point is specified with i=0. The jE command is used to begin motion.

OF

0	0	0	0	0	0	0	0

Disable the Robot Component: The Robot Component's stop hook is executed, bringing the component into its stopped state and preventing further motion of the arm. This is a more robust way of stopping the arm than the ST command.

ON

0	0	0	0	0	0	0	0

Enable the Robot Component: The Robot Component's start hook is executed, bringing the component into its running state and enabling motion of the arm. As presently configured, the Robot Component is started automatically and this command is needed only if the component has been disabled via the OF command.

PS

0	0	0	0	0	0	0	0

Query Cartesian pose: The control application will reply with a message containing the current end effector pose in the same format as used for the GL command.

RC

Th1	Th2	Th3	Th4	Th5	Th6	0	0

Recalibrate joint angles: Declare the current joint angles to be (Th1,...,Th6), in radians. The Robot Component will adjust its home position (corresponding to zero encoder counts) accordingly. Because the reported joint angles will change discontinuously, this should only be performed in command mode 0. This change will not persist after the control application exits.

ST

0	0	0	0	0	0	0	0

All stop: The trajectory generator will be reset, with the current Cartesian pose taken as the new desired pose.

TM

0	0	0	0	0	0	0	0

Query drive temperatures: The control application will reply with the drive temperatures reported by the Robot Component, in degrees Celsius. A value of -100°C or lower indicates that temperature information has not been reported for that drive. The SimplIQ standard for the servo drives used by the present Robot Component specifies that a servo drive that does not support temperature sensing will report -55°C.

VC

0	0	0	0	0	0	0	0

Clear all specified Cartesian space via points: Clear all via points previously specified with the VP command.

VE

0	0	0	0	0	0	0	0

Execute the Cartesian space multipoint trajectory: Execute the sequence of waypoints, which has been specified previously using the VP command. Motion will continue until it reaches an index for which a via point has not been specified or the system fails to reach a via point within tolerance.

VP

x	y	z	roll	pitch	yaw	T	i

Specify Cartesian space via point: Specifies the i^{th} via point in a Cartesian space multipoint trajectory with minimum duration of motion T in seconds. The first via point is specified with i=0. The VE command is used to begin motion.

Appendix B: Header Files

h/elmoarm2.h

```
/*
 * An orocos component (ElmoArm2) for interfacing with the Elmo arm hardware
 * via CANbus.  There should be only a single instance of this component per arm,
 * regardless of the number of servos comprising it.  It should be run as a periodic
 * task at precisely 3x the speed of the kinematic/planning components.  Furthermore,
 * the period of the kinematic/planning components must be an integer multiple of the
 * Whistle controller sampling rate (360 microseconds by default).
 *
 * Communication with a particular joint may be suppressed by setting its node-ID
 * to zero.  The EncoderReadings Data Port will indicate that that joint is
 * taking on the desired value.  This can be used, for example, to disable end
 * effector roll on the actual robot without affecting the kinematic components.
 * Delay on any such suppressed joints will be greatly less than that of the
 * real ones (because no buffering/smoothing occurs).  Joints may be suppressed
 * only in position control mode.
 *
 * Fork of ElmoArm.  Trajectory smoothing via Whistle binary interpreter.
 *
 */


#ifndef SSL_ELMOARM2_H_
#define SSL_ELMOARM2_H_


#include <libpcan.h> // requires linking with pcan library during build
#include <fcntl.h> // for CAN initialization
#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
//include <rtt/marsh/PropertyMarshaller.hpp>
#include <rtt/Logger.hpp>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
//using namespace RTT;


// Unit Mode (UM) control codes as defined on page 3-148 of the SimplIQ
// Command Reference Manual.  Any new control modes must also be added
// to the validity check in startHook().
#define ELMO_ARM_POSITION_CONTROL 5
#define ELMO_ARM_SPEED_CONTROL 2
#define ELMO_ARM_TORQUE_CONTROL 1


namespace SSL {


// A structure for keeping track of the latest knowledge as to the
// state of the servo drive.
struct ElmoArmStatus
{
     char nodeID;

     int BG;
     bool wait_BG;
     DWORD time_BG;

     float IQ;
     bool wait_IQ;
     DWORD time_IQ;
     int JV;
     bool wait_JV;
     DWORD time_JV;

     int PA;
     bool wait_PA;
     DWORD time_PA;

     int PX;
     bool wait_PX;
     DWORD time_PX;
     char tally_PX;

     float TC;
     bool wait_TC;
     DWORD time_TC;
     char tally_TC;

     int TI;
     bool wait_TI;
     DWORD time_TI;

     int VX;
     bool wait_VX;
     DWORD time_VX;
     char tally_VX;

     int PT;
     int prevPT;
     bool wait_PT;
     DWORD time_PT;
     char tally_PT;

     int UM;

     bool PTmap;

     bool buffer[6];

     bool overrun;
     bool underrun;

     // When adding elements to this, remember to put them in process_messages as
well.
     // Also, initialize the values in configureHook.

};



class ElmoArm2 : public RTT::TaskContext
{
     protected:
       RTT::Property<int> numServos;
       RTT::Property< std::vector<double> > home;
       RTT::Property<int> ControlMode;
       RTT::Property< std::vector<double> > EncoderCountsPerRev;
       RTT::Property< std::vector<double> > NodeIDarr;
       RTT::Property< std::string > CANdevice;
       RTT::DataPort< std::vector<double> > DriveValue; // Radians
       RTT::DataPort< std::vector<double> > SensorValue;
       RTT::DataPort< std::vector<double> > Temperature; // degrees Celsius
       RTT::Method< bool(std::vector<double>) > recalibrateJoints;
       RTT::Method< void(void) > printMethod;

     public:
       ElmoArm2(std::string name);
       bool configureHook();
       bool startHook();
       virtual void updateHook();
       void stopHook();
```

```
        void cleanupHook();

     private:
        double* drivevalue; // array of drivevalues converted to counts
        double* prevdrivevalue; // for holding onto the last drive value
        HANDLE CANhandle; // handle for the CAN card/port
        //char* NodeIDarr; // array of node ID's for the servos in each joint
        TPCANRdMsg msg_in; // Structure for incoming CAN messages
        TPCANMsg msg_out; // Structure for outgoing CAN messages
        int intCANset(HANDLE& h, char nodeID, char C1, char C2, int index, int value);
        int ptCANset(HANDLE& h, char nodeID, int value1, int value2);
        int floatCANset(HANDLE& h, char nodeID, char C1, char C2, int index, float
value);
        int CANquery(HANDLE& h, char nodeID, char C1, char C2, int index);
        bool process_messages();
        ElmoArmStatus* statuses; // array of status structs for all the servos
        int unpackData();
        float unpackFloat(); // when the data's not an int
        std::vector<double> output;
        bool command;
        char* PTindex;
        int cycle;
        bool once;
        int temp_counter;
        bool recalibrateFunc( std::vector<double> vec );
        void printMethodFunc();
};

} // namespace SSL

#endif // SSL_ELMOARM2_H_
```

h/ethernet.h

```
/*
 * An orocos component (EthernetInterface) to accept commands and return
 * feedback via an ethernet interface.
 *
 */

#ifndef SSL_ETHERNET_H_
#define SSL_ETHERNET_H_


#include<rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/Logger.hpp>
#include <rtt/TimeService.hpp>
#include <server.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
//using namespace RTT;


namespace SSL
{
     struct message{
        char array[31];
        int size;
        RTT::TimeService::Seconds timestamp;
     };

     class EthernetInterface : public RTT::TaskContext
     {
        protected:
                RTT::Property<int> port; // port number
                RTT::BufferPort< struct message > IncomingMessage;
                RTT::BufferPort< struct message > OutgoingMessage;

        public:
                EthernetInterface(std::string name);
                bool configureHook();
                bool startHook();
                virtual void updateHook();
                void stopHook();
                void cleanupHook();

        private:
                communications::Server myserver;
                struct message receivedmessage;
                struct message sendbuffer;
     };

} // namespace SSL


#endif // SSL_ETHERNET_H_
```

h/jointlimiter.h

```
/*
 * An orocos component (JointLimiter) for ensuring joints do not move
 * beyond their respective limitations.  In speed control mode (UM=2),
 * it is possible for the joints to slightly exceed the specified
 * range of motion limit because the joint limiter will not engage
 * instantly, nor will the joint come to a stop instantly.  Similarly,
 * in position control mode (UM=5), the maximum joint rates are NOT
 * enforced.
 *
 */

#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/marsh/PropertyMarshaller.hpp>
#include <kdl/jntarray.hpp>
#include <cmath>
#include <assert.h>

#ifndef SSL_JOINTLIMITER_H_
#define SSL_JOINTLIMITER_H_

namespace SSL
{

     class JointLimiter : public RTT::TaskContext
     {
       protected:
               RTT::Property<int> numServos;
               RTT::Property<int> ControlMode;
               RTT::Property< std::vector<double> > UpLim;
               RTT::Property< std::vector<double> > LowLim;
               RTT::Property< std::vector<double> > MaxRates;
               RTT::Event<void(void)> limitevent;
               RTT::DataPort< std::vector<double> > DriveValueRequested; // Radians
               RTT::DataPort< std::vector<double> > DriveValue; // Radians
               RTT::DataPort< std::vector<double> > EncoderReading; // Radians
               RTT::Method<void(KDL::JntArray*,KDL::JntArray*)> getLims;

       public:
               JointLimiter(std::string name);
               bool configureHook();
               bool startHook();
               virtual void updateHook();
               void stopHook();
               void cleanupHook();

       private:
               std::vector<double> vec;
               double dbl;
               void getLimsFunc(KDL::JntArray*,KDL::JntArray*);
               std::vector<double> emptyvec;
     };

} // namespace SSL

#endif // SSL_JOINTLIMITER_H_
```

h/kinematics.h

```
/*
 * Orocos components for computing forward and inverse kinematics to allow
 * for control of a serial-chain arm in cartesian space.  ForwardKinematics
 * should run before the control (in the same thread) to update the position
 * feedback, and InverseKinematics should run after the controller to update
 * the desired commands.
 */

#ifndef SSL_KINEMATICS_COMPONENTS_H_
#define SSL_KINEMATICS_COMPONENTS_H_

#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <kdl/chain.hpp>
#include <rtt/Command.hpp>
#include <kdl/chainfksolver.hpp>
#include <kdl/chainfksolverpos_recursive.hpp>
#include <kdl/chainiksolvervel_pinv.hpp>
#include <kdl/chainiksolverpos_nr_jl.hpp>
//using namespace RTT;

namespace SSL {

class ForwardKinematics : public RTT::TaskContext
{
     protected:
        RTT::Property<int> numJoints;
        RTT::Property< std::vector<double> > DHparams;
        RTT::Property< std::vector<double> > EndEffDH;
        RTT::DataPort< std::vector<double> > JointPoses; // Input
        RTT::DataPort< KDL::Frame > CartesianPose; // Output
        RTT::Method< void(void) > printMethod;

        void printMethodFunc();

     public:
        ForwardKinematics(std::string name);
        bool configureHook();
        bool startHook();
        virtual void updateHook();
        void stopHook();
        void cleanupHook();

     private:
        KDL::Chain* armChain;
        KDL::ChainFkSolverPos_recursive* fksolver;
        KDL::Frame cartframe;
        KDL::JntArray jntarr;
};



class InverseVelKinematics : public RTT::TaskContext
{
     protected:
        RTT::Property<int> numJoints;
        RTT::Property< std::vector<double> > DHparams;
        RTT::Property< std::vector<double> > EndEffDH;
        RTT::Method< void(void) > printMethod;
        void printMethodFunc();

        //Input
        RTT::DataPort< std::vector<double>  > JointPoses;
        RTT::DataPort< KDL::Twist > DesiredTwist;

        //Output
        RTT::DataPort< std::vector<double> > JointVelocities;

     public:
        InverseVelKinematics(std::string name);
        bool configureHook();
        bool startHook();
        virtual void updateHook();
        void stopHook();
        void cleanupHook();

     private:
        KDL::Chain* armChain;
        KDL::ChainIkSolverVel_pinv* iksolver;
        KDL::JntArray qdot;
        KDL::JntArray jntarr;
        std::vector<double> v;
};


class InversePosKinematics : public RTT::TaskContext
{
     protected:
        RTT::Property<int> numJoints;
        RTT::Property< std::vector<double> > DHparams;
        RTT::Property< std::vector<double> > EndEffDH;
        RTT::Method< void(void) > printMethod;
        void printMethodFunc();
        RTT::Event<void(void)> divergenceEvent;

        //Input
        RTT::DataPort< std::vector<double>  > CurrentJointPose;
        RTT::DataPort< KDL::Frame > CurrentFrame;
        RTT::DataPort< KDL::Frame > DesiredFrame;

        //Output
        RTT::DataPort< std::vector<double> > NewJointPose;

     public:
        InversePosKinematics(std::string name);
        bool configureHook();
        bool startHook();
        virtual void updateHook();
        void stopHook();
        void cleanupHook();

     private:
        KDL::Chain* armChain;
        KDL::ChainIkSolverPos_NR_JL* iksolver;
        KDL::ChainFkSolverPos_recursive* fksolver;
        KDL::ChainIkSolverVel_pinv* ikvelsolver;
        KDL::JntArray q;
        KDL::JntArray jntarr;
        KDL::Frame solnframe, localdesframe;
        KDL::Twist error;
        RTT::TaskContext* jl;
        KDL::JntArray qmin;
        KDL::JntArray qmax;
        RTT::Method<void(KDL::JntArray*,KDL::JntArray*)> jlmeth;
        std::vector<double> v;
        bool go, once;
};


} // namespace SSL

#endif // SSL_KINEMATICS_COMPONENTS_H_
```

h/logger.h

```
/*
 * A very simple component for logging information to log files
 * because the OCL ReportingComponent is too complicated and
 * depends upon sampling data ports, thus potentially missing
 * messages.  This component gets its messages to be logged via
 * a buffered data port so it won't miss anything unless the
 * buffer overflows, but it is more intrusive in that it
 * requires other components to prepare messages for logging
 * rather than just passively reading what goes out on their
 * data ports.
 *
 * This component performs file I/O and thus is decidedly
 * nonrealtime.  It should not be put in the same thread as any
 * critical components.
 */

#ifndef SSL_LOGGER_H_
#define SSL_LOGGER_H_

#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <string>
//using namespace RTT;

namespace SSL
{
     struct LogElement
     {
        char c_str[100];
        int index;
     };

     class FileLogger: public RTT::TaskContext
     {
        protected:
               RTT::Property< int > numLogs;
               RTT::BufferPort< struct LogElement > Incoming;
        public:
               FileLogger(std::string name);
               bool configureHook();
               bool startHook();
               virtual void updateHook();
               void stopHook();
               void cleanupHook();
        private:
               FILE** pArr; // an array of pointers
               struct LogElement local_element;
     };

} // namespace SSL

#endif // SSL_LOGGER_H_
```

h/messagehandler.h

```
/*
 * An orocos component to handle received messages, providing
 * a simple interface originally written for interaction with
 * the Spring 2010 ENAE 484 rover.  Incoming messages are
 * expected to consist of 31 bytes.  The first two bytes form
 * a two-character (ASCII) command, and the remainder contain
 * any necessary data.  Upon successful receipt of an
 * instruction, this component will send a reply with the two
 * command characters reversed.
 *
 * This component will need to be peers with the path
 * generator as well as the arm.  It must be connected to the
 * forward kinematics.
 *
 * Commands:
 *
 * -BG: Begin motion
 *      Data: NULL
 * -CM: Switch command mode ( 0.0=off, 1.0=Joint, 2.0=Cartesian)
 *      Data: (float) mode, NULL
 * -GL: Set goal state to the specified Cartesian coordinates
 *          and orientation quaternion.  The quaternion follows
 *          the same convention as Craig (1989) equation 2.89.
 *      Data: (float) x, y, z, quat1, quat2, quat3, quat4
 * -GT: Set goal state to the specified cartesian coordinates
 *          and orientation roll/pitch/yaw, and specify
 *          minimum duration of motion.
 *      Data: (float) x, y, z, R, P, Y, t
 * -ON: Start the arm
 *      Data: NULL
 * -OF: Stop the arm by shutting down the arm component
 *      Data: NULL
 * -PS: Request end effector pose (Or should this be transmitted
 *          periodically?)
 *      Data: NULL  (reply contains floats x,y,z,quat1,...,quat4)
 * -jV: Specify a joint-space waypoint
 *      Data: (float) theta1,...,theta6,time, (byte) index
 * -jC: Clear all specified joint-space waypoints
 *      Data: NULL
 * -jE: Execute the specified joint-space waypoint sequence
 *      Data: NULL
 * -jP: Request joint-space pose (joint angles)
 *      Data: NULL  (reply contains floats)
 * -jG: Set joint-space goal (initiate motion with jB, not BG)
 *      Data: (float) q1,q2,q3,q4,q5,q6,0
 * -ST: Stop the arm by resetting the path generator
 *      Data: NULL
 * -TM: Request servo drive temperatures (deg Celsius)
 *      Data: NULL  (reply contains floats)
 */


#ifndef SSL_MESSAGEHANDLER_H_
#define SSL_MESSAGEHANDLER_H_


#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/Command.hpp>
#include <rtt/Logger.hpp>
#include <rtt/TimeService.hpp>
#include <kdl/frames.hpp>
#include <cmath>
#include <server.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <logger.h>
//using namespace RTT;



namespace SSL
{
     struct message{
       char array[31];
       int size;
       RTT::TimeService::Seconds timestamp;
     };

     class MessageHandler : public RTT::TaskContext
     {
       protected:
               RTT::BufferPort< struct SSL::message > IncomingMessage;
               RTT::BufferPort< struct SSL::message > OutgoingMessage;
               RTT::BufferPort< struct LogElement > Logger;
               RTT::DataPort< KDL::Frame > CartesianPose;
               RTT::DataPort< KDL::Frame > SetPointPose;
               RTT::DataPort< std::vector<double> > JointPose;
               RTT::DataPort< std::vector<double> > JointTemperature;
               RTT::DataPort< std::vector<double> > DriveValue;

       public:
               MessageHandler(std::string name);
               bool configureHook();
               bool startHook();
               virtual void updateHook();
               void stopHook();
               void cleanupHook();

       private:
               struct message receivedmessage;
               struct message outgoingmessage;
               double* dblarr; // incoming numbers unpacked here
               float* floatarr; // outgoing numbers packed here
               void* pointer;
               bool unpackFloats();
               bool sendFloats(char c1, char c2, unsigned char index);
               bool sendNull(char c1, char c2, unsigned char index);
               bool understood;
               KDL::Rotation goal_rot;
               KDL::Vector goal_vec;
               double goal_time;
               std::vector<double> goal_jnt;
               double angle; KDL::Vector axis; double sinhalfang; // intermediate
state for attitude representation
               KDL::Frame currentFrame;
               RTT::Command<bool(KDL::Frame,double)> cmd_moveTo;
               RTT::Command<bool(std::vector<double>,double)> cmd_jntMoveTo;

               RTT::Method<void(void)> methPG_reset;
               RTT::Method<void(void)> methJG_reset;

               RTT::Method<bool(void)> meth_roboStart;
               RTT::Method<bool(void)> meth_roboStop;

               RTT::Command<bool(void)> cmd_execSeq;
               RTT::Method<void(void)> meth_stopSeq;
               RTT::Method<bool(KDL::Frame,double,char)> meth_setVP;
               RTT::Method<void(void)> meth_clearVP;
               RTT::Attribute<int>* handle_switchCM;

               RTT::Command<bool(void)> cmd_jntExecSeq;
               RTT::Method<void(void)> meth_jntStopSeq;
               RTT::Method< bool(std::vector<double>,double,char) > meth_jntSetVP;
               RTT::Method<void(void)> meth_jntClearVP;
               RTT::Method<bool(std::vector<double>)> meth_recalJoints;

               struct LogElement localLogElement;
               RTT::Handle handleWsCartLimit;
               bool WsCartLimitCallback();
               int WsCartLimit;
               RTT::Handle handleJointLimit;
               bool JointLimitCallback();
               int JointLimit;
               bool InvKinDivCallback();
               int InvKinDiv;
               bool havegoal;
               bool jntC_avail;
               bool multipoint_avail;
               int logPose;
               int logSP;
               int logJnt;
               int logDrive;
               int rc;
               void logOutgoing(); // for logging errors/warnings
               std::vector< double >  localJnt;
               std::vector< double >  localTemperature;
               std::vector< double >  vec;
               bool bootingup;
               RTT::TaskContext* ptr;
     };

} // namespace SSL


#endif // SSL_MESSAGEHANDLER_H_
```

h/multipoint.h

```
/*
 * An orocos component (MultipointManager) for managing a set of
 * multiple via points to be sent sequentially to the path planner.
 */

#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/Command.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/marsh/PropertyMarshaller.hpp>
#include <kdl/frames.hpp>
#include <kdl/jntarray.hpp>
#include <cmath>
#include <assert.h>


#ifndef SSL_MULTIPOINT_H_
#define SSL_MULTIPOINT_H_

namespace SSL
{

     // Cartesian space multipoint manager
     class MultipointManager : public RTT::TaskContext
     {
       protected:
               //RTT::Property<int> maxSize;
               //RTT::Attribute< std::vector<KDL::Frame> > viaPoints;
               //RTT::Attribute< std::vector<double> > viaTimes;
               //RTT::Attribute< int > lastIndex;
               RTT::Command< bool(void) > execSequence;
               RTT::Method<void(void)> haltSequence;
               RTT::Method<bool(KDL::Frame,double,char)> viaPointSet;
               RTT::Method<void(void)> viaClear;
               RTT::DataPort< KDL::Frame > CurrentPose;

       public:
               MultipointManager(std::string name);
               bool configureHook();
               bool startHook();
               virtual void updateHook();
               void stopHook();
               void cleanupHook();

       private:
               KDL::Frame* localViaPoints;
               double* localViaTimes;
               int localLastIndex;
               bool* viaReady;
               int currentIndex;
               bool execSeqFunc();
               bool execSeqFinished();
               bool haltSeqFunc();
               RTT::Command<bool(KDL::Frame,double)> cmd_moveTo;
               RTT::Method<void(void)> methPG_reset;
               bool executing;
               KDL::Frame* viaPoints;
               double* viaTimes;
               bool busy;
               bool viaPointFunc(KDL::Frame frame, double time, char index);
               void viaClearFunc();
               bool findPG();
               KDL::Twist error;
               double normerr;
               int count;
     };
```

```
    // Joint space multipoint manager
    class MultipointManagerJnt : public RTT::TaskContext
    {
      protected:
              RTT::Command< bool(void) > execSequence;
              RTT::Method<void(void)> haltSequence;
              RTT::Method<bool(std::vector<double>,double,char)> viaPointSet;
              RTT::Method<void(void)> viaClear;
              RTT::DataPort< std::vector<double> > CurrentPose;

      public:
              MultipointManagerJnt(std::string name);
              bool configureHook();
              bool startHook();
              virtual void updateHook();
              void stopHook();
              void cleanupHook();

      private:
              std::vector< std::vector<double> > localViaPoints;
              double* localViaTimes;
              int localLastIndex;
              bool* viaReady;
              int currentIndex;
              bool execSeqFunc();
              bool execSeqFinished();
              bool haltSeqFunc();
              RTT::Command<bool(std::vector<double>,double)> cmd_moveTo;
              RTT::Method<void(void)> methPG_reset;
              bool executing;
              std::vector< std::vector<double> > viaPoints;
              double* viaTimes;
              bool busy;
              bool viaPointFunc(std::vector<double> point, double time, char index);
              void viaClearFunc();
              bool findPG();
              double error;
              double normerr;
              int count;
    };

} // namespace SSL

#endif // SSL_MULTIPOINT_H_
```

h/shapegens.h

```
/*
 * Orocos components for generating Cartesian paths of useful shapes
 * for assessing system performance.
 */

#ifndef SSL_SHAPE_GENERATORS_H_
#define SSL_SHAPE_GENERATORS_H_

#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <kdl/chain.hpp>
#include <rtt/Command.hpp>
#include <rtt/TimeService.hpp>
#include <cmath>
//using namespace RTT;

namespace SSL {

class SinusoidGen : public RTT::TaskContext
{
     protected:
        RTT::Attribute<double> K; // Gain (meters)
        RTT::Attribute<double> T; // Period (seconds)
        RTT::Attribute<int> axis;
        RTT::DataPort< KDL::Frame > CartesianPosDes; // Output
        RTT::DataPort< KDL::Frame > CartesianPosMeas; // Input
        RTT::Method< void(void) > printMethod;
        RTT::Method< void(void) > methReset;
        RTT::Command<bool(KDL::Frame,double)> cmdMoveTo; // Ignored
        RTT::Command<bool(void)> cmdWave;
        RTT::Command<bool(void)> cmdEndWave;

     public:
        SinusoidGen(std::string name);
        bool configureHook();
        bool startHook();
        virtual void updateHook();
        void stopHook();
        void cleanupHook();

     private:
        KDL::Frame cartframe;
        KDL::Frame zero;
        void printMethodFunc();
        void resetMethodFunc();
        bool waveFunc();
        bool waveDone();
        bool waving;
        bool stopAtZero;
        bool funcMoveTo( KDL::Frame, double );
        bool moveDone();
        bool endWaveFunc();
        bool waveEnded();
        RTT::TimeService::Seconds startTime;
        RTT::TimeService::Seconds now;
        double dx, dx_prev;
        KDL::Vector vec;
};



class CircleGen : public RTT::TaskContext
{
     protected:
        RTT::Attribute<double> R; // Radius (meters)
        RTT::Attribute<double> T; // Period (seconds)
        RTT::Attribute<int> axis_i;
        RTT::Attribute<int> axis_j;
        RTT::DataPort< KDL::Frame > CartesianPosDes; // Output
        RTT::DataPort< KDL::Frame > CartesianPosMeas; // Input
        RTT::Method< void(void) > printMethod;
        RTT::Method< void(void) > methReset;
        RTT::Command<bool(KDL::Frame,double)> cmdMoveTo; // Ignored
        RTT::Command<bool(void)> cmdCircle;
        RTT::Command<bool(void)> cmdEndCircle;

     public:
        CircleGen(std::string name);
        bool configureHook();
        bool startHook();
        virtual void updateHook();
        void stopHook();
        void cleanupHook();

     private:
        KDL::Frame cartframe;
        KDL::Frame center;
        void printMethodFunc();
        void resetMethodFunc();
        bool circFunc();
        bool circDone();
        bool circling;
        bool stopAtZero;
        bool funcMoveTo( KDL::Frame, double );
        bool moveDone();
        bool endCircFunc();
        bool circEnded();
        RTT::TimeService::Seconds startTime;
        RTT::TimeService::Seconds now;
        double th, th_prev;
        KDL::Vector vec;
};

} // namespace SSL

#endif // SSL_SHAPE_GENERATORS_H_
```

h/simarm.h

```
/*
 * An orocos component (SimArm) that pretends to be a real arm for testing purposes.
 * Currently supports only speed control.
 *
 */

#ifndef SSL_SIMARM_H_
#define SSL_SIMARM_H_

#include<rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
//using namespace RTT;

namespace SSL
{

     class SimArm : public RTT::TaskContext
     {
       protected:
               RTT::Property<int> numServos;
               RTT::Property< std::vector<double> > home;
               RTT::Property<int> ControlMode;
               RTT::DataPort< std::vector<double> > DriveValue;
               RTT::DataPort< std::vector<double> > SensorValue;
               RTT::DataPort< std::vector<double> > Temperature;

       public:
               SimArm(std::string name);
               bool configureHook();
               bool startHook();
               virtual void updateHook();
               void stopHook();
               void cleanupHook();

       private:
               std::vector<double> positions;
               std::vector<double> speeds;
     };

} // namespace SSL


#endif // SSL_SIMARM_H_
```

h/simarm_naxes.h

```
/*
 * An orocos component that connects to nAxesGeneratorPos and uses it
 * to simulate an arm in position control mode.  Unlike SimArm, this
 * works only in position mode (UM=5), but is more realistic in that
 * it does not instantaneously and discontinuously do exactly what
 * you tell it to.
 *
 */

#ifndef SSL_SIMARM_NAXES_H_
#define SSL_SIMARM_NAXES_H_

#include<rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/Activity.hpp>
#include <ocl/nAxesGeneratorPos.hpp>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
//using namespace RTT;

namespace SSL
{
     class SimArm_nAxes : public RTT::TaskContext
     {
       protected:
               RTT::Property<int> numServos;
               RTT::Property< std::vector<double> > home;
               RTT::Property<int> ControlMode;
               RTT::DataPort< std::vector<double> > DriveValue;
               RTT::DataPort< std::vector<double> > SensorValue;
               RTT::DataPort< std::vector<double> > PathPort;
               RTT::DataPort< std::vector<double> > Temperature;
               RTT::Method< bool(std::vector<double>) > recalibrateJoints;

       public:
               SimArm_nAxes(std::string name);
               bool configureHook();
               bool startHook();
               virtual void updateHook();
               void stopHook();
               void cleanupHook();

       private:
               std::vector<double> positions;
               std::vector<double> speeds;
               std::vector<double> lastcommand;
               RTT::Command<bool(std::vector<double>,double)> cmd_moveTo;
               RTT::Method<void(void)> meth_reset;
               bool recalibrateFunc( std::vector<double> vec );
     };

} // namespace SSL

#endif // SSL_SIMARM_NAXES_H_
```

h/switch.h

```
/*
 * An orocos component (ControlSwitch) to enable runtime switching
 * between joint- and cartesian-space control.
 *
 */

#ifndef SSL_CTRLSWITCH_H_
#define SSL_CTRLSWITCH_H_

#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/Logger.hpp>
#include <rtt/TimeService.hpp>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
//using namespace RTT;

namespace SSL
{

     class ControlSwitch : public RTT::TaskContext
     {
       protected:
               RTT::DataPort< std::vector<double> > CartSystem;
               RTT::DataPort< std::vector<double> > JointSystem;
               RTT::DataPort< std::vector<double> > ControlOutput;
               RTT::Attribute<int> mode;
               RTT::Method<bool(void)> cartStart;
               RTT::Method<bool(void)> cartStop;
               RTT::Method<bool(void)> jointStart;
               RTT::Method<bool(void)> jointStop;
               RTT::Method<bool(void)> openSwitch;

       public:
               ControlSwitch(std::string name);
               bool configureHook();
               bool startHook();
               virtual void updateHook();
               void stopHook();
               void cleanupHook();

       private:
               RTT::TaskContext* ptr;
               RTT::StateMachinePtr JsWsInstance;
               RTT::Method<bool(void)> meth_ikStart;
               RTT::Method<bool(void)> meth_ikStop;
               RTT::Method<bool(void)> meth_pgStart;
               RTT::Method<bool(void)> meth_pgStop;
               RTT::Method<bool(void)> meth_jgStart;
               RTT::Method<bool(void)> meth_jgStop;
               bool once;
               bool cartStartFn();
               bool cartStopFn();
               bool jointStartFn();
               bool jointStopFn();
               bool openSwitchFn();
               std::vector<double> emptyvector;
               int count;
     };

} // namespace SSL


#endif // SSL_CTRLSWITCH_H_
```

```
/*
 * An orocos component (WsLimiterCartPos) to restrict the Cartesian
 * positions that the end effector is allowed to take on.  The
 * property Bounds specifies overall x, y, and z limits.  Property
 * Boxes specifies no-fly zones internal to those workspace bounds.
 * It will try to stop the robot before it enters an illegal pose,
 * but if it makes it into a disallowed region then there one must
 * switch to joint space control to get out of it.
 */

#ifndef SSL_WSLIMITER_H_
#define SSL_WSLIMITER_H_

#include <rtt/os/main.h>
#include <rtt/PeriodicActivity.hpp>
#include <rtt/Ports.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/Command.hpp>
#include <kdl/frames.hpp>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
//using namespace RTT;


namespace SSL
{
     class WsLimiterCart : public RTT::TaskContext
     {
       protected:
               RTT::Property< std::vector<double> > Bounds;
               RTT::Property< std::vector<double> > Boxes;
               RTT::Event<void(void)> limitevent;
               RTT::DataPort< KDL::Frame > SetPointPos; // input
               RTT::DataPort< KDL::Frame > FkPos; // input
               RTT::DataPort< KDL::Frame > LimitedPos; // output

       public:
               WsLimiterCart(std::string name);
               bool configureHook();
               bool startHook();
               virtual void updateHook();
               void stopHook();
               void cleanupHook();

       private:
               KDL::Frame spframe, currframe;
               bool fired;
               bool ok;
               int numBoxes;
               RTT::Method<void(void)> methPG_reset;
     };

} // namespace SSL

#endif // SSL_WSLIMITER_H_
```

Appendix C: Source Files

src/elmoarm2.cpp

```
#include <elmoarm2.h>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::ElmoArm2 )
#endif


#ifndef PI
#define PI 3.14159265358979323846264338
#endif


     SSL::ElmoArm2::ElmoArm2(std::string name) :
        TaskContext(name, PreOperational),
        numServos("NumberOfServos","Number of servos"),
        home("home","home positions"),
        ControlMode("ControlMode", "Control Mode"),
        EncoderCountsPerRev("EncoderCountsPerRev","Encoder counts per joint
revolution"),
        NodeIDarr("NodeIDarr", "Node ID arrays"),
        CANdevice("CANdevice", "CAN interface device, e.g. /dev/pcan0"),
        DriveValue("DriveValue"),
        SensorValue("EncoderReading"),
        Temperature("Temperature"),
        recalibrateJoints("recalibrateJoints", &SSL::ElmoArm2::recalibrateFunc, this),
        printMethod("print", &SSL::ElmoArm2::printMethodFunc, this)
     {
        // Add the attributes and ports
        this->properties()->addProperty( &numServos );
        this->properties()->addProperty( &home );
        this->properties()->addProperty( &ControlMode );
        this->properties()->addProperty( &EncoderCountsPerRev );
        this->properties()->addProperty( &NodeIDarr );
        this->properties()->addProperty( &CANdevice );
        this->ports()->addPort( &DriveValue, "DriveValue");
        this->ports()->addPort( &SensorValue, "EncoderReading");
        this->ports()->addPort( &Temperature, "Temperature");
        this->methods()->addMethod( &recalibrateJoints, "recalibrateJoints", "vec",
"new numbers for current pose" );
        this->methods()->addMethod( &printMethod, "print" );
     }



     bool SSL::ElmoArm2::configureHook()
     {
        temp_counter = 0; // temporarily used to print the first few messages received
                          //              to the terminal for debugging purposes.

        // Initialize the CAN interface
        // (Right now it's hard-coded for 500 kbps and standard CAN frames)
        CANhandle = LINUX_CAN_Open( CANdevice.get().c_str(), O_RDWR );
        if ( (NULL == CANhandle) ||
(CAN_Init(CANhandle,CAN_BAUD_500K,CAN_INIT_TYPE_ST)<0) )
        {
               RTT::Logger::log() << RTT::Logger::Error << "[ElmoArm2] Error
initializing CAN interface" << CANdevice.get() << RTT::Logger::endl;
```

```
                return false;
        }
        // Could call CAN_Status to clear any errors in the status


        // Start all servo drives - Without this, they won't respond to any messages
        msg_out.MSGTYPE = MSGTYPE_STANDARD;
        msg_out.ID = 0x00; // NMT message COB-ID
        msg_out.LEN = 2; // NMT message length
        msg_out.DATA[0] = 0x01; // Command 0x01: Start remote node (go to operational
start)
        msg_out.DATA[1] = 0x00; // Node-ID 0x00: All connected servo drives
        CAN_Write(CANhandle, &(msg_out));
        usleep(12e3);


        // Prepare for specified number of servo drives
        printf(" %d\n", numServos.get() );
        assert( NodeIDarr.get().size() == numServos.get() );
        statuses = new ElmoArmStatus[numServos.get()];
        drivevalue = new double[numServos.get()];
        prevdrivevalue = new double[numServos.get()];
        PTindex = new char[numServos.get()];
        for (int i=0; i<numServos.get(); i++)
        {

                // Prepare status structure
                if(  (NodeIDarr.get()[i] >= -0.4)  &&  (NodeIDarr.get()[i] < 127.4)  )
                        statuses[i].nodeID = (int)(NodeIDarr.get()[i]+0.5);
                else
                {
                        RTT::Logger::log() << RTT::Logger::Error << "[ElmoArm2] NodeID
" << NodeIDarr.get()[i] << " (joint " << i+1 << ") out of range" << RTT::Logger::endl;
                        return false;
                }

                statuses[i].BG = 0;
                statuses[i].wait_BG = false;
                statuses[i].time_BG = NULL;

                statuses[i].IQ = 0;
                statuses[i].wait_IQ = false;
                statuses[i].time_IQ = NULL;

                statuses[i].JV = 0;
                statuses[i].wait_JV = true;
                statuses[i].time_JV = NULL;

                statuses[i].PA = 0;
                statuses[i].wait_PA = true;
                statuses[i].time_PA = NULL;

                statuses[i].PX = 0;
                statuses[i].wait_PX = true;
                statuses[i].time_PX = NULL;
                statuses[i].tally_PX = 0;

                statuses[i].TC = 0;
                statuses[i].wait_TC = true;
                statuses[i].time_TC = NULL;
                statuses[i].tally_TC = 0;

                statuses[i].TI = -100;
                statuses[i].wait_TI = false;
                statuses[i].time_TI = NULL;

                statuses[i].VX = 0;
                statuses[i].wait_VX = true;
                statuses[i].time_VX = NULL;
                statuses[i].tally_VX = 0;
```

```
            statuses[i].PT = 1;
            statuses[i].prevPT = 1;
            statuses[i].wait_PT = true;
            statuses[i].time_PT = NULL;
            statuses[i].tally_PT = 0;

            statuses[i].UM = 0;

            statuses[i].PTmap = false;

            statuses[i].buffer[0] = false;
            statuses[i].buffer[1] = false;
            statuses[i].buffer[2] = false;
            statuses[i].buffer[3] = false;
            statuses[i].buffer[4] = false;
            statuses[i].buffer[5] = false;

            statuses[i].overrun = false;

            // Inquire as to the node's Unit Mode.  If it's just a
            // pretend node, pretend it has the correct mode.
            // (Pretend nodes are only available in position mode.)
            if (0 == statuses[i].nodeID)
                    statuses[i].UM = ELMO_ARM_POSITION_CONTROL;
            else
                    CANquery( CANhandle, statuses[i].nodeID, 'U', 'M', 0 );


            drivevalue[i] = home.get()[i];
            }

    output.resize(numServos.get());
    output = home.get();
    SensorValue.Set( output );

    // Don't let the configuration finish until we've received
    // a response to our Unit Mode inquiry from all nodes.
    for ( int i = 0; i < numServos.get(); i++)
    {
            cycle = 0;
            while( 0 == statuses[i].UM )
            {
                    process_messages();
                    usleep(3e3);
                    if(cycle++ > 10)
                    {
                            RTT::Logger::log() << RTT::Logger::Error << "[ElmoArm2]
Timed out waiting for Unit Mode " << NodeIDarr.get()[i] << " (joint " << i+1 << ")" <<
RTT::Logger::endl;
                            return false;
                    }
            }
    }

    return true;
  }



  bool SSL::ElmoArm2::startHook()
  {
    // Confirm we're set to a valid control mode
    switch ( ControlMode.get() )
    {
            case ELMO_ARM_TORQUE_CONTROL:
            case ELMO_ARM_SPEED_CONTROL:
            case ELMO_ARM_POSITION_CONTROL:
                    break;

            default:
                    return false;
```

```
        }


        // Request necessary data and enable motor
        output.resize(numServos.get());
        for (int i=0; i<numServos.get(); i++)
        {
                drivevalue[i] = 0;
                PTindex[i] = 1;

                // Confirm we're in the right control mode before turning the motor on
                if ( ControlMode.get() != statuses[i].UM )
                {
                        intCANset( CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'M', 'O',
0, 0); // Motor off
                        intCANset( CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'U', 'M',
0, ControlMode.get());
                        usleep(100e3);
                }
                intCANset( CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'M', 'O', 0, 1);

                // Give the servo drive a chance to catch up
                //usleep(100e3);

                // Request any necessary data
                CANquery( CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'V', 'X', 0);
                CANquery( CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'P', 'X', 0);
                usleep(6e3);
                CANquery( CANhandle, statuses[i].nodeID, 'U', 'M', 0 );

                if( ELMO_ARM_SPEED_CONTROL == ControlMode.get() )
                {
                        CANquery( CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'J', 'V',
0);
                        statuses[i].wait_JV = true;
                        usleep(1e3);
                }
                //CANquery( CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'T', 'C', 0);

                // If it's a pretend node, pretend it responded with the necessary data
                if ( 0 == statuses[i].nodeID )
                {
                        statuses[i].PX = 0;
                        statuses[i].wait_PX = false;
                        statuses[i].VX = 0;
                        statuses[i].wait_VX = false;
                }


                // Let's wait for the responses
                //printf("RC's %d %d %d %d\n", rc[0], rc[1], rc[2], rc[3]);
                printf("Waiting for responses to initial data queries...\n");
                cycle = 0;
                while( statuses[i].wait_PX )
                {
                        process_messages();
                        printf("Jnt %d: PX %d; PA %d; VX %d; JV
%d\n",(int)(NodeIDarr.get()[i]+0.5), statuses[i].wait_PX, statuses[i].wait_PA,
statuses[i].wait_VX, statuses[i].wait_JV);
                        usleep(3e3);
                        if(cycle++ > 4)
                        {
                                RTT::Logger::log() << RTT::Logger::Error << "[ElmoArm2]
Timed out waiting for NodeID " << NodeIDarr.get()[i] << " (joint " << i+1 << ")" <<
RTT::Logger::endl;
                                return false;
                        }
                }
                printf("Done.\n");

                output[i] = statuses[i].PX/EncoderCountsPerRev.get()[i]*2.0*M_PI +
home.get()[i];
                usleep(12000);

                prevdrivevalue[i] = statuses[i].PX;
        }

        for (int i=0; i<numServos.get(); i++)
        {
                if ( ControlMode.get() != statuses[i].UM )
                {
                        RTT::Logger::log() << RTT::Logger::Warning << "[ElmoArm2] Unit
mode mismatch on NodeID " << NodeIDarr.get()[i] << " (joint " << i+1 << ")" <<
RTT::Logger::endl;
                        return false;
                }
        }

        SensorValue.Set( output );
        //usleep(1000);

        cycle = 0;

        if ( ELMO_ARM_POSITION_CONTROL == ControlMode.get() )
                once = true;
        else
                once = false;

        return true;
     }



     void SSL::ElmoArm2::updateHook()
     {
        command = false;

        if( once )
        {
                printf("ElmoArm2 initializing PT buffered trajectory...\n");

                switch(cycle)
                {
                        case 0:
                                for( int i=0; i<numServos.get(); i++)
                                {
                                        intCANset( CANhandle, statuses[i].nodeID, 'M',
'P', 5, 1); // EMCY underflow message at 1 setpoint
                                        intCANset( CANhandle, statuses[i].nodeID, 'M',
'P', 3, 1); // cyclical PT mode
                                }
                                break;
                        case 1:
                                for( int i=0; i<numServos.get(); i++)
                                {
                                        intCANset( CANhandle, statuses[i].nodeID, 'M',
'P', 6, 1); // set write pointer to index 1
                                        intCANset( CANhandle, statuses[i].nodeID, 'M',
'P', 4, (int)(this->getPeriod()*3.0/0.000360+0.5) ); // Default Whistle position
sampling period is 360 microseconds.
                                        //printf(" Setting MP[4] = %d\n", (int)(this-
>getPeriod()*3.0/0.000360));
                                }
                                break;
                        case 2:
                                for( int i=0; i<numServos.get(); i++)
                                {
                                        intCANset( CANhandle, statuses[i].nodeID, 'M',
'P', 1, 1); // first index of PT buffer
                                        intCANset( CANhandle, statuses[i].nodeID, 'M',
'P', 2, 6); // last index
                                }
```

```
                                        break;
                                case 3:
                                        for( int i=0; i<numServos.get(); i++)
                                        {
                                                for(int j=1; j<=6; j++)
                                                {
                                                        intCANset( CANhandle, statuses[i].nodeID,
'Q', 'T', j, (int)(this->getPeriod()*3*1000+0.5));
                                                        intCANset( CANhandle, statuses[i].nodeID,
'Q', 'P', j, statuses[i].PX );
                                                        statuses[i].buffer[j] = true;
                                                }

                                                PTindex[i] = 6;
                                                statuses[i].buffer[5] = false; // sixth element
ready for new data
                                                statuses[i].overrun = false;
                                                statuses[i].underrun = false;
                                        }
                                        break;
                                case 4:
                                        for( int i=0; i<numServos.get(); i++)
                                        {
                                                intCANset( CANhandle, statuses[i].nodeID, 'P',
'T', 0, 1);
                                                CANquery( CANhandle, statuses[i].nodeID, 'B',
'G', 0);
                                        }

                                        if( ControlMode.get() == ELMO_ARM_POSITION_CONTROL )
                                                for( int i=0; i<numServos.get(); i++)
                                                        statuses[i].buffer[0] = false;

                                        once = false;
                                        break;
                                case 5:
                                        printf("Elmoarm2: Huh?\n");
                                        once = false;
                                        break;
                                default:
                                        RTT::Logger::log() << RTT::Logger::Error << "[ElmoArm2]
Got lost in the startup procedures." << RTT::Logger::endl;
                                        break;
                        }

                        cycle = cycle+1;
                        return;
                }

                // Process incoming messages
                process_messages();

                // Update the the outputs
                for( int i = 0; i < numServos.get(); i++)
                {
                        // If it's a pretend node, pretend it's at the right position.
                        if (0==statuses[i].nodeID)
                        {
                                statuses[i].wait_PX = false;
                                statuses[i].PX = drivevalue[i];

                                statuses[i].wait_VX = false;
                                statuses[i].wait_JV = false;
                        }

                        if (statuses[i].wait_PX)
                                RTT::Logger::log() << RTT::Logger::Warning << "Unacknowledged
PX request on joint " << i+1 << RTT::Logger::endl;

                        output[i] = statuses[i].PX/EncoderCountsPerRev.get()[i]*2.0*M_PI +
home.get()[i];
```

```
        }
        SensorValue.Set( output ); // Joint position readings
        for ( int i = 0; i < numServos.get(); i++)
                output[i] = statuses[i].TI;
        Temperature.Set( output ); // Temperature readings


        switch (ControlMode.get())
        {
                case ELMO_ARM_TORQUE_CONTROL:

                        switch (cycle)
                        {
                                case 0: // Send latest drive value
                                        if( DriveValue.Get().size() == numServos.get() )
                                                for( int i=0; i<numServos.get(); i++)
                                                {
                                                        // Update the drive speed.
Unlike speed (JV) and position (PA) commands,
                                                        // torque (TC) goes into effect
immediately, without BG activation.
                                                        drivevalue[i] =
DriveValue.Get()[i] * EncoderCountsPerRev.get()[i] / 2.0 / PI;
                                                        floatCANset(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'T','C', 0, DriveValue.Get()[i] );
                                                        statuses[i].wait_TC = true;
                                                }
                                        else
                                                for( int i=0; i<numServos.get(); i++)
                                                {
                                                        // If we're not getting a valid
command then do nothing, I guess...
                                                        // The arm will fall down if we
do this!
                                                        floatCANset(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'T','C', 0, 0.0 );
                                                        statuses[i].wait_TC = true;
                                                }
                                        break;
                                case 1:
                                        for( int i=0; i<numServos.get(); i++)
                                        {
                                                CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'T', 'I', 1); // Temperature update
                                                statuses[i].wait_TI = true;
                                                CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'P', 'X', 0); // Position update
                                                statuses[i].wait_PX = true;
                                        }
                                        break;
                                case 2:
                                        for( int i=0; i<numServos.get(); i++)
                                        {
                                                CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'P', 'X', 0); // Position update
                                                statuses[i].wait_PX = true;
                                                CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'V', 'X', 0); // Velocity update
                                                statuses[i].wait_VX = true;
                                        }
                                        break;
                        }

                        cycle = (cycle+1)%3;

                        break;

                case ELMO_ARM_SPEED_CONTROL:

                        switch (cycle)
                        {
```

```
                        case 0: // Send latest drive value
                                if( DriveValue.Get().size() == numServos.get() )
                                        for( int i=0; i<numServos.get(); i++)
                                        {
                                                // Update the drive speed
                                                drivevalue[i] = 
DriveValue.Get()[i] * EncoderCountsPerRev.get()[i] / 2.0 / PI;
                                                intCANset(CANhandle, 
(int)(NodeIDarr.get()[i]+0.5), 'J', 'V', 0, (int)(drivevalue[i]+0.5) );
                                                CANquery( CANhandle, 
(int)(NodeIDarr.get()[i]+0.5), 'B', 'G', 0);
                                        }
                                else
                                        for( int i=0; i<numServos.get(); i++)
                                        {
                                                // Stop movement
                                                intCANset(CANhandle, 
(int)(NodeIDarr.get()[i]+0.5), 'J', 'V', 0, 0 );
                                                CANquery( CANhandle, 
(int)(NodeIDarr.get()[i]+0.5), 'B', 'G', 0);
                                        }
                                break;
                        case 1:
                                for( int i=0; i<numServos.get(); i++)
                                {
                                        CANquery(CANhandle, 
(int)(NodeIDarr.get()[i]+0.5), 'T', 'I', 1); // Temperature update
                                        statuses[i].wait_TI = true;
                                        CANquery(CANhandle, 
(int)(NodeIDarr.get()[i]+0.5), 'P', 'X', 0); // Position update
                                        statuses[i].wait_PX = true;
                                }
                                break;
                        case 2:
                                for( int i=0; i<numServos.get(); i++)
                                {
                                        CANquery(CANhandle, 
(int)(NodeIDarr.get()[i]+0.5), 'P', 'X', 0); // Position update
                                        statuses[i].wait_PX = true;
                                        CANquery(CANhandle, 
(int)(NodeIDarr.get()[i]+0.5), 'V', 'X', 0); // Velocity update
                                        statuses[i].wait_VX = true;
                                }
                                break;
                }

                cycle = (cycle+1)%3;
                break;

        case ELMO_ARM_POSITION_CONTROL:

                switch (cycle)
                {
                        case 0:
                                for(int i=0; i<numServos.get(); i++)
                                {
                                        if ( DriveValue.Get().size() > i )
                                        {
                                                prevdrivevalue[i] = 
drivevalue[i];
                                                drivevalue[i] = (
DriveValue.Get()[i] - home.get()[i]) * EncoderCountsPerRev.get()[i] / 2.0 / PI;
                                        }
                                        else
                                                drivevalue[i] = 
prevdrivevalue[i]; //statuses[i].PX;

                                        if( statuses[i].prevPT != statuses[i].PT
)
                                        {
                                                for( int j=0; j<
((statuses[i].PT-statuses[i].prevPT+6)%6); j++)

     statuses[i].buffer[(statuses[i].prevPT-1+j)%6] = false;
                                             }

                                             if ( 0 < statuses[i].nodeID) // If it's
not a pretend node
                                             {
                                                     if (
((true==statuses[i].buffer[statuses[i].PT-1])) &&
(true==statuses[i].buffer[(statuses[i].PT)%6]) &&
(true==statuses[i].buffer[(statuses[i].PT+1)%6]) &&
(true==statuses[i].buffer[(statuses[i].PT+2)%6]) ) // if the current and next 2
waypoints are valid
                                                     {
                                                             // Convert from radians to
counts, subtracting offsets from home (zero encoder
                                                             // counts corresponds to
the home position.
                                                             if( statuses[i].overrun )
// Overrun on previous cycle (average the point we should have sent with the one we
now want to send)
                                                             {

     statuses[i].overrun = false;
                                                                     drivevalue[i] =
0.5*(drivevalue[i]+prevdrivevalue[i]);
                                                             }
                                                             if ( PTindex[i] ==
statuses[i].PT )
                                                             { // BUFFER OVERRUN (don't
send, don't increment index)

     statuses[i].overrun = true;
                                                                     RTT::Logger::log()
<< RTT::Logger::Warning << "Buffer full on joint " << i+1 << RTT::Logger::endl;
                                                             } else {
                                                                     intCANset(
CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'Q', 'P', PTindex[i], floor(drivevalue[i])
);

     statuses[i].buffer[PTindex[i]-1] = true;
                                                                     PTindex[i] =
PTindex[i]%6 + 1;
                                                             }
                                                             if ( statuses[i].underrun
)
                                                             {

     statuses[i].underrun = false;
                                                                     //CANquery(
CANhandle, statuses[i].nodeID, 'B', 'G', 0);
                                                             }
                                                     } else { // BUFFER UNDERFLOW
                                                             RTT::Logger::log() <<
RTT::Logger::Warning << "Buffer underflow on joint " << i+1 << "(" <<
statuses[i].buffer[statuses[i].PT-1] << ", " << statuses[i].buffer[statuses[i].PT%6]
<< ") " << statuses[i].PT%6 << RTT::Logger::endl;
                                                             //CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'S','T',0);
                                                             statuses[i].underrun =
true;
                                                             intCANset( CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'Q', 'P', PTindex[i], floor(drivevalue[i]) );

     statuses[i].buffer[PTindex[i]-1] = true;
                                                             PTindex[i] = PTindex[i]%6
+ 1;
                                                             intCANset( CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'Q', 'P', PTindex[i], floor(drivevalue[i]) );
```

```
     statuses[i].buffer[PTindex[i]-1] = true;
                                                             PTindex[i] = PTindex[i]%6
+ 1;
                                                             command = true;

                                                             // Print status of buffer
                                                             //printf(" Joint 1 buffer:
%d %d %d %d %d (write %d) (read %d)\n\n", statuses[0].buffer[0],
statuses[0].buffer[1], statuses[0].buffer[2], statuses[0].buffer[3],
statuses[0].buffer[4], PTindex[0], statuses[0].PT);
                                                             //printf("  statuses[0].PT
= %d, statuses[0].prevPT = %d\n", statuses[0].PT, statuses[0].prevPT);
                                                     }
                                             }
                                             statuses[i].prevPT = statuses[i].PT;
                                     }
                                     break;
                             case 1:
                                     for( int i=0; i<numServos.get(); i++)
                                     {
                                             CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'T', 'I', 1); // Temperature update
                                             statuses[i].wait_TI = true;
                                             CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'P', 'T', 0); // Position-Time read index update
                                             //statuses[i].wait_VX = true;
                                     }
                                     break;
                             case 2:
                                     for( int i=0; i<numServos.get(); i++)
                                     {
                                             CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'P', 'X', 0); // Position update
                                             statuses[i].wait_PX = true;
                                             CANquery(CANhandle,
(int)(NodeIDarr.get()[i]+0.5), 'P', 'T', 0); // Position-time read index update
                                             //statuses[i].wait_VX = true;
                                     }
                                     break;
                     }
                     cycle = (cycle+1)%3;
                     break;

             default:
                     RTT::Logger::log() << RTT::Logger::Error << "ElmoArm2 in
unknown control mode!  " << RTT::Logger::endl;
                     stop(); // If we're not in a known control mode, that's bad.
                     return;
        }  // end of switch( ControlMode.get() )


     }



     void SSL::ElmoArm2::stopHook()
     {
        // Clear any error status that may exist on the CAN card.
        // Is there a way to purge the output buffer too?
        CAN_Status(CANhandle);

        // Make sure nothing is moving.  This could probably be done faster with group
ID?
        for(int i=0; i<numServos.get(); i++)
        {
                CANquery(CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'S','T',0);
                usleep(100);
        }

        // Need to get confirmation that it stopped!
        // Print last message received in case it's needed for debugging
        //printf("All servos are (supposedly) stopped.  Last message received before
stopping:\n");
        printf("\n  %c %c 0x%08x %1d ",
        (msg_in.Msg.MSGTYPE & MSGTYPE_RTR)      ? 'r' : 'm',
        (msg_in.Msg.MSGTYPE & MSGTYPE_EXTENDED) ? 'e' : 's',
        msg_in.Msg.ID,  msg_in.Msg.LEN);
        // Make sure it's not a remote frame or invalid DLC before printing contents
        if (!(msg_in.Msg.MSGTYPE & MSGTYPE_RTR)&&(msg_in.Msg.LEN<=8))
                for (int i = 0; i < msg_in.Msg.LEN; i++)
                        printf("0x%02x ", msg_in.Msg.DATA[i]);
        printf("\n\n");

        // Print status of buffer
        printf(" Joint 1 buffer: %d %d %d %d %d (write %d) (read %d)\n\n",
statuses[0].buffer[0], statuses[0].buffer[1], statuses[0].buffer[2],
statuses[0].buffer[3], statuses[0].buffer[4], PTindex[0], statuses[0].PT);
        printf("  statuses[0].PT = %d, statuses[0].prevPT = %d\n", statuses[0].PT,
statuses[0].prevPT);
     }



     void SSL::ElmoArm2::cleanupHook()
     {
        for(int i=0; i<numServos.get(); i++)
        {
                //intCANset(CANhandle, (int)(NodeIDarr.get()[i]+0.5), 'M', 'O', 0, 0);
        }
     }



     // Sends a command to the servo driver consisting of command characters C1 and
C2, and
     // a (signed) int value
     int SSL::ElmoArm2::intCANset(HANDLE& h, char nodeID, char C1, char C2, int index,
int value)
     {
        assert( 128 > index );

        // Ignore anything for nodeID zero
        if( 0 == nodeID )
                return 0;

        // Prepare message
        msg_out.MSGTYPE = MSGTYPE_STANDARD;
        msg_out.ID = 0x300 + nodeID; // 0x300 corresponds to RPDO2 message type
        msg_out.LEN = 8;
        msg_out.DATA[0] = C1;
        msg_out.DATA[1] = C2;
        msg_out.DATA[2] = index;
        msg_out.DATA[3] = 0; // Doesn’t support indices > 127.

        // Divide up value into 4 bytes, little endian (LSByte first)
        msg_out.DATA[4] = (0x000000FF & value);
        msg_out.DATA[5] = (0x0000FF00 & value)/0x100;
        msg_out.DATA[6] = (0x00FF0000 & value)/0x10000;
        msg_out.DATA[7] = (0xFF000000 & value)/0x1000000;

        // Send it
        return CAN_Write(h, &(msg_out));
     }

     int SSL::ElmoArm2::ptCANset(HANDLE& h, char nodeID, int value1, int value2)
     {
        // This function sends two consecutive waypoints (value1 and value2)
        // to the Whistle for trajectory smoothing.  This works because RPDO3
        // is mapped for this purpose during configureHook.
        // ...But it doesn't work for unknown reasons.
        // Ignore anything for nodeID zero
        if( 0 == nodeID )
                return 0;

        // Prepare message
        msg_out.MSGTYPE = MSGTYPE_STANDARD;
        msg_out.ID = 0x400 + nodeID; // 0x400 corresponds to RPDO3 message type
        msg_out.LEN = 8;

        msg_out.DATA[0] = (0x000000FF & value1);
        msg_out.DATA[1] = (0x0000FF00 & value1)/0x100;
        msg_out.DATA[2] = (0x00FF0000 & value1)/0x10000;
        msg_out.DATA[3] = (0xFF000000 & value1)/0x1000000;

        // Divide up value into 4 bytes, little endian (LSByte first)
        msg_out.DATA[4] = (0x000000FF & value2);
        msg_out.DATA[5] = (0x0000FF00 & value2)/0x100;
        msg_out.DATA[6] = (0x00FF0000 & value2)/0x10000;
        msg_out.DATA[7] = (0xFF000000 & value2)/0x1000000;

        // Send it
        return CAN_Write(h, &(msg_out));
     }

     int SSL::ElmoArm2::floatCANset(HANDLE& h, char nodeID, char C1, char C2, int
index, float value)
     {
        assert(128 > index);

        // Ignore anything for nodeID zero
        if( 0 == nodeID )
                return 0;

        // Prepare message
        msg_out.MSGTYPE = MSGTYPE_STANDARD;
        msg_out.ID = 0x300 + nodeID; // 0x300 corresponds to RPDO2 message type
        msg_out.LEN = 8;
        msg_out.DATA[0] = C1;
        msg_out.DATA[1] = C2;
        msg_out.DATA[2] = index;
        msg_out.DATA[3] = 0x80; // Support indices > 127.

        // This can't be a good idea
        void* ptr = &value;
        int ival = *((int*)ptr);

        // Divide up value into 4 bytes, little endian (LSByte first)
        msg_out.DATA[4] = (0x000000FF & ival);
        msg_out.DATA[5] = (0x0000FF00 & ival)/0x100;
        msg_out.DATA[6] = (0x00FF0000 & ival)/0x10000;
        msg_out.DATA[7] = (0xFF000000 & ival)/0x1000000;

        // Send it
        return CAN_Write(h, &(msg_out));
     }



     int SSL::ElmoArm2::CANquery(HANDLE& h, char nodeID, char C1, char C2, int index)
     {
        assert(128 > index);

        // Ignore anything for nodeID zero
        if( 0 == nodeID )
                return 0;

        // Prepare message
        msg_out.MSGTYPE = MSGTYPE_STANDARD;
        msg_out.ID = 0x300 + nodeID; // 0x300 corresponds to RPDO2 message type
        msg_out.LEN = 4;
        msg_out.DATA[0] = C1;
        msg_out.DATA[1] = C2;
        msg_out.DATA[2] = index;
        msg_out.DATA[3] = 0; // Doesn't support indices > 127.

        // Send it
        return CAN_Write(h, &(msg_out));
      }



      // Process incoming messages (called by updateHook)
      bool SSL::ElmoArm2::process_messages()
      {
        ElmoArmStatus* senderStatus = NULL;

        while( 0 == LINUX_CAN_Read_Timeout(CANhandle,&(msg_in),0) )
        {

                // Print last message received in case it's needed for debugging
                //printf("All servos are (supposedly) stopped.  Last message received
before stopping:\n");
                if( 15 > temp_counter++ )
                {
                        printf("\n  %c %c 0x%08x %1d ",
                        (msg_in.Msg.MSGTYPE & MSGTYPE_RTR)      ? 'r' : 'm',
                        (msg_in.Msg.MSGTYPE & MSGTYPE_EXTENDED) ? 'e' : 's',
                        msg_in.Msg.ID,  msg_in.Msg.LEN);
                        // Make sure it's not a remote frame or invalid DLC before
printing contents
                        if (!(msg_in.Msg.MSGTYPE & MSGTYPE_RTR)&&(msg_in.Msg.LEN<=8))
                                for (int i = 0; i < msg_in.Msg.LEN; i++)
                                        printf("0x%02x ", msg_in.Msg.DATA[i]);
                        printf("\n\n");
                }

                // Check if something is wrong
                if ( (msg_in.Msg.ID <= 0xff) && (msg_in.Msg.ID >= 0x81) ) // if EMCY
message
                {
                        // First check to see if it's something we can handle.
                        // If not, panic and log the offending message.
                        if
((0x80==msg_in.Msg.DATA[0])&&(0x83==msg_in.Msg.DATA[1])&&(0x81==msg_in.Msg.DATA[2]))
                        {
                                RTT::Logger::log() << RTT::Logger::Warning << "Ignoring
digital hall error on node-ID " << msg_in.Msg.ID-0x80 << RTT::Logger::endl;
                        } else {
                                RTT::Logger::log() << RTT::Logger::Error << "[ElmoArm2]
See p13-26 in DS301 Implementation Guide" << RTT::Logger::endl;
                                this->stop();
                                return false;
                        }
                } else if (msg_in.Msg.DATA[3] & 0x40 ) { // if error TPDO2 message
                        // First check to see if it's something we can handle.
                        // If not, panic and log the offending message.
                        RTT::Logger::log() << RTT::Logger::Error << "[ElmoArm2] See EC
in Command Reference Manual for Error TPDO" << RTT::Logger::endl;
                        this->stop();
                        return false;
                }


                // Determine who sent the message
                if ( (msg_in.Msg.ID < 0x281) || (msg_in.Msg.ID > 0x2ff) ) // if not
TPDO 2
                {
                        // There's only one non-TPDO2 we're equipped to deal with:
                        if( (msg_in.Msg.ID > 0x580) && (msg_in.Msg.ID < 0x600) ) //
RSDO
                        {
                                for (int i=0; i<numServos.get(); i++)
```

```
                                        {
                                                if ((char)(msg_in.Msg.ID-0x280) ==
statuses[i].nodeID)
                                                {
                                                        senderStatus = &(statuses[i]);
                                                        break;
                                                }
                                        }
                                        if (NULL == senderStatus)
                                                break; // If we don't know who sent it, ignore
it.

                                        // If it matches the message given on page 5-3 of
CANopen DS301 Implementation
                                        // Guide, then presumably we've successfully mapped the
PT motion command to
                                        // RPDO3.  I changed the header to 0x60 because page 4-2
indicates bits 0 to 4
                                        // are not used and always zero for an SDO download
response.
                                        if ( (0x60==msg_in.Msg.DATA[0]) &&
(0x02==msg_in.Msg.DATA[1]) && (0x16==msg_in.Msg.DATA[2]) && (0x01==msg_in.Msg.DATA[3])
)
                                                senderStatus->PTmap = true;
                                }
                                continue;
                        } else {
                                for (int i=0; i<numServos.get(); i++)
                                {
                                        if ((char)(msg_in.Msg.ID-0x280) == statuses[i].nodeID)
                                        {
                                                senderStatus = &(statuses[i]);
                                                break;
                                        }
                                }
                                if (NULL == senderStatus)
                                {
                                        printf("Ignoring message from node %d\n", msg_in.Msg.ID-
0x280);
                                        continue; // ignore message from unknown node
                                }
                        }

                        // File away the new information
                        switch (msg_in.Msg.DATA[0])
                        {
                                case 'B':
                                        switch (msg_in.Msg.DATA[1])
                                        {
                                                case 'G': // BG - Begin Motion
                                                        senderStatus->BG = unpackData();
                                                        if(0!=senderStatus->BG)
                                                        {
                                                                // If something went wrong, it
shouldn't have gotten to this point
                                                                RTT::Logger::log() <<
RTT::Logger::Error << "[ElmoArm2] one of the joints failed to begin motion" <<
RTT::Logger::endl;
                                                                stop();
                                                                return false;
                                                        }
                                                        senderStatus->wait_BG = false;
                                                        senderStatus->time_BG = msg_in.dwTime; //
unnecessary?
                                                        break;
                                        }
                                        break;
                                case 'I':
                                        switch (msg_in.Msg.DATA[1])
                                        {
                                                case 'Q': // IQ - Reactive Current
                        senderStatus->IQ = unpackFloat();
                        senderStatus->wait_IQ = false;
                        senderStatus->time_IQ = msg_in.dwTime;
                        break;
        }
        break;
case 'J':
        switch (msg_in.Msg.DATA[1])
        {
                case 'V': // JV - Jogging Velocity
                        senderStatus->JV = unpackData();
                        senderStatus->wait_JV = false;
                        senderStatus->time_JV = msg_in.dwTime;
                        break;
        }
        break;
case 'P':
        switch (msg_in.Msg.DATA[1])
        {
                case 'A': // PA - Absolute Position
                        senderStatus->PA = unpackData();
                        senderStatus->wait_PA = false;
                        senderStatus->time_PA = msg_in.dwTime;
                        break;

                case 'T': // PT - position-time read index
                        senderStatus->PT = unpackData();
                        senderStatus->wait_PT = false;
                        senderStatus->time_PT = msg_in.dwTime;
                        break;

                case 'X': // PX - Main Position
                        senderStatus->PX = unpackData();
                        senderStatus->wait_PX = false;
                        senderStatus->time_PX = msg_in.dwTime;
                        senderStatus->tally_PX = 0;
                        break;
        }
        break;
case 'T':
        switch (msg_in.Msg.DATA[1])
        {
                case 'C': // TC - Torque Command
                        senderStatus->TC = unpackFloat();
                        senderStatus->wait_TC = false;
                        senderStatus->time_PX = msg_in.dwTime;
                        senderStatus->tally_TC = 0;
                        break;

                case 'I': // TI - Temperature array
                        senderStatus->TI = unpackData();
                        senderStatus->wait_TI = false;
                        senderStatus->time_TI = msg_in.dwTime;
                        break;
        }
        break;
case 'U':
        switch (msg_in.Msg.DATA[1])
        {
                case 'M': // UM - Unit Mode
                        senderStatus->UM = unpackData();
                        break;
        }
        break;
case 'V':
        switch (msg_in.Msg.DATA[1])
        {
                case 'X': // VX - Main Feedback Velocity
                        senderStatus->VX = unpackData();
                        senderStatus->wait_VX = false;
                        senderStatus->time_VX = msg_in.dwTime;
```

```
                                        senderStatus->tally_VX = 0;
                                        break;
                              }
                              break;
                    }
          }
          return true;
     }



     // Unpacks an integer value contained in a TPDO2.  Called by process_messages.
     // This should probably be set to disregard DATA[7] if not present so the same
     // function can be used with the ChipF40 absolute encoder readings.
     int SSL::ElmoArm2::unpackData()
     {
          return  msg_in.Msg.DATA[4]
                    + msg_in.Msg.DATA[5] * 0x100
                    + msg_in.Msg.DATA[6] * 0x10000
                    + msg_in.Msg.DATA[7] * 0x1000000;
     }


     float SSL::ElmoArm2::unpackFloat()
     {
          int value = msg_in.Msg.DATA[4]
                              + msg_in.Msg.DATA[5] * 0x100
                              + msg_in.Msg.DATA[6] * 0x10000
                              + msg_in.Msg.DATA[7] * 0x1000000;

          // This can't be a good idea:
          void* ptr = &(value);
          return (*((float*)ptr));
     }


     bool SSL::ElmoArm2::recalibrateFunc( std::vector<double> vec )
     {
          if( vec.size() != numServos.get() )
                    return false;

          for( int i=0; i<numServos.get(); i++)
                    vec[i] = vec[i] - statuses[i].PX/EncoderCountsPerRev.get()[i]*2.0*M_PI;
          home.set(vec);
          return true;
     }


     void SSL::ElmoArm2::printMethodFunc()
     {
          printf("\n");
          for( int i=0; i<numServos.get(); i++)
                    printf(" %.6f ", statuses[i].PX/EncoderCountsPerRev.get()[i]*2.0*M_PI +
home.get()[i] );
          printf("\n");
          for( int i=0; i<numServos.get(); i++)
                    printf(" %d ", statuses[i].PX );
          printf("\n\n");
          return;
     }
```

src/ethernet.cpp

```
#include <ethernet.h>

// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::EthernetInterface )
#endif


     SSL::EthernetInterface::EthernetInterface(std::string name) :
        TaskContext(name, PreOperational),
        port("port", "port number"),
        IncomingMessage("IncomingMessage", 20),
        OutgoingMessage("OutgoingMessage", 20)
     {
        // Add the attributes and ports and stuff
        this->properties()->addProperty( &port );
        this->ports()->addPort( &IncomingMessage, "IncomingMessage");
        this->ports()->addPort( &OutgoingMessage, "OutgoingMessage");
     }



     bool SSL::EthernetInterface::configureHook()
     {
        return true;
     }



     bool SSL::EthernetInterface::startHook()
     {
        myserver.Initialize(port.get(),false,true);

        return true;
     }



     void SSL::EthernetInterface::updateHook()
     {

        // Incoming Messages
        do
        {
                receivedmessage.size = myserver.Receive(receivedmessage.array, 31);
                if( 0 < receivedmessage.size )
                {
                        receivedmessage.timestamp = RTT::TimeService::Instance()-
>secondsSince(RTT::Logger::log().getReferenceTime());
                        IncomingMessage.Push(receivedmessage);
                }

                //if(31 == receivedmessage.size)
                //      printf(" Last byte = %d\n", receivedmessage.array[30]);
        } while ( 0 < receivedmessage.size );


        // Outgoing Messages
        while( OutgoingMessage.size() ) // while messages await to be sent
        {
                OutgoingMessage.Pop(sendbuffer);
                //printf("EthernetInterface transmitting...\n");
                myserver.Send( sendbuffer.array, sendbuffer.size );
        }

     }
```

```
void SSL::EthernetInterface::stopHook()
{

}



void SSL::EthernetInterface::cleanupHook()
{
  // Clean things up

}
```

src/jointlimiter.cpp

```
#include <jointlimiter.h>

// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::JointLimiter )
#endif


void SSL::JointLimiter::getLimsFunc(KDL::JntArray* qmin,KDL::JntArray* qmax)
{
     for(int i = 0; i < numServos.get(); i++)
     {
        (*qmin)(i) = LowLim.get()[i];
        (*qmax)(i) = UpLim.get()[i];
     }
     return;
}


SSL::JointLimiter::JointLimiter(std::string name) :
     TaskContext(name, PreOperational),
     numServos("NumberOfServos","Number of servos"),
     ControlMode("ControlMode", "Control Mode"),
     UpLim("UpLim","upper joint limits"),
     LowLim("LowLim","lower joint limits"),
     MaxRates("MaxRates","maximum joint rates"),
     limitevent("JointLimitEvent"),
     DriveValueRequested("DriveValueRequested"),
     DriveValue("DriveValue"),
     EncoderReading("EncoderReading"),
     getLims("getLims", &SSL::JointLimiter::getLimsFunc, this)
{
     // Add the attributes and ports
     this->properties()->addProperty( &numServos );
     this->properties()->addProperty( &ControlMode );
     this->properties()->addProperty( &UpLim );
     this->properties()->addProperty( &LowLim );
     this->properties()->addProperty( &MaxRates );
     this->events()->addEvent( &limitevent, "JointLimitEvent");
     this->ports()->addPort( &DriveValueRequested, "DriveValueRequested");
     this->ports()->addPort( &DriveValue, "DriveValue");
     this->ports()->addPort( &EncoderReading, "EncoderReading");
     this->methods()->addMethod( &getLims, "Get joint limits",
                                 "&qmin", "min vals",
                                 "&qmax", "max vals");
}


bool SSL::JointLimiter::configureHook()
{
     if (  (numServos.get() != UpLim.get().size())   ||   (numServos.get() !=
LowLim.get().size()) || (numServos.get() != MaxRates.get().size())  )
        return false;
     vec.resize( numServos.get() );
     //std::vector<double> v;
     //v.resize(numServos.get());
     //DriveValue.Set(v);
     return true;
}


bool SSL::JointLimiter::startHook()
{
     emptyvec.resize(0);
     return true;
}
```

```
void SSL::JointLimiter::updateHook()
{
     if(DriveValueRequested.Get().size() == numServos.get() )
     {
        vec = DriveValueRequested.Get();

        switch(ControlMode.get())
        {
                case 2:
                        if ( EncoderReading.Get().size() == numServos.get() )
                        {

                                for(int i=0; i<numServos.get(); i++)
                                {
                                        if ( (EncoderReading.Get()[i] >= UpLim.get()[i])
&& (vec[i] > 0) ) // Enforce Upper Limit
                                        {
                                                vec[i] = 0.0;
                                                limitevent();
                                        }
                                        if ( (EncoderReading.Get()[i] <=
LowLim.get()[i]) && (vec[i] < 0) ) // Enforce Lower Limit
                                        {
                                                vec[i] = 0.0;
                                                limitevent();
                                        }
                                        if ( fabs(vec[i]) > MaxRates.get()[i] ) //
Enforce Maximum Rate
                                                vec[i] = MaxRates.get()[i] * (vec[i] < 0?
-1.0 : 1.0);
                                }

                        }

                        break;
                case 5:
                        for(int i=0; i<numServos.get(); i++)
                        {

                                if( vec[i] >= UpLim.get()[i] ) // Enforce Upper Limit
                                {
                                        vec[i] = UpLim.get()[i];
                                        limitevent();
                                }
                                if( vec[i] <= LowLim.get()[i] ) // Enforce Lower Limit
                                {
                                        vec[i] = LowLim.get()[i];
                                        limitevent();
                                }

                        }
                        break;
                default:
                        assert(false);
        }

        DriveValue.Set(vec);
     } else {
        DriveValue.Set(emptyvec);
     }
}


void SSL::JointLimiter::stopHook()
{

}
```

```
void SSL::JointLimiter::cleanupHook()
{
     // Clean things up
}
```

src/kinematics.cpp

```
#include <kinematics.h>
#include <iostream>
#include <kdl/frames_io.hpp>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::ForwardKinematics )
ORO_LIST_COMPONENT_TYPE( SSL::InverseVelKinematics )
ORO_LIST_COMPONENT_TYPE( SSL::InversePosKinematics )
#endif


SSL::ForwardKinematics::ForwardKinematics(std::string name) :
     TaskContext(name, PreOperational),
     numJoints("numJoints","Number of joints"),
     DHparams("DHparams","D-H Parameters"),
     EndEffDH("EndEffDH","End effector D-H"),
     JointPoses("JointPoses"),
     CartesianPose("CartesianPose"),
     printMethod("printMethod", &ForwardKinematics::printMethodFunc, this)
{
     this->ports()->addPort( &JointPoses, "JointPoses" );
     this->ports()->addPort( &CartesianPose, "CartesianPose" );
     this->properties()->addProperty( &numJoints );
     this->properties()->addProperty( &DHparams );
     this->properties()->addProperty( &EndEffDH );
     printf(" bool addMethod = %d\n", this->methods()->addMethod( &printMethod,
"printMethod" ) );
}


// For debugging purposes only
void SSL::ForwardKinematics::printMethodFunc()
{
     std::cout << std::endl << cartframe << std::endl;
     for ( int i=0; i<numJoints.get(); i++)
       printf(" %.6f", JointPoses.Get()[i]);
     printf("\n");
     return;
}


bool SSL::ForwardKinematics::configureHook()
{
     // Set up the arm geometry and initialize to specified home position
     std::vector<double> myarray(numJoints.get());
     armChain = new KDL::Chain;
     if ( (DHparams.get().size() != 5*numJoints.get()) || (EndEffDH.get().size() != 4)
)
       return false;
     if (DHparams.get()[0])
     {
       armChain->addSegment( KDL::Segment(KDL::Joint::None, KDL::Frame::DH_Craig1989(
DHparams.get()[1],DHparams.get()[2],0,DHparams.get()[4])));
       myarray[0] = DHparams.get()[3];
     }
     else
     {
       armChain->addSegment( KDL::Segment(KDL::Joint::None, KDL::Frame::DH_Craig1989(
DHparams.get()[1],DHparams.get()[2],DHparams.get()[3],0)));
       myarray[0] = DHparams.get()[4];
     }
     for(int i=1; i<numJoints.get(); i++)
     {
       if (DHparams.get()[5*(i-1)])
```

```
        {
                armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::TransZ),
KDL::Frame(KDL::Frame::DH_Craig1989(
DHparams.get()[5*i+1],DHparams.get()[5*i+2],0,DHparams.get()[5*i+4])))  );
                myarray[i] = DHparams.get()[5*i+3];
        }
        else
        {
                armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::RotZ  ),
KDL::Frame(KDL::Frame::DH_Craig1989(
DHparams.get()[5*i+1],DHparams.get()[5*i+2],DHparams.get()[5*i+3],0)))  );
                myarray[i] = DHparams.get()[5*i+4];
        }
     }
     if (DHparams.get()[5*(6-1)])
        armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::TransZ),
KDL::Frame::DH_Craig1989( EndEffDH.get()[0], EndEffDH.get()[1], EndEffDH.get()[2],
EndEffDH.get()[3]))  );
     else
        armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::RotZ  ),
KDL::Frame::DH_Craig1989( EndEffDH.get()[0], EndEffDH.get()[1], EndEffDH.get()[2],
EndEffDH.get()[3]))  );
     fksolver = new KDL::ChainFkSolverPos_recursive(*armChain);

     return true;
}


bool SSL::ForwardKinematics::startHook()
{
     jntarr.resize(numJoints.get());
     return true;
}


void SSL::ForwardKinematics::updateHook()
{
     if (JointPoses.Get().size())
     {
        for(int i=0; i<numJoints.get(); i++)
                jntarr(i) = JointPoses.Get()[i];

        // Use the forward kinematics solver to update the output cartesian frame
        if ( fksolver->JntToCart(jntarr,cartframe) >= 0 )
        {
                CartesianPose.Set(cartframe);
        } else {
                // Do something to deal with the problem?
                // If forward kinematics fail, something is very wrong.
        }
     }

     return;
}


void SSL::ForwardKinematics::stopHook()
{
     // No preparation needed
     return;
}


void SSL::ForwardKinematics::cleanupHook()
{
     // Undo whatever got done in configureHook
     delete armChain;
     delete fksolver;
     return;
}
```

```
SSL::InverseVelKinematics::InverseVelKinematics(std::string name) :
     TaskContext(name, PreOperational),
     numJoints("numJoints","Number of joints"),
     DHparams("DHparams", "D-H parameters"),
     EndEffDH("EndEffDH","End effector D-H"),
     printMethod("printMethod", &InverseVelKinematics::printMethodFunc, this),
     JointPoses("JointPoses"),
     DesiredTwist("DesiredTwist"),
     JointVelocities("JointVelocities")
{
     this->properties()->addProperty( &numJoints );
     this->properties()->addProperty( &DHparams );
     this->properties()->addProperty( &EndEffDH );
     this->methods()->addMethod( &printMethod, "printMethod");
     this->ports()->addPort( &JointPoses, "JointPoses" );
     this->ports()->addPort( &DesiredTwist, "DesiredTwist" );
     this->ports()->addPort( &JointVelocities, "JointVelocities" );
}


// For debugging purposes only
void SSL::InverseVelKinematics::printMethodFunc()
{
     printf(" des_trans_vel = [ %f %f %f ]\n",DesiredTwist.Get().vel.x(),
DesiredTwist.Get().vel.y(), DesiredTwist.Get().vel.z() );

     printf(" qdot = [ ");
     for ( int i=0; i<numJoints.get(); i++)
        printf("%f ",qdot(i));
     printf("]\n");
     return;
}


bool SSL::InverseVelKinematics::configureHook()
{
     // Set up the arm geometry and initialize to specified home position
     armChain = new KDL::Chain;
     if ( (DHparams.get().size() != 5*numJoints.get()) || (EndEffDH.get().size() != 4)
)
        return false;
     if (DHparams.get()[0])
     {
        armChain->addSegment( KDL::Segment(KDL::Joint::None, KDL::Frame::DH_Craig1989(
DHparams.get()[1],DHparams.get()[2],0,DHparams.get()[4])));
     }
     else
     {
        armChain->addSegment( KDL::Segment(KDL::Joint::None, KDL::Frame::DH_Craig1989(
DHparams.get()[1],DHparams.get()[2],DHparams.get()[3],0)));
     }
     for(int i=1; i<numJoints.get(); i++)
     {
        if (DHparams.get()[5*(i-1)])
        {
              armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::TransZ),
KDL::Frame(KDL::Frame::DH_Craig1989(
DHparams.get()[5*i+1],DHparams.get()[5*i+2],0,DHparams.get()[5*i+4]))) );
        }
        else
        {
              armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::RotZ  ),
KDL::Frame(KDL::Frame::DH_Craig1989(
DHparams.get()[5*i+1],DHparams.get()[5*i+2],DHparams.get()[5*i+3],0))) );
        }
     }
     if (DHparams.get()[5*(6-1)])
```

```
        armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::TransZ),
KDL::Frame::DH_Craig1989( EndEffDH.get()[0], EndEffDH.get()[1], EndEffDH.get()[2],
EndEffDH.get()[3])) );
     else
        armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::RotZ  ),
KDL::Frame::DH_Craig1989( EndEffDH.get()[0], EndEffDH.get()[1], EndEffDH.get()[2],
EndEffDH.get()[3])) );
     iksolver = new KDL::ChainIkSolverVel_pinv(*armChain);

     return true;
}


bool SSL::InverseVelKinematics::startHook()
{
     std::vector <double> jointposes(3);
     //JointPoses.Set( jointposes );
     jntarr.resize(numJoints.get());
     qdot.resize(numJoints.get());
     v.resize(numJoints.get());
     return true;
}


void SSL::InverseVelKinematics::updateHook()
{
     for(int i=0; i<numJoints.get(); i++)
        jntarr(i) = JointPoses.Get()[i];

     // Use the forward kinematics solver to update the output cartesian frame
     if ( iksolver->CartToJnt(jntarr,DesiredTwist.Get(),qdot) >= 0 )
     {
        for( int i = 0; i<numJoints.get(); i++)
                v[i] = qdot(i);
        JointVelocities.Set(v);
     } else {
        // Do something to deal with the problem?
        printf("Warning: Inverse kinematics failure!\n");
     }

     return;
}


void SSL::InverseVelKinematics::stopHook()
{
     // No preparation needed
     return;
}


void SSL::InverseVelKinematics::cleanupHook()
{
     // Undo whatever got done in configureHook
     delete armChain;
     delete iksolver;
     return;
}




SSL::InversePosKinematics::InversePosKinematics(std::string name) :
     TaskContext(name, PreOperational),
     numJoints("numJoints","Number of joints"),
     DHparams("DHparams", "D-H parameters"),
     EndEffDH("EndEffDH","End effector D-H"),
     printMethod("printMethod", &InversePosKinematics::printMethodFunc, this),
     divergenceEvent("InvKinDivEvent"),
     CurrentJointPose("CurrentJointPose"),
     CurrentFrame("CurrentFrame"),
     DesiredFrame("DesiredFrame"),
     NewJointPose("NewJointPose")
{
     this->properties()->addProperty( &numJoints );
     this->properties()->addProperty( &DHparams );
     this->properties()->addProperty( &EndEffDH );
     this->methods()->addMethod( &printMethod, "printMethod");
     this->events()->addEvent( &divergenceEvent, "InvKinDivEvent" );
     this->ports()->addPort( &CurrentJointPose, "CurrentJointPose" );
     this->ports()->addPort( &CurrentFrame, "CurrentFrame" );
     this->ports()->addPort( &DesiredFrame, "DesiredFrame" );
     this->ports()->addPort( &NewJointPose, "NewJointPose" );
}


// For debugging purposes only
void SSL::InversePosKinematics::printMethodFunc()
{
     /*printf(" des_trans_vel = [ %f %f %f ]\n",DesiredTwist.Get().vel.x(),
DesiredTwist.Get().vel.y(), DesiredTwist.Get().vel.z() );

     printf(" qdot = [ ");
     for ( int i=0; i<numJoints.get(); i++)
       printf("%f ",qdot(i));
     printf("]\n");*/
     printf("\nPrint method not implemented.\n");
     return;
}


bool SSL::InversePosKinematics::configureHook()
{
     // Set up the arm geometry and initialize to specified home position
     armChain = new KDL::Chain;
     if ( (DHparams.get().size() != 5*numJoints.get()) || (EndEffDH.get().size() != 4)
)
       return false;
     if (DHparams.get()[0])
     {
       armChain->addSegment( KDL::Segment(KDL::Joint::None, KDL::Frame::DH_Craig1989(
DHparams.get()[1],DHparams.get()[2],0,DHparams.get()[4])));
     }
     else
     {
       armChain->addSegment( KDL::Segment(KDL::Joint::None, KDL::Frame::DH_Craig1989(
DHparams.get()[1],DHparams.get()[2],DHparams.get()[3],0)));
     }
     for(int i=1; i<numJoints.get(); i++)
     {
       if (DHparams.get()[5*(i-1)])
       {
               armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::TransZ),
KDL::Frame(KDL::Frame::DH_Craig1989(
DHparams.get()[5*i+1],DHparams.get()[5*i+2],0,DHparams.get()[5*i+4]))) );
       }
       else
       {
               armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::RotZ  ),
KDL::Frame(KDL::Frame::DH_Craig1989(
DHparams.get()[5*i+1],DHparams.get()[5*i+2],DHparams.get()[5*i+3],0))) );
       }
     }
     if (DHparams.get()[5*(6-1)])
       armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::TransZ),
KDL::Frame::DH_Craig1989( EndEffDH.get()[0], EndEffDH.get()[1], EndEffDH.get()[2],
EndEffDH.get()[3])) );
     else
       armChain->addSegment( KDL::Segment(KDL::Joint(KDL::Joint::RotZ  ),
KDL::Frame::DH_Craig1989( EndEffDH.get()[0], EndEffDH.get()[1], EndEffDH.get()[2],
EndEffDH.get()[3])) );
```

```
     fksolver = new KDL::ChainFkSolverPos_recursive(*armChain);
     ikvelsolver = new KDL::ChainIkSolverVel_pinv(*armChain);
     iksolver = new KDL::ChainIkSolverPos_NR_JL(*armChain, qmin, qmax, *fksolver,
*ikvelsolver, 5, 1e-6); // last two args: maxiter, tolerance

     return true;
}


bool SSL::InversePosKinematics::startHook()
{
     jntarr.resize(numJoints.get());
     q.resize(numJoints.get());
     qmin.resize(numJoints.get());
     qmax.resize(numJoints.get());


     // See if we've got a joint limiter in place
     jl = getPeer("JointLimiter");
     if( NULL == jl )
     {
        // Allow full +/- pi range of motion
        for ( int i = 0; i < numJoints.get(); i++)
        {
                qmax(i) = M_PI;
                qmin(i) = -M_PI;
        }
     } else {
        // Get joint limits from the joint limiter
        jlmeth = jl->methods()-
>getMethod<void(KDL::JntArray*,KDL::JntArray*)>("getLims");
        assert(jlmeth.ready());
        jlmeth(&qmin,&qmax);
     }

     v.resize(numJoints.get());
     go = false;
     return true;
}


void SSL::InversePosKinematics::updateHook()
{
     localdesframe = DesiredFrame.Get();

     if ( (!go) && (localdesframe!=(KDL::Frame())) )
        go = true;

     if ( go && CurrentJointPose.Get().size() )
     {

        for(int i=0; i<numJoints.get(); i++)
                jntarr(i) = CurrentJointPose.Get()[i];

        // Use the inverse kinematics solver to update the desired joint pose
        iksolver->CartToJnt(jntarr,DesiredFrame.Get(),q); // returns < 0 if it doesn't
converge

        // Divergence check (if the current position is closer to the
        // desired pose than the IK solution is, disregard the IK
        // solution and just stay where we are)
        fksolver->JntToCart(q,solnframe);
        error = diff(localdesframe, solnframe);
        v[0] = error.vel.Norm() + error.rot.Norm();      // This could stand to be
tuned. Right now
        error = diff(localdesframe, CurrentFrame.Get()); // 1 meter of position error
is considered
        v[1] = error.vel.Norm() + error.rot.Norm();      // to be equal to 1 rad
orientation error.
        if ( v[1] - v[0] > -1.0e-4 )
        {
                for( int i = 0; i<numJoints.get(); i++)
                        v[i] = q(i);
        }
        else
        {
                for( int i = 0; i<numJoints.get(); i++)
                        v[i] = jntarr(i);
                divergenceEvent();

        }


        NewJointPose.Set(v);
     }

     return;
}


void SSL::InversePosKinematics::stopHook()
{
     // No preparation needed
     return;
}


void SSL::InversePosKinematics::cleanupHook()
{
     // Undo whatever got done in configureHook
     delete armChain;
     delete iksolver;
     return;
}
```

src/logger.cpp

```
#include <logger.h>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::FileLogger )
#endif


     SSL::FileLogger::FileLogger(std::string name)
        : TaskContext(name, PreOperational),
          numLogs("numLogs","number of log files"),
          Incoming("Incoming",50)
     {
        // Add the attributes and ports and stuff
        this->properties()->addProperty(&numLogs);
        this->ports()->addPort(&Incoming, "Incoming");
        pArr = NULL;
     }



     bool SSL::FileLogger::configureHook()
     {
        pArr = new FILE*[numLogs.get()+1];
        pArr[0] = NULL; // There is no log zero
        for( int i = 1; i <= numLogs.get(); i++)
        {
               sprintf( local_element.c_str, "log%d.dat", i); // uses local_element to
store filename just for a moment
               pArr[i] = fopen(local_element.c_str,"w");
               if( NULL == pArr[i] )
                       return false;
        }
        //fprintf(pFile, "Start of log\n");
        return true;
     }



     bool SSL::FileLogger::startHook()
     {
        return true;
     }


     void SSL::FileLogger::updateHook()
     {
        while ( Incoming.size() ) // while messages await in buffer
        {
               Incoming.Pop(local_element);
               //printf(" Log index %d\n",local_element.index);
               if( (0<local_element.index) && (numLogs.get()>=local_element.index) )
                       fprintf(pArr[local_element.index], "%s\n",
local_element.c_str);
        }
     }


     void SSL::FileLogger::stopHook()
     {

     }
```

```
void SSL::FileLogger::cleanupHook()
{
  for( int i = 1; i <= numLogs.get(); i++)
  {
          fflush(pArr[i]);
          fclose(pArr[i]);
  }
  delete pArr;
}
```

```
#include <messagehandler.h>

//Only need this if we're printing KDL frames for testing/debugging
//#include <kdl/frames_io.hpp>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::MessageHandler )
#endif


#define FLOATSIZE 4 // float size = 4 bytes


     SSL::MessageHandler::MessageHandler(std::string name) :
        TaskContext(name, PreOperational),
        IncomingMessage("IncomingMessage",20),
        OutgoingMessage("OutgoingMessage",20),
        Logger("Logger",50),
        CartesianPose("CartesianPose"),
        SetPointPose("SetPointPose"),
        JointPose("JointPose"),
        JointTemperature("JointTemperature"),
        DriveValue("DriveValue")
     {
        // Add the attributes and ports and stuff
        this->ports()->addPort( &IncomingMessage, "IncomingMessage");
        this->ports()->addPort( &OutgoingMessage, "OutgoingMessage");
        this->ports()->addPort( &Logger, "Logger");
        this->ports()->addPort( &CartesianPose, "CartesianPose" );
        this->ports()->addPort( &SetPointPose, "SetPointPose" );
        this->ports()->addPort( &JointPose, "JointPose" );
        this->ports()->addPort( &JointTemperature, "JointTemperature" );
        this->ports()->addPort( &DriveValue, "DriveValue" );
     }




     bool SSL::MessageHandler::configureHook()
     {
        /* Find all of our peers and grab the necessary commands/methods.
         * Some of these are optional and only produce a warning if not
         * found.
         */

        // Find the moveTo command in peer PathGenerator
        ptr = getPeer("PathGenerator");
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Error << "MessageHandler could not
find peer PathGenerator.\n";
                return false;
        }
        cmd_moveTo = ptr->commands()->getCommand<bool(KDL::Frame, double)>("moveTo");
        methPG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
        if(!cmd_moveTo.ready() || !methPG_reset.ready())
                return false;


        // Find the start and stop methods in peer Robot
        ptr = getPeer("Robot");
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Error << "MessageHandler could not
find peer Robot" << RTT::Logger::endl;
                return false;
        }
        meth_roboStart = ptr->methods()->getMethod<bool(void)>("start");
        if(!meth_roboStart.ready())
                return false;
        meth_roboStop = ptr->methods()->getMethod<bool(void)>("stop");
        if(!meth_roboStop.ready())
                return false;


        // Find workspace limiter, if present.  Otherwise, skip it.
        ptr = getPeer("WsCartLimiter");
        if ( NULL != ptr)
        {
                handleWsCartLimit = ptr->events()-
>setupConnection("WorkSpaceCartLimitEvent").callback( this,
&SSL::MessageHandler::WsCartLimitCallback, this->engine()->events() ).handle();
                if( handleWsCartLimit.ready() )
                        handleWsCartLimit.connect();
                else
                        RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler
found WsCartLimiter, but couldn't find limit event!" << RTT::Logger::endl;
                if( 0 == handleWsCartLimit)
                        RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler
couldn't connect to ws limit event." << RTT::Logger::endl;
        } else {
                RTT::Logger::log() << RTT::Logger::Warning << "WsCartLimiter not found
among MessageHandler's peers." << RTT::Logger::endl;
        }


        // Find joint limiter, if present.  Otherwise, skip it.
        ptr = getPeer("JointLimiter");
        if ( NULL != ptr)
        {
                handleJointLimit = ptr->events()-
>setupConnection("JointLimitEvent").callback( this,
&SSL::MessageHandler::JointLimitCallback, this->engine()->events() ).handle();
                if( handleJointLimit.ready() )
                        handleJointLimit.connect();
                else
                        RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler
found JointLimiter, but couldn't find limit event!" << RTT::Logger::endl;
                if( 0 == handleJointLimit) RTT::Logger::log() << RTT::Logger::Warning
<< "MessageHandler couldn't connect to joint limit event." << RTT::Logger::endl;
        } else {
                RTT::Logger::log() << RTT::Logger::Warning << "JointLimiter not found
among MessageHandler's peers." << RTT::Logger::endl;
        }


        // Find inverse kinematics if present.  Otherwise, skip it.
        ptr = getPeer("InvKin");
        if ( NULL != ptr)
        {
                handleJointLimit = ptr->events()-
>setupConnection("InvKinDivEvent").callback( this,
&SSL::MessageHandler::InvKinDivCallback, this->engine()->events() ).handle();
                if( handleJointLimit.ready() )
                        handleJointLimit.connect();
                else
                        RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler
found InvKin, but couldn't find divergence event!" << RTT::Logger::endl;
                if( 0 == handleJointLimit) RTT::Logger::log() << RTT::Logger::Warning
<< "MessageHandler couldn't connect to InvKin divergence event." << RTT::Logger::endl;
        } else {
                RTT::Logger::log() << RTT::Logger::Warning << "InvKin not found among
MessageHandler's peers." << RTT::Logger::endl;
        }
        // Find control switch, if present.  Otherwise, skip it.
        ptr = getPeer("cSwitch");
        if ( NULL != ptr)
        {
                handle_switchCM = ptr->attributes()->getAttribute<int>("mode");
                if( !handle_switchCM->ready() )
                        RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler
found cSwitch, but couldn't find mode attribute!" << RTT::Logger::endl;
        } else {
                RTT::Logger::log() << RTT::Logger::Warning << "cSwitch not found among
MessageHandler's peers." << RTT::Logger::endl;
        }


        // Find the moveTo command in peer JointGenerator, if present
        ptr = getPeer("JointGenerator");
        jntC_avail = true;
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler could not
find peer JointGenerator.\n";
                jntC_avail = false;
        } else {
                cmd_jntMoveTo = ptr->commands()->getCommand<bool(std::vector<double>,
double)>("moveTo");
                methJG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
                if(!cmd_jntMoveTo.ready() || !methJG_reset.ready())
                        jntC_avail = false;
        }

        if( !jntC_avail )
                RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler could not
establish joint control.\n";



        // Find the executeSeq command in peer MultiMan, if present
        ptr = getPeer("MultiMan");
        multipoint_avail = true; // prove otherwise
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler could not
find peer MultiMan.\n";
                multipoint_avail = false;
        } else {
                cmd_execSeq = ptr->commands()->getCommand<bool(void)>("execSequence");
                meth_stopSeq = ptr->methods()->getMethod<void(void)>("haltSequence");
                if(!cmd_execSeq.ready() || !meth_stopSeq.ready())
                        multipoint_avail = false;
                meth_setVP = ptr->methods()->getMethod< bool(KDL::Frame,double,char)
>("viaPointSet");
                meth_clearVP = ptr->methods()->getMethod< void(void) >("viaClear");
                if(!meth_setVP.ready() || !meth_clearVP.ready() )
                        multipoint_avail = false;
        }



        // Find the executeSeq command in peer MultiManJnt, if present
        ptr = getPeer("MultiManJnt");
        multipoint_avail = true; // prove otherwise
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler could not
find peer MultiManJnt.\n";
                multipoint_avail = false;
        } else {
                cmd_jntExecSeq = ptr->commands()-
>getCommand<bool(void)>("execSequence");
```

```
                meth_jntStopSeq = ptr->methods()-
>getMethod<void(void)>("haltSequence");
                meth_jntSetVP = ptr->methods()->getMethod<
bool(std::vector<double>,double,char) >("viaPointSet");
                meth_jntClearVP = ptr->methods()->getMethod< void(void) >("viaClear");
        }


        // Find the recalibrateJoints method in peer Robot, if present
        ptr = getPeer("Robot");
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler could not
find peer Robot.\n";
        } else {
                meth_recalJoints = ptr->methods()->getMethod< bool(std::vector<double>)
>("recalibrateJoints");
        }


        // All outgoing messages are 31 bytes
        outgoingmessage.size = 31;

        return true;
     }



     bool SSL::MessageHandler::startHook()
     {
        dblarr = new double[7];
        floatarr = new float[7];
        localLogElement.index = 1;
        WsCartLimit = 0;
        JointLimit = 0;
        InvKinDiv = 0;
        havegoal = false;

        logPose = -1; // Logs synchronized here.  Could stagger
        logSP =   -1; // them if it's too much at once.
        logJnt =  -1;
        logDrive= -1;

        localJnt.resize(10); // oversized
        goal_jnt.resize(6); // TEMPORARY - 6DOF shouldn't be hardcoded
        //sendNull('R','D');

        goal_time = 0.0;

        bootingup = false;

        return true;
     }



     void SSL::MessageHandler::logOutgoing() // not equipped for floats
     {
        if (Logger.ready())
        {
                snprintf( localLogElement.c_str, 99, "[%010.4f] OUT: %s",
RTT::TimeService::Instance()->secondsSince(RTT::Logger::log().getReferenceTime()),
outgoingmessage.array );
                localLogElement.index = 4;
                Logger.Push(localLogElement);
        }
        return;
     }
     void SSL::MessageHandler::updateHook()
     {
        // Relay any errors/warnings to client
        if (1 == WsCartLimit)
        {
                strcpy( outgoingmessage.array, "ER21 Work space limit.\0");
                OutgoingMessage.Push(outgoingmessage);
                logOutgoing();
                WsCartLimit++;
        }
        if (1 == JointLimit)
        {
                strcpy( outgoingmessage.array, "WN12 Joint limit.\0");
                OutgoingMessage.Push(outgoingmessage);
                logOutgoing();
                JointLimit++;
        }
        if (1 == InvKinDiv)
        {
                strcpy( outgoingmessage.array, "WN22 Inv Kin Divergence.\0");
                OutgoingMessage.Push(outgoingmessage);
                logOutgoing();
                InvKinDiv++;
        }

        if(bootingup)
        {
                methPG_reset();
                bootingup = false;
        }

        // Process any incoming messages from client
        while( IncomingMessage.size() ) // while messages await in the buffer
        {
                // Retrieve the next message and verify size
                IncomingMessage.Pop(receivedmessage);
                if (receivedmessage.size < 30)
                {
                        strcpy( outgoingmessage.array, "ER01 Msg XX undersized.\0");
                        outgoingmessage.array[9] = receivedmessage.array[0];
                        outgoingmessage.array[10] = receivedmessage.array[1];
                        OutgoingMessage.Push(outgoingmessage);
                        logOutgoing();
                        continue;
                }
                // For now we'll just ignore it if the client didn't send
                // a thirty-first byte
                if (30==receivedmessage.size)
                        receivedmessage.array[30] = 0;
                unpackFloats();

                // Log the message if possible
                if ( Logger.ready() )
                {
                        snprintf( localLogElement.c_str, 99, "[%010.4f] MSG: %c%c %.4f
%.4f %.4f %.4f %.4f %.4f %.4f (%d)", receivedmessage.timestamp,
receivedmessage.array[0],receivedmessage.array[1], dblarr[0], dblarr[1], dblarr[2],
dblarr[3], dblarr[4], dblarr[5], dblarr[6], receivedmessage.array[30]);

                        //snprintf( &(localLogElement.c_str[21]), 79, "%.4f %.4f %.4f
%.4f %.4f %.4f %.4f", dblarr[0], dblarr[1],
                        //                            dblarr[2], dblarr[3], dblarr[4],
dblarr[5], dblarr[6]);
                        localLogElement.c_str[99] = NULL; // just in case
                        localLogElement.index = 1;

                        Logger.Push(localLogElement);
                }

                // Process the message
                understood = true; // prove otherwise
                switch( receivedmessage.array[0] )
                {
                        case 'B':
                                switch( receivedmessage.array[1] )
                                {
                                        case 'G': // BG - begin motion
                                                if( !havegoal ||
!cmd_moveTo(KDL::Frame(goal_rot,goal_vec),goal_time) )
                                                {
                                                        strcpy( outgoingmessage.array,
"ER20 Cmd BG rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                        logOutgoing();
                                                        //printf("cmd_moveTo.ready() =
%d\n", cmd_moveTo.ready());
                                                } else {
                                                        sendNull('G','B',0);
                                                }
                                                break;
                                        default:
                                                understood = false;
                                                break;
                                }
                                break;
                        case 'C':
                                switch( receivedmessage.array[1] )
                                {
                                        case 'M': // CM - Command Mode
                                                if( handle_switchCM->ready() &&
jntC_avail )

                                                {
                                                        methPG_reset();
                                                        methJG_reset();
                                                        handle_switchCM-
>set((int)(dblarr[0]+0.5));

                                                        floatarr[0] = handle_switchCM-
>get();

                                                        for (int i=1; i<7; i++)
                                                                floatarr[i] = 0.0;

                                                        // Stopping a component breaks
our access to its commands/methods,
                                                        // so we must repeat the
initialization procedure (with less robust
                                                        // error checking since things
are presumably as they should be if
                                                        // we made it through the
configureHook):

                                                        switch(handle_switchCM->get())
                                                        {
                                                                case 0:
                                                                        break;
                                                                case 1:

                                                                        // Find the moveTo
command in peer JointGenerator
                                                                        ptr =
getPeer("JointGenerator");
                                                                        if(NULL == ptr)
                                                                        {

     RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler could not find peer
JointGenerator.\n";
                                                                                jntC_avail
= false;
                                                                        } else {

     cmd_jntMoveTo = ptr->commands()->getCommand<bool(std::vector<double>,
double)>("moveTo");
```

```
     methJG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
                                                                            }
                                                                            break;
                                                                    case 2:
                                                                            // Find the moveTo
command in peer PathGenerator
                                                                            ptr =
getPeer("PathGenerator");
                                                                            if(NULL == ptr)
                                                                            {

     RTT::Logger::log() << RTT::Logger::Error << "MessageHandler could not find peer
PathGenerator.\n";
                                                                            } else {
                                                                                    cmd_moveTo
= ptr->commands()->getCommand<bool(KDL::Frame, double)>("moveTo");

     methPG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
                                                                            }

                                                                            // Find inverse
kinematics
                                                                            ptr =
getPeer("InvKin");
                                                                            if ( NULL != ptr)
                                                                            {

     handleJointLimit = ptr->events()->setupConnection("InvKinDivEvent").callback(
this, &SSL::MessageHandler::InvKinDivCallback, this->engine()->events() ).handle();
                                                                                    if(
handleJointLimit.ready() )

     handleJointLimit.connect();
                                                                                    else

     RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler found InvKin, but
couldn't find divergence event!" << RTT::Logger::endl;
                                                                                    if( 0 ==
handleJointLimit) RTT::Logger::log() << RTT::Logger::Warning << "MessageHandler
couldn't connect to InvKin divergence event." << RTT::Logger::endl;
                                                                            }

                                                                            break;
                                                                    default:
                                                                            RTT::Logger::log()
<< RTT::Logger::Warning << "MessageHandler doesn't know how to repair any
commands/methods that may have been broken by CM switch." << RTT::Logger::endl;
                                                                            break;
                                                            }

                                                            sendFloats('M','C',0);
                                                    } else {
                                                            strcpy( outgoingmessage.array,
"ER20 Cmd CM rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                            logOutgoing();
                                                    }
                                                    break;
                                            default:
                                                    understood = false;
                                                    break;
                                    }
                                    break;
                            case 'G':
                                    switch( receivedmessage.array[1] )
                                    {
                                            case 'L': // GL - set goal state

                                                    // position vector
```

```
                                        goal_vec = KDL::Vector(dblarr[0],
dblarr[1], dblarr[2]); // x,y,z position

                                        // normalize the quaternion before
accepting it
                                        angle =
sqrt(dblarr[3]*dblarr[3]+dblarr[4]*dblarr[4]+dblarr[5]*dblarr[5]+dblarr[6]*dblarr[6]);
// this is not really an angle, but I see no reason to declare another variable
                                        dblarr[3] = dblarr[3]/angle;
                                        dblarr[4] = dblarr[4]/angle;
                                        dblarr[5] = dblarr[5]/angle;
                                        dblarr[6] = dblarr[6]/angle;
                                        goal_rot =
KDL::Rotation::Quaternion(dblarr[3], dblarr[4], dblarr[5], dblarr[6]); // quaternion
w,x,y,z
                                        //std::cout << " Goal rot: " << std::endl
<< KDL::Frame(goal_rot,goal_vec) << std::endl;

                                        goal_time = 0.0; // go there as fast as
possible

                                        for ( int i = 0; i < 7; i++ )
                                               floatarr[i] = dblarr[i];
                                        sendFloats('L','G',0);
                                        //printf("\n Received goal: %f, %f, %f;\n
%f, %f, %f,
%f\n",dblarr[0],dblarr[1],dblarr[2],dblarr[3],dblarr[4],dblarr[5],dblarr[6]);

                                        havegoal = true;

                                        break;
                                case 'T': // GT - set Goal pose with travel Time

                                        // position vector
                                        goal_vec = KDL::Vector(dblarr[0],
dblarr[1], dblarr[2]); // x,y,z position

                                        // orientation
                                        goal_rot =
KDL::Rotation::RPY(dblarr[3],dblarr[4],dblarr[5]); // roll, pitch, yaw

                                        // travel time
                                        if( 0.0 > dblarr[6] )
                                               dblarr[6] = 0.0;
                                        goal_time = dblarr[6];

                                        //std::cout << " Goal pose: " <<
std::endl << KDL::Frame(goal_rot,goal_vec) << std::endl;
                                        //std::cout << " Goal time: " <<
goal_time << std::endl;

                                        // Send confirmation to client
                                        for ( int i = 0; i < 7; i++ )
                                               floatarr[i] = dblarr[i];
                                        sendFloats('T','G',0);

                                        havegoal = true;
                                        break;
                                default:
                                        understood = false;
                                        break;
                        }
                        break;
                case 'j':
                        switch( receivedmessage.array[1] )
                        {
                                case 'B': // jB - Joint space Begin goal seek
                                        if (jntC_avail)
                                        {
                                               bool temp =
cmd_jntMoveTo(goal_jnt,0);
```

```
                                                        sendNull('B','j',0);
                                                        printf(" Joint command: %f %f %f
%f (size %d) (ret %c)\n", goal_jnt[0], goal_jnt[1], goal_jnt[2], goal_jnt[3],
goal_jnt.size(), (1==temp)?'1':'0');
                                                } else {
                                                        strcpy( outgoingmessage.array,
"ER20 Cmd jB rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                        logOutgoing();
                                                }
                                                break;
                                        case 'C': // jC - Clear Joint space sequence
                                                if( meth_jntClearVP.ready() )
                                                {
                                                        meth_jntClearVP();
                                                        sendNull('C','j',0);
                                                } else {
                                                        strcpy( outgoingmessage.array,
"ER20 Cmd jC rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                        logOutgoing();
                                                }
                                                break;
                                        case 'E': // jE - Execute Joint space sequence
                                                if( !cmd_jntExecSeq() )
                                                {
                                                        strcpy( outgoingmessage.array,
"ER20 Cmd jE rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                        logOutgoing();
                                                } else sendNull('E','j',0);
                                                break;
                                        case 'G': // jG - Joint space Goal
                                                if (jntC_avail)
                                                {
                                                        for (int i=0; i<6; i++)
                                                        {
                                                                goal_jnt[i] = dblarr[i];
                                                                floatarr[i] = dblarr[i];
                                                        }
                                                        sendFloats('G','j',0);
                                                } else {
                                                        strcpy( outgoingmessage.array,
"ER20 Cmd jG rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                        logOutgoing();
                                                }
                                                break;
                                        case 'P': // jP - Joint Position request
                                                localJnt = JointPose.Get();
                                                for(int i=0; i<localJnt.size(); i++)
                                                        if (i < 7)
                                                                floatarr[i] = localJnt[i];
                                                for(int i=localJnt.size(); i<7; i++)
                                                        floatarr[i] = 0.0;
                                                sendFloats('P','j',0);
                                                break;
                                        case 'R': // jR - Joint space Reset
                                                // This command is actually redundant
since ST resets both path generators
                                                if (jntC_avail)
                                                {
                                                        methJG_reset();
                                                        sendNull('R','j',0);
                                                } else {
                                                        strcpy( outgoingmessage.array,
"ER20 Cmd jR rejected\0");
```

```
     OutgoingMessage.Push(outgoingmessage);
                                                logOutgoing();
                                        }
                                        break;
                                case 'V': // jV - specify Joint space Via point
                                        for (int i=0; i<6; i++)
                                        {
                                                localJnt[i] = dblarr[i]; //
localJnt is too big but MultiManJnt ignores the extra entries
                                                floatarr[i] = dblarr[i];
                                        }
                                        floatarr[6] = dblarr[6]; // minimum
duration

                                        // Specify goal and minimum duration of
motion
                                        printf(" Received setpoint for index
%d\n", receivedmessage.array[30]); // TEMPORARY
                                        if( !meth_jntSetVP(localJnt, dblarr[6],
receivedmessage.array[30]) )
                                        {
                                                strcpy( outgoingmessage.array,
"ER20 Cmd jV rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                logOutgoing();
                                        } else
sendFloats('V','j',receivedmessage.array[30]);
                                        break;
                                default:
                                        understood = false;
                                        break;
                        }
                        break;
                case 'O':
                        switch( receivedmessage.array[1] )
                        {
                                case 'N': // ON - start robot
                                        methPG_reset(); // Makes sure there are
no old commands trying to execute
                                        if ( meth_roboStart() )
                                        {
                                                methPG_reset();
                                                bootingup = true;
                                                sendNull('N','O',0);
                                        } else  {
                                                strcpy( outgoingmessage.array,
"ER11 Arm did not start\0" );

     OutgoingMessage.Push(outgoingmessage);
                                                logOutgoing();
                                        }
                                        break;
                                case 'F': // OF - stop robot
                                        if( meth_roboStop() )
                                        {
                                                sendNull('F','O',0);
                                        } else  {
                                                strcpy( outgoingmessage.array,
"ER10 Arm did not stop\0");

     OutgoingMessage.Push(outgoingmessage);
                                                logOutgoing();
                                        }
                                        break;
                                default:
                                        understood = false;
                                        break;
                        }
                        break;
                case 'P':
                        switch( receivedmessage.array[1] )
                        {
                                case 'S': // PS - request end effector Pose
                                        currentFrame = CartesianPose.Get();

                                        // Cartesian coordinates:
                                        floatarr[0] = currentFrame.p.x();
                                        floatarr[1] = currentFrame.p.y();
                                        floatarr[2] = currentFrame.p.z();

                                        // Orientation:
                                        // KDL does have a GetQuaternion command,
but it's not shown
                                        // in the API documentation and it
requires doubles rather
                                        // than floats.  Here I use formula 2.89
from Craig (1989).
                                        angle = currentFrame.M.GetRotAngle(axis);
                                        sinhalfang = sin(angle/2);
                                        floatarr[3] = axis.x()*sinhalfang;
                                        floatarr[4] = axis.y()*sinhalfang;
                                        floatarr[5] = axis.z()*sinhalfang;
                                        floatarr[6] = cos(angle/2);

                                        sendFloats('S','P',0);
                                        break;
                                default:
                                        understood = false;
                                        break;
                        }
                        break;
                case 'R':
                        switch( receivedmessage.array[1] )
                        {
                                case 'C': // RC - Recalibrate joints
                                        vec.resize(6); // TEMPORARY - shouldn't
be hardcoded to six
                                        for( int i=0; i<6; i++)
                                                vec[i] = dblarr[i];
                                        meth_recalJoints(vec);
                                        // to do: need to reply
                                        break;
                                default:
                                        understood = false;
                                        break;
                        }
                        break;
                case 'S':
                        switch( receivedmessage.array[1] )
                        {
                                case 'T': // ST - Stop
                                        methPG_reset();
                                        if( jntC_avail )
                                                methJG_reset();
                                        if( multipoint_avail )
                                                meth_stopSeq();
                                        meth_jntStopSeq();
                                        sendNull('T','S',0);
                                        break;
                                default:
                                        understood = false;
                                        break;
                        }
                        break;
                case 'T':
                        switch( receivedmessage.array[1] )
                        {
                                case 'M': // TM - Temperature
                                        localTemperature =
JointTemperature.Get();
```

```
                                             for(int i=0; i<localTemperature.size();
i++)
                                                     if (i < 7)
                                                            floatarr[i] =
localTemperature[i];
                                             for(int i=localTemperature.size(); i<7;
i++)
                                                     floatarr[i] = -1000.0;
                                             sendFloats('M','T',0);
                                             break;
                                      default:
                                             understood = false;
                                             break;
                              }
                              break;
                       case 'V':
                              switch( receivedmessage.array[1] )
                              {
                                      case 'C': // VC - Clear Via points (really just
resets the lastIndex count)
                                             if( multipoint_avail)
                                             {
                                                     meth_clearVP();
                                                     sendNull('P','V',0);
                                             } else {
                                                     strcpy( outgoingmessage.array,
"ER20 Cmd VC rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                     logOutgoing();
                                             }
                                             break;
                                      case 'E': // VE - Execute Via point trajectory
                                             if( !multipoint_avail || !cmd_execSeq() )
                                             {
                                                     strcpy( outgoingmessage.array,
"ER20 Cmd VE rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                     logOutgoing();
                                             } else sendNull('E','V',0);
                                             break;
                                      case 'P': // VP - specify Via Point
                                             if( multipoint_avail )
                                             {
                                                     // position vector
                                                     goal_vec = KDL::Vector(dblarr[0],
dblarr[1], dblarr[2]); // x,y,z position

                                                     // orientation
                                                     goal_rot =
KDL::Rotation::RPY(dblarr[3],dblarr[4],dblarr[5]); // roll, pitch, yaw

                                                     // Specify goal frame and minimum
duration of motion
                                                     printf(" Received setpoint for
index %d\n", receivedmessage.array[30]); // TEMPORARY
                                                     if(
!meth_setVP(KDL::Frame(goal_rot,goal_vec), dblarr[6], receivedmessage.array[30]) )
                                                     {
                                                            strcpy(
outgoingmessage.array, "ER20 Cmd VP rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                            logOutgoing();
                                                     } else {
                                                            // Copy back the goal
position
                                                            floatarr[0]=goal_vec.x();
floatarr[1]=goal_vec.y(); floatarr[2]=goal_vec.z();
```

```
                                                                    // ...and orientation
                                                                    goal_rot.GetRPY(
dblarr[0], dblarr[1], dblarr[2]);
                                                                    floatarr[3]=dblarr[0];
floatarr[4]=dblarr[1]; floatarr[5]=dblarr[2];

                                                                    // ...and specified
minimum duration
                                                                    floatarr[6] = dblarr[6];


     sendFloats('P','V',receivedmessage.array[30]);
                                                            }
                                                    } else {
                                                            strcpy( outgoingmessage.array,
"ER20 Cmd VP rejected\0");

     OutgoingMessage.Push(outgoingmessage);
                                                            logOutgoing();
                                                    }
                                                    break;
                                            default:
                                                    understood = false;
                                                    break;
                                    }
                                    break;
                            default:
                                    understood = false;
                    }

                    if (!understood)
                    {
                            strcpy( outgoingmessage.array, "ER00 Cmd XX not understood\0"
);
                            outgoingmessage.array[9] = receivedmessage.array[0];
                            outgoingmessage.array[10] = receivedmessage.array[1];
                            OutgoingMessage.Push(outgoingmessage);
                            logOutgoing();
                    }
            }

            // Periodically log pose and setpoint if possible
            if ( Logger.ready() )
            {
                    if( 0 == logPose )
                    {
                            currentFrame = CartesianPose.Get();

     currentFrame.M.GetQuaternion(dblarr[0],dblarr[1],dblarr[2],dblarr[3]);
                            snprintf( localLogElement.c_str, 99, "[%010.4f] POS: %.6f %.6f
%.6f %.6f %.6f %.6f %.6f", RTT::TimeService::Instance()-
>secondsSince(RTT::Logger::log().getReferenceTime()),currentFrame.p.x(),currentFrame.p
.y(),currentFrame.p.z(),dblarr[0],dblarr[1],dblarr[2],dblarr[3]);
                            localLogElement.c_str[99] = NULL; // just in case
                            localLogElement.index = 2;
                            Logger.Push(localLogElement);
                    }
                    logPose = (logPose+1)%5;

                    if( (0==logSP)&&(SetPointPose.ready()) )
                    {
                            currentFrame = SetPointPose.Get();

     currentFrame.M.GetQuaternion(dblarr[0],dblarr[1],dblarr[2],dblarr[3]);
                            snprintf( localLogElement.c_str, 99, "[%010.4f] SP: %.6f %.6f
%.6f %.6f %.6f %.6f %.6f", RTT::TimeService::Instance()-
>secondsSince(RTT::Logger::log().getReferenceTime()),
currentFrame.p.x(),currentFrame.p.y(),currentFrame.p.z(),dblarr[0],dblarr[1],dblarr[2]
,dblarr[3]);
                            localLogElement.c_str[99] = NULL; // just in case
                            localLogElement.index = 3;
```

```
                        Logger.Push(localLogElement);
                }
                logSP = (logSP+1)%5;

                if( (0==logJnt)&&(JointPose.ready()) )
                {
                        localJnt = JointPose.Get();
                        rc = snprintf( localLogElement.c_str, 99, "[%010.4f] Jnt:",
RTT::TimeService::Instance()->secondsSince(RTT::Logger::log().getReferenceTime()));
//, localJnt[0], localJnt[1], localJnt[2], localJnt[3], localJnt[4], localJnt[5]);
                        for( int i=0; i<JointPose.Get().size(); i++)
                                rc += snprintf( &(localLogElement.c_str[rc]), 99-rc, "
%.6f", localJnt[i]);
                        localLogElement.index = 5;
                        Logger.Push(localLogElement);
                }
                logJnt = (logJnt+1)%5;

                if( (0==logDrive)&&(DriveValue.ready()) )
                {
                        vec = DriveValue.Get();
                        rc = snprintf( localLogElement.c_str, 99, "[%010.4f] Drv:",
RTT::TimeService::Instance()->secondsSince(RTT::Logger::log().getReferenceTime()));
//, localJnt[0], localJnt[1], localJnt[2], localJnt[3], localJnt[4], localJnt[5]);
                        for( int i=0; i<vec.size(); i++)
                                rc += snprintf( &(localLogElement.c_str[rc]), 99-rc, "
%.6f", vec[i]);
                        localLogElement.index = 6;
                        Logger.Push(localLogElement);
                }
                logDrive = (logDrive+1)%5;
        }

     }



     void SSL::MessageHandler::stopHook()
     {
       delete dblarr;
       delete floatarr;
     }



     void SSL::MessageHandler::cleanupHook()
     {
       // Clean things up
     }


     bool SSL::MessageHandler::unpackFloats()
     {
       // NOTE this function unpacks the floats into dblarr,
       // not floatarr.  Doubles are used internally and
       // floats only for communication with the outside
       // world.

       // This function should be called only from within updateHook.
       for (int i = 0; i < 7; i++)
       {
               pointer = receivedmessage.array + 2 + FLOATSIZE*i;
               dblarr[i] = *((float*)pointer);
       }
       return true;
     }

     bool SSL::MessageHandler::sendFloats(char c1, char c2, unsigned char index)
     {
       outgoingmessage.array[0] = c1;
       outgoingmessage.array[1] = c2;
```

```
    memcpy( &(outgoingmessage.array[2]), floatarr, 7*4 );
    outgoingmessage.array[30] = index;
    OutgoingMessage.Push(outgoingmessage);
    //logOutgoingF();
    return true;
}

bool SSL::MessageHandler::sendNull(char c1, char c2, unsigned char index)
{
    outgoingmessage.array[0] = c1;
    outgoingmessage.array[1] = c2;
    for ( int i = 2; i < 30; i++ )
            outgoingmessage.array[i] = NULL;
    outgoingmessage.array[30] = index;
    OutgoingMessage.Push(outgoingmessage);
    //logOutgoingF();
    return true;
}

bool SSL::MessageHandler::WsCartLimitCallback()
{
    WsCartLimit = (WsCartLimit + 1)%11; // modulo 11 to avoid bombarding client
    return false; // ignored
}

bool SSL::MessageHandler::JointLimitCallback()
{
    JointLimit = (JointLimit + 1)%11;
    return false; // ignored
}

bool SSL::MessageHandler::InvKinDivCallback()
{
    InvKinDiv = (InvKinDiv + 1)%11;
    return false; // ignored
}
```

src/multipoint.cpp

```
#include <multipoint.h>

// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::MultipointManager )
ORO_LIST_COMPONENT_TYPE( SSL::MultipointManagerJnt )
#endif

#define VIASIZE 20


//Only need this if we're printing KDL frames for testing/debugging
//#include <kdl/frames_io.hpp>



/***************************************************************
 *           Cartesian Space Multipoint Manager               *
 ***************************************************************/

SSL::MultipointManager::MultipointManager(std::string name) :
     TaskContext(name, PreOperational),
     //viaPoints("viaPoints"),
     //viaTimes("viaTimes"),
     //lastIndex("lastIndex"),
     execSequence("execSequence", &SSL::MultipointManager::execSeqFunc,
&SSL::MultipointManager::execSeqFinished, this),
     haltSequence("haltSequence", &SSL::MultipointManager::haltSeqFunc, this),
     viaPointSet("viaPointSet", &SSL::MultipointManager::viaPointFunc, this),
     viaClear("viaClear", &SSL::MultipointManager::viaClearFunc, this),
     CurrentPose("CurrentPose")
{
     // Add the attributes and ports
     //this->properties()->addProperty( &maxSize );
     this->commands()->addCommand( &execSequence, "beginSequence" );
     this->methods()->addMethod( &haltSequence, "haltSequence" );
     this->methods()->addMethod( &viaPointSet, "viaPointSet", "frame", "goal",
"time","minimum duration of motion", "index", "index");
     this->methods()->addMethod( &viaClear, "viaClear" );
     this->ports()->addPort( &CurrentPose, "CurrentPose" );
     //this->attributes()->addAttribute( &viaPoints );
     //this->attributes()->addAttribute( &viaTimes );
     //this->attributes()->addAttribute( &lastIndex );
}


bool SSL::MultipointManager::viaPointFunc(KDL::Frame frame, double time, char index)
{
     if ( (index < VIASIZE) && !busy )
     {
        viaPoints[index] = frame;
        viaTimes[index] = time;
        viaReady[index] = true;
        return true;
     }
     return false;
}


void SSL::MultipointManager::viaClearFunc()
{
     for(int i=0; i<VIASIZE; i++)
        viaReady[i] = false;
     return;
}
```

```
bool SSL::MultipointManager::execSeqFunc()
{
     if (!executing)
     {
       busy = true;
       for(int i=0; i<VIASIZE; i++)
       {
               if(viaReady[i])
               {
                       localViaPoints[i] = viaPoints[i];
                       localViaTimes[i] = viaTimes[i];
               } else {
                       localLastIndex = i-1;
                       break;
               }
       }
       busy = false;
       currentIndex = 1;
       executing = (0<=localLastIndex);
       if(executing)
               cmd_moveTo(localViaPoints[0], localViaTimes[0]);
       return executing;
     } else { // still executing a previous sequence
       return false;
     }
}

bool SSL::MultipointManager::execSeqFinished()
{
     return ( !executing && cmd_moveTo.done() );
}

bool SSL::MultipointManager::haltSeqFunc()
{
     executing = false;
     return true;
}


bool SSL::MultipointManager::configureHook()
{
     /*viaPoints.get().resize(20);
     viaTimes.get().resize(20);
     lastIndex.set(0);*/
     viaPoints = new KDL::Frame[VIASIZE];
     viaTimes = new double[VIASIZE];

     localViaPoints = new KDL::Frame[VIASIZE];
     localViaTimes = new double[VIASIZE];
     localLastIndex = 0;
     viaReady = new bool[VIASIZE];


     // Find the moveTo command in peer PathGenerator
     RTT::TaskContext* ptr = getPeer("PathGenerator");
     if(NULL == ptr)
     {
       RTT::Logger::log() << RTT::Logger::Error << "MultipointManager could not find
peer PathGenerator.\n";
       return false;
     }
     cmd_moveTo = ptr->commands()->getCommand<bool(KDL::Frame, double)>("moveTo");
     methPG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
     if(!cmd_moveTo.ready() || !methPG_reset.ready())
       return false;

     return true;
}
```

```
bool SSL::MultipointManager::findPG()
{
     // Find the moveTo command in peer PathGenerator
     RTT::TaskContext* ptr = getPeer("PathGenerator");
     if(NULL == ptr)
     {
        RTT::Logger::log() << RTT::Logger::Error << "MultipointManager could not find
peer PathGenerator.\n";
        return false;
     }
     cmd_moveTo = ptr->commands()->getCommand<bool(KDL::Frame, double)>("moveTo");
     methPG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
     if(!cmd_moveTo.ready() || !methPG_reset.ready())
        return false;

     return true;
}


bool SSL::MultipointManager::startHook()
{
     executing = false;
     busy = false;
     count = 0;

     for( int i=0; i<VIASIZE; i++)
     {
        viaReady[i] = false;
     }

     return true;
}


void SSL::MultipointManager::updateHook()
{
     if (executing)
     {
        if (cmd_moveTo.done()) // If the path planner has reached the current via
point
        {

                if (currentIndex > localLastIndex)
                {
                        // If we're out of via points, stop.
                        executing = false;
                } else {

                        // Check how far we are from the current via point.
                        error = diff(localViaPoints[currentIndex-1],
CurrentPose.Get());
                        normerr = error.vel.Norm() + error.rot.Norm();


                        if ( (0.002 > normerr) &&
cmd_moveTo(localViaPoints[currentIndex], localViaTimes[currentIndex]) )
                        {
                                // If we're within tolerance of the previous via point,
move on to the next.
                                printf("\nmoving to index %d\n",currentIndex); //
TEMPORARY
                                currentIndex++;
                                count = 0;
                        } else if (8<count++)  {
                                if ( 0.1 > normerr )
                                {
                                        // If we've been trying to get there for a few
cycles but
                                        // it didn't quite happen, move on anyway.
                                        printf("\nmoving to index %d\n",currentIndex);
// TEMPORARY
```

```
                                        currentIndex++;
                                        count = 0;
                                } else {
                                        // If we're nowhere near the via point, abort
the trajectory.
                                        //this->warning();
                                        executing = false;
                                }
                        }
                }


        }
     } else if (!methPG_reset.ready()) // If the path generator gets turned off and
on,
        findPG();                // then our access to its commands/methods will break.
}


void SSL::MultipointManager::stopHook()
{

}


void SSL::MultipointManager::cleanupHook()
{
     // Clean things up
     delete viaPoints;
     delete viaTimes;
     delete viaReady;
     delete localViaPoints;
     delete localViaTimes;
}





/**************************************************************
 *            Joint Space Multipoint Manager                 *
 **************************************************************/


SSL::MultipointManagerJnt::MultipointManagerJnt(std::string name) :
     TaskContext(name, PreOperational),
     execSequence("execSequence", &SSL::MultipointManagerJnt::execSeqFunc,
&SSL::MultipointManagerJnt::execSeqFinished, this),
     haltSequence("haltSequence", &SSL::MultipointManagerJnt::haltSeqFunc, this),
     viaPointSet("viaPointSet", &SSL::MultipointManagerJnt::viaPointFunc, this),
     viaClear("viaClear", &SSL::MultipointManagerJnt::viaClearFunc, this),
     CurrentPose("CurrentPose")
{
     // Add the attributes and ports
     this->commands()->addCommand( &execSequence, "beginSequence" );
     this->methods()->addMethod( &haltSequence, "haltSequence" );
     this->methods()->addMethod( &viaPointSet, "viaPointSet", "point", "goal",
"time","minimum duration of motion", "index", "index");
     this->methods()->addMethod( &viaClear, "viaClear" );
     this->ports()->addPort( &CurrentPose, "CurrentPose" );
}


bool SSL::MultipointManagerJnt::viaPointFunc(std::vector<double> point, double time,
char index)
{
     if ( point.size() < CurrentPose.Get().size() )
       return false;
     if ( point.size() > CurrentPose.Get().size() )
       point.resize( CurrentPose.Get().size() );
     if ( (index < VIASIZE) && !busy )
     {
        viaPoints[index] = point;
        viaTimes[index] = time;
        viaReady[index] = true;
        return true;
     }
     return false;
}


void SSL::MultipointManagerJnt::viaClearFunc()
{
     for(int i=0; i<VIASIZE; i++)
        viaReady[i] = false;
     return;
}


bool SSL::MultipointManagerJnt::execSeqFunc()
{
     if (!executing)
     {
        busy = true;
        for(int i=0; i<VIASIZE; i++)
        {
                if(viaReady[i])
                {
                        localViaPoints[i] = viaPoints[i];
                        localViaTimes[i] = viaTimes[i];
                } else {
                        localLastIndex = i-1;
                        break;
                }
        }
        busy = false;
        currentIndex = 1;
        executing = (0<=localLastIndex);
        if(executing)
                cmd_moveTo(localViaPoints[0], localViaTimes[0]);
        return executing;
     } else { // still executing a previous sequence
        return false;
     }
}

bool SSL::MultipointManagerJnt::execSeqFinished()
{
     return ( !executing && cmd_moveTo.done() );
}

bool SSL::MultipointManagerJnt::haltSeqFunc()
{
     executing = false;
     return true;
}


bool SSL::MultipointManagerJnt::configureHook()
{
     /*viaPoints.get().resize(20);
     viaTimes.get().resize(20);
     lastIndex.set(0);*/
     //viaPoints = new std::vector<double>[VIASIZE];
     viaPoints.resize(VIASIZE);
     viaTimes = new double[VIASIZE];

     //localViaPoints = new std::vector<double>[VIASIZE];
     localViaPoints.resize(VIASIZE);
     localViaTimes = new double[VIASIZE];
     localLastIndex = 0;
```

```
      viaReady = new bool[VIASIZE];

      // Find the moveTo command in peer PathGenerator
      RTT::TaskContext* ptr = getPeer("JointGenerator");
      if(NULL == ptr)
      {
         RTT::Logger::log() << RTT::Logger::Error << "MultipointManager could not find
peer JointGenerator.\n";
         return false;
      }
      cmd_moveTo = ptr->commands()->getCommand<bool(std::vector<double>,
double)>("moveTo");
      methPG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
      if(!cmd_moveTo.ready() || !methPG_reset.ready())
         return false;

      return true;
}


bool SSL::MultipointManagerJnt::findPG()
{
      // Find the moveTo command in peer PathGenerator
      RTT::TaskContext* ptr = getPeer("JointGenerator");
      if(NULL == ptr)
      {
         RTT::Logger::log() << RTT::Logger::Error << "MultipointManager could not find
peer JointGenerator.\n";
         return false;
      }
      cmd_moveTo = ptr->commands()->getCommand<bool(std::vector<double>,
double)>("moveTo");
      methPG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
      if(!cmd_moveTo.ready() || !methPG_reset.ready())
         return false;

      return true;
}


bool SSL::MultipointManagerJnt::startHook()
{
      executing = false;
      busy = false;
      count = 0;

      for( int i=0; i<VIASIZE; i++)
      {
         viaReady[i] = false;
      }

      return true;
}


void SSL::MultipointManagerJnt::updateHook()
{
      if (executing)
      {
         if (cmd_moveTo.done()) // If the path planner has reached the current via
point
         {

                  if (currentIndex > localLastIndex)
                  {
                          // If we're out of via points, stop.
                          executing = false;
                  } else {

                          // Check how far we are from the current via point.
                          normerr = 0;
```

```
                        for( int i=0; i<CurrentPose.Get().size(); i++)
                        {
                                error = (localViaPoints[currentIndex-1])[i] - 
CurrentPose.Get()[i];
                                normerr += error*error;
                        }
                        normerr = sqrt(normerr);


                        if ( (0.001 > normerr) && 
cmd_moveTo(localViaPoints[currentIndex], localViaTimes[currentIndex]) )
                        {
                                // If we're within tolerance of the previous via point, 
move on to the next.
                                printf("\nmoving to index %d\n",currentIndex); // 
TEMPORARY
                                currentIndex++;
                                count = 0;
                        } else if (8<count++)  {
                                if ( 0.05 > normerr )
                                {
                                        // If we've been trying to get there for a few 
cycles but
                                        // it didn't quite happen, move on anyway.
                                        printf("\nmoving to index %d\n",currentIndex); 
// TEMPORARY
                                        currentIndex++;
                                        count = 0;
                                } else {
                                        // If we're nowhere near the via point, abort 
the trajectory.
                                        //this->warning();
                                        executing = false;
                                }
                        }
                }


        }
     } else if (!methPG_reset.ready()) // If the path generator gets turned off and 
on,
        findPG();               // then our access to its commands/methods will break.
}


void SSL::MultipointManagerJnt::stopHook()
{

}


void SSL::MultipointManagerJnt::cleanupHook()
{
     // Clean things up
     viaPoints.clear();
     delete viaTimes;
     delete viaReady;
     localViaPoints.clear();
     delete localViaTimes;
}
```

```
#include <shapegens.h>
#include <iostream>
#include <kdl/frames_io.hpp>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::SinusoidGen )
ORO_LIST_COMPONENT_TYPE( SSL::CircleGen )
#endif


SSL::SinusoidGen::SinusoidGen(std::string name) :
     TaskContext(name, PreOperational),
     K("K",0.1),
     T("T",10),
     axis("axis",0),
     CartesianPosDes("CartesianDesiredPosition"),
     CartesianPosMeas("CartesianSensorPosition"),
     printMethod("printMethod", &SinusoidGen::printMethodFunc, this),
     methReset("resetPosition", &SinusoidGen::resetMethodFunc, this),
     cmdMoveTo( "moveTo", &SinusoidGen::funcMoveTo, &SinusoidGen::moveDone, this),
     cmdWave("wave", &SinusoidGen::waveFunc, &SinusoidGen::waveDone, this),
     cmdEndWave("endWave", &SinusoidGen::endWaveFunc, &SinusoidGen::waveEnded, this)
{
     this->attributes()->addAttribute( &K );
     this->attributes()->addAttribute( &T );
     this->attributes()->addAttribute( &axis );
     this->ports()->addPort( &CartesianPosDes, "CartesianDesiredPosition" );
     this->ports()->addPort( &CartesianPosMeas, "CartesianSensorPosition" );
     this->methods()->addMethod( &printMethod, "printMethod" );
     this->methods()->addMethod( &methReset, "resetPosition" );
     this->commands()->addCommand( &cmdMoveTo, "moveTo", "setpoint", "ignored",
"when", "ignored" );
     this->commands()->addCommand( &cmdWave, "wave" );
     this->commands()->addCommand( &cmdEndWave, "endWave" );
}


// For debugging purposes only
void SSL::SinusoidGen::printMethodFunc()
{
     std::cout << std::endl << cartframe << std::endl;
     return;
}


bool SSL::SinusoidGen::configureHook()
{

     return true;
}


bool SSL::SinusoidGen::startHook()
{
     zero = CartesianPosMeas.Get();
     waving = false;
     stopAtZero = false;

     switch( axis.get() )
     {
       case 0:
              vec = KDL::Vector(1,0,0);
              break;
       case 1:
               vec = KDL::Vector(0,1,0);
               break;
        case 2:
               vec = KDL::Vector(0,0,1);
               break;
        default:
               return false;
               break;
     }

     return true;
}


void SSL::SinusoidGen::updateHook()
{
     if (waving)
     {
        now = RTT::TimeService::Instance()-
>secondsSince(RTT::Logger::log().getReferenceTime());
        dx_prev = dx;
        dx = K.get() * sin( 2*M_PI/T.get() * (now - startTime) );
        cartframe = zero;

        if (stopAtZero && (dx_prev*dx <= 0))
        {
               waving = false;
               stopAtZero = false;
        } else
               cartframe.p += vec*dx;
     }
     CartesianPosDes.Set(cartframe);
     return;
}


void SSL::SinusoidGen::stopHook()
{

     return;
}


void SSL::SinusoidGen::cleanupHook()
{
     // Undo whatever got done in configureHook

     return;
}


bool SSL::SinusoidGen::waveFunc()
{
     if (!waving)
     {
        startTime = RTT::TimeService::Instance()-
>secondsSince(RTT::Logger::log().getReferenceTime());
        zero = CartesianPosMeas.Get();
        waving = true;
        return true;
     } else return false;
}


bool SSL::SinusoidGen::waveDone()
{
     return true;
}


bool SSL::SinusoidGen::funcMoveTo( KDL::Frame x, double t )
```

```
{
     // Ignore it
     return true;
}


bool SSL::SinusoidGen::moveDone()
{
     // The command was ignored to start with
     return true;
}

void SSL::SinusoidGen::resetMethodFunc()
{
     waving = false;
     cartframe = CartesianPosMeas.Get();
     return;
}

bool SSL::SinusoidGen::endWaveFunc()
{
     stopAtZero = true;
     return true;
}

bool SSL::SinusoidGen::waveEnded()
{
     return !waving;
}







SSL::CircleGen::CircleGen(std::string name) :
     TaskContext(name, PreOperational),
     R("R",0.1),
     T("T",10),
     axis_i("axis_i",0),
     axis_j("axis_j",1),
     CartesianPosDes("CartesianDesiredPosition"),
     CartesianPosMeas("CartesianSensorPosition"),
     printMethod("printMethod", &CircleGen::printMethodFunc, this),
     methReset("resetPosition", &CircleGen::resetMethodFunc, this),
     cmdMoveTo( "moveTo", &CircleGen::funcMoveTo, &CircleGen::moveDone, this),
     cmdCircle("circle", &CircleGen::circFunc, &CircleGen::circDone, this),
     cmdEndCircle("endCircle", &CircleGen::endCircFunc, &CircleGen::circEnded, this)
{
     this->attributes()->addAttribute( &R );
     this->attributes()->addAttribute( &T );
     this->attributes()->addAttribute( &axis_i );
     this->attributes()->addAttribute( &axis_j );
     this->ports()->addPort( &CartesianPosDes, "CartesianDesiredPosition" );
     this->ports()->addPort( &CartesianPosMeas, "CartesianSensorPosition" );
     this->methods()->addMethod( &printMethod, "printMethod" );
     this->methods()->addMethod( &methReset, "resetPosition" );
     this->commands()->addCommand( &cmdMoveTo, "moveTo", "setpoint", "ignored",
"when", "ignored" );
     this->commands()->addCommand( &cmdCircle, "circle" );
     this->commands()->addCommand( &cmdEndCircle, "endCircle" );
}


// For debugging purposes only
void SSL::CircleGen::printMethodFunc()
{
     std::cout << std::endl << cartframe << std::endl;
     return;
}
```

```
bool SSL::CircleGen::configureHook()
{

     return true;
}


bool SSL::CircleGen::startHook()
{
     switch (axis_i.get())
     {
       case 0:
       case 1:
       case 2:
               break;
       default:
               return false;
     }

     center = CartesianPosMeas.Get();
     center.p.data[axis_i.get()] -= R.get();
     circling = false;
     stopAtZero = false;

     return true;
}


void SSL::CircleGen::updateHook()
{
     if (circling)
     {
       now = RTT::TimeService::Instance()-
>secondsSince(RTT::Logger::log().getReferenceTime());
       cartframe = center;
       th_prev = th;
       th = 2*M_PI/T.get() * (now - startTime);

       if (stopAtZero && ( floor(th/2/M_PI) > (th_prev/2/M_PI)))
       {
               circling = false;
               stopAtZero = false;
               cartframe.p.data[axis_i.get()] += R.get();
       } else {
               cartframe.p.data[axis_i.get()] += R.get() * cos( th );
               cartframe.p.data[axis_j.get()] += R.get() * sin( th );
       }
     }
     CartesianPosDes.Set(cartframe);
     return;
}


void SSL::CircleGen::stopHook()
{

     return;
}


void SSL::CircleGen::cleanupHook()
{
     // Undo whatever got done in configureHook

     return;
}


bool SSL::CircleGen::circFunc()
```

```
{
     if (!circling)
     {
       startTime = RTT::TimeService::Instance()-
>secondsSince(RTT::Logger::log().getReferenceTime());
       center = CartesianPosMeas.Get();
       center.p.data[axis_i.get()] -= R.get();
       stopAtZero = false;
       circling = true;
       return true;
     } else return false;
}


bool SSL::CircleGen::circDone()
{
     return true;
}


bool SSL::CircleGen::funcMoveTo( KDL::Frame x, double t )
{
     // Ignore it
     return true;
}


bool SSL::CircleGen::moveDone()
{
     // The command was ignored to start with
     return true;
}

void SSL::CircleGen::resetMethodFunc()
{
     circling = false;
     cartframe = CartesianPosMeas.Get();
     return;
}

bool SSL::CircleGen::endCircFunc()
{
     stopAtZero = true;
     return true;
}

bool SSL::CircleGen::circEnded()
{
     return !circling;
}
```

src/simarm.cpp

```
#include <simarm.h>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::SimArm )
#endif



     SSL::SimArm::SimArm(std::string name) :
        TaskContext(name, PreOperational),
        numServos("NumberOfServos","Number of servos"),
        home("home","home positions"),
        ControlMode("ControlMode","Mode of control"),
        DriveValue("DriveValue",(std::vector<double>)0),
        SensorValue("EncoderReading"),
        Temperature("Temperature")
     {
        // Add the attributes and ports and stuff
        this->properties()->addProperty( &numServos );
        this->properties()->addProperty( &home );
        this->properties()->addProperty( &ControlMode );
        this->ports()->addPort( &DriveValue, "DriveValue");
        this->ports()->addPort( &SensorValue, "EncoderReading");
        this->ports()->addPort( &Temperature, "Temperature");
     }



     bool SSL::SimArm::configureHook()
     {
        std::vector<double> v;
        v.resize(numServos.get());

        positions.resize(numServos.get());
        speeds.resize(numServos.get());

        for( int i=0; i<numServos.get(); i++)
        {
                positions[i] = home.get()[i];
                speeds[i] = 0;
                v[i] = positions[i];
        }
        SensorValue.Set(v);

        return true;
     }



     bool SSL::SimArm::startHook()
     {
        std::vector<double> v;
        v.resize(6);

        for( int i=0; i<numServos.get(); i++)
        {
                v[i] = -100;
        }
        Temperature.Set(v);

        return true;
     }
     void SSL::SimArm::updateHook()
     {
       if( DriveValue.Get().size() > 0 )
       switch(ControlMode.get())
       {
               case 2:
                       for(int i=0; i<numServos.get(); i++)
                       {
                               if (speeds[i] == speeds[i]) // if !nan
                                       positions[i] += speeds[i] * 0.01; // presumes
100 Hz
                               speeds[i] = DriveValue.Get()[i];
                       }
                       SensorValue.Set(positions);
                       break;
               case 5:
                       for(int i=0; i<numServos.get(); i++)
                       {
                               if( DriveValue.Get()[i] == DriveValue.Get()[i] )
                                       positions[i] = DriveValue.Get()[i];
                       }
                       SensorValue.Set(positions);
                       break;
               default:
                       assert(false);
       }

     }



     void SSL::SimArm::stopHook()
     {
       printf("SimArm has stopped.\n");
     }



     void SSL::SimArm::cleanupHook()
     {
       // Clean things up

     }
```

src/simarm_naxes.cpp

```
#include <simarm_naxes.h>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::SimArm_nAxes )
#endif



     SSL::SimArm_nAxes::SimArm_nAxes(std::string name) :
        TaskContext(name, PreOperational),
        numServos("NumberOfServos","Number of servos"),
        home("home","home positions"),
        ControlMode("ControlMode","Mode of control"),
        DriveValue("DriveValue",(std::vector<double>)0),
        SensorValue("EncoderReading"),
        PathPort("PathPort"),
        Temperature("Temperature"),
        recalibrateJoints("recalibrateJoints", &SSL::SimArm_nAxes::recalibrateFunc,
this)
     {
        // Add the attributes and ports and stuff
        this->properties()->addProperty( &numServos );
        this->properties()->addProperty( &home );
        this->properties()->addProperty( &ControlMode );
        this->ports()->addPort( &DriveValue, "DriveValue");
        this->ports()->addPort( &SensorValue, "EncoderReading");
        this->ports()->addPort( &PathPort, "PathPort");
        this->ports()->addPort( &Temperature, "Temperature");
        this->methods()->addMethod( &recalibrateJoints, "recalibrateJoints", "vec",
"new numbers for current pose" );
     }



     bool SSL::SimArm_nAxes::configureHook()
     {
        if ( 5 != ControlMode.get() )
        {
                printf("Error: must be in position control (UM=5)\n");
                return false;
        }

        std::vector<double> v;
        v.resize(numServos.get());

        positions.resize(numServos.get());
        speeds.resize(numServos.get());
        lastcommand.resize(numServos.get());

        for( int i=0; i<numServos.get(); i++)
        {
                positions[i] = home.get()[i];
                speeds[i] = 0;
                v[i] = positions[i];
        }
        SensorValue.Set(v);

        PathPort.connectTo( getPeer("SimGen")->ports()-
>getPort("nAxesDesiredPosition") );
        cmd_moveTo = getPeer("SimGen")->commands()-
>getCommand<bool(std::vector<double>,double)>("moveTo");
        meth_reset = getPeer("SimGen")->methods()-
>getMethod<void(void)>("resetPosition");
```

```
  return (cmd_moveTo.ready() && meth_reset.ready());
}



bool SSL::SimArm_nAxes::startHook()
{
  std::vector<double> v;
  v.resize(numServos.get());

  for( int i=0; i<numServos.get(); i++)
  {
          v[i] = -100;
  }
  Temperature.Set(v);
  return true;
}



void SSL::SimArm_nAxes::updateHook()
{
  if( DriveValue.Get().size() > 0 )
  {
          if( DriveValue.Get() == DriveValue.Get() )
          {
                  if( lastcommand != DriveValue.Get() )
                  {
                          meth_reset();
                          cmd_moveTo(DriveValue.Get(),0.0);
                          lastcommand = DriveValue.Get();
                  }
          }

          if( PathPort.Get() == PathPort.Get() )
                  SensorValue.Set(PathPort.Get());

  }

}



void SSL::SimArm_nAxes::stopHook()
{
  printf("SimArm_nAxes has stopped.\n");
}



void SSL::SimArm_nAxes::cleanupHook()
{
  // Clean things up

}


bool SSL::SimArm_nAxes::recalibrateFunc( std::vector<double> vec )
{
  if( (vec == vec) && (vec.size()==numServos.get()) )
  {
          SensorValue.Set(vec);
          meth_reset();
          return true;
  } else return false;
}
```

```
#include <switch.h>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::ControlSwitch )
#endif



     SSL::ControlSwitch::ControlSwitch(std::string name) :
        TaskContext(name, PreOperational),
        CartSystem("CartSystem"),
        JointSystem("JointSystem"),
        ControlOutput("ControlOutput"),
        mode("mode"),
        cartStart("cartStart", &SSL::ControlSwitch::cartStartFn, this),
        cartStop("cartStop", &SSL::ControlSwitch::cartStopFn, this),
        jointStart("jointStart", &SSL::ControlSwitch::jointStartFn, this),
        jointStop("jointStop", &SSL::ControlSwitch::jointStopFn, this),
        openSwitch("openSwitch", &SSL::ControlSwitch::openSwitchFn, this)
     {
        // Add the attributes and ports and stuff
        this->attributes()->addAttribute( &mode );
        this->ports()->addPort( &CartSystem );
        this->ports()->addPort( &JointSystem );
        this->ports()->addPort( &ControlOutput );
        this->methods()->addMethod( &cartStart, "cartStart" );
        this->methods()->addMethod( &cartStop, "cartStop" );
        this->methods()->addMethod( &jointStart, "jointStart" );
        this->methods()->addMethod( &jointStop, "jointStop" );
        this->methods()->addMethod( &openSwitch, "openSwitch" );
     }



     bool SSL::ControlSwitch::configureHook()
     {
        mode.set(0);

        // Find the start and stop methods in peer InvKin.
        ptr = getPeer("InvKin");
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Error << "ControlSwitch could not
find peer InvKin" << RTT::Logger::endl;
                return false;
        }
        meth_ikStart = ptr->methods()->getMethod<bool(void)>("start");
        if(!meth_ikStart.ready())
                return false;
        meth_ikStop = ptr->methods()->getMethod<bool(void)>("stop");
        if(!meth_ikStop.ready())
                return false;


        // Find the start and stop methods in peer PathGenerator.
        ptr = getPeer("PathGenerator");
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Error << "ControlSwitch could not
find peer JointGenerator" << RTT::Logger::endl;
                return false;
        }
        meth_pgStart = ptr->methods()->getMethod<bool(void)>("start");
        if(!meth_pgStart.ready())
```

```
                return false;
        meth_pgStop = ptr->methods()->getMethod<bool(void)>("stop");
        if(!meth_pgStop.ready())
                return false;


        // Find the start and stop methods in peer JointGenerator.
        ptr = getPeer("JointGenerator");
        if(NULL == ptr)
        {
                RTT::Logger::log() << RTT::Logger::Error << "ControlSwitch could not
find peer JointGenerator" << RTT::Logger::endl;
                return false;
        }
        meth_jgStart = ptr->methods()->getMethod<bool(void)>("start");
        if(!meth_jgStart.ready())
                return false;
        meth_jgStop = ptr->methods()->getMethod<bool(void)>("stop");
        if(!meth_jgStop.ready())
                return false;


        scripting()->loadStateMachines( "./controlswitch.osd" );
        // need to add a check that the SM loaded successfully

        return true;
      }



      bool SSL::ControlSwitch::startHook()
      {
        JsWsInstance = engine()->states()->getStateMachine("JsWsInstance");
        once = true;

        mode.set(0);

        emptyvector.resize(0);

        count = 0;

        return true;
      }



      void SSL::ControlSwitch::updateHook()
      {
        if( once )
        {
                // Start up the state machine in automatic mode
                JsWsInstance->activate();
                JsWsInstance->automatic();
                once = false;
        }

        // Pass through the drive value from whichever control
        // system is active.
        if ( 0 == count)
        {
                switch( mode.get() )
                {
                        case 0:
                                ControlOutput.Set(emptyvector);
                                break;
                        case 1:
                                ControlOutput.Set(JointSystem.Get());
                                break;
                        case 2:
                                ControlOutput.Set(CartSystem.Get());
                                break;
                        default:
                                mode = 0;
                                RTT::Logger::log() << RTT::Logger::Warning <<
"ControlSwitch mode invalid. Disabling control." << RTT::Logger::endl;
                                break;
                }
        } else {
                count = (count+1)%3; // Disable motion for a couple cycles after a mode
switch
                ControlOutput.Set(emptyvector);
        }

     }



     void SSL::ControlSwitch::stopHook()
     {
        // This doesn't work because updateHook is no longer running at this point!
        engine()->states()->getStateMachine("JsWsInstance")->deactivate();
        engine()->states()->getStateMachine("JsWsInstance")->deactivate();
     }



     void SSL::ControlSwitch::cleanupHook()
     {
        // Clean things up
        engine()->states()->unloadStateMachine("JsWsInstance");
     }


     bool SSL::ControlSwitch::cartStartFn()
     {
        printf("\nAttempting to start Cartesian mode...\n");
        if( meth_ikStart.ready() && meth_pgStart.ready() )
        {
                count = 1;
                return ( meth_ikStart() && meth_pgStart() );
        } else {
                return false;
        }
     }


     bool SSL::ControlSwitch::cartStopFn()
     {
        if( meth_ikStop.ready() && meth_pgStop.ready() )
                return ( meth_ikStop() && meth_pgStop() );
        else
                return false;
     }


     bool SSL::ControlSwitch::jointStartFn()
     {
        if( meth_jgStart.ready() )
        {
                count = 1;
                return meth_jgStart();
        } else {
                return false;
        }
     }


     bool SSL::ControlSwitch::jointStopFn()
     {
        if( meth_jgStop.ready() )
                return meth_jgStop();
        else
```

```
          return false;
}


bool SSL::ControlSwitch::openSwitchFn()
{
  ControlOutput.Set(emptyvector);
  return true;
}
```

src/wslimiter.cpp

```
#include <wslimiter.h>


// For compilation of a shared object library that can be
// loaded in Deployer.
#ifdef OCL_COMPONENT_ONLY
#include <ocl/ComponentLoader.hpp>
ORO_CREATE_COMPONENT_TYPE( )
ORO_LIST_COMPONENT_TYPE( SSL::WsLimiterCart )
#endif



     SSL::WsLimiterCart::WsLimiterCart(std::string name) :
        TaskContext(name, PreOperational),
        Bounds("xyzBounds","Maximal x-y-z limits"),
        Boxes("Boxes","Internal off-limits regions"),
        limitevent("WorkSpaceCartLimitEvent"),
        SetPointPos("SetPointPos"),
        FkPos("FkPos"),
        LimitedPos("LimitedPos")
     {
        // Add the attributes and ports and stuff
        this->properties()->addProperty( &Bounds );
        this->properties()->addProperty( &Boxes );
        this->events()->addEvent( &limitevent, "WorkSpaceCartLimitEvent");
        this->ports()->addPort( &SetPointPos, "SetPointPos");
        this->ports()->addPort( &FkPos, "FkPos");
        this->ports()->addPort( &LimitedPos, "LimitedPos");
     }



     bool SSL::WsLimiterCart::configureHook()
     {
        if( (0!=Boxes.get().size()%6) || (6!=Bounds.get().size()) )
                return false;
        numBoxes = Boxes.get().size() / 6;
        RTT::TaskContext* ptr = getPeer("PathGenerator");
        if ( NULL == ptr )
                return false;
        methPG_reset = ptr->methods()->getMethod<void(void)>("resetPosition");
        return methPG_reset.ready();
     }



     bool SSL::WsLimiterCart::startHook()
     {
        fired = false;
        ok = true;
        return true;
     }



     void SSL::WsLimiterCart::updateHook()
     {
        spframe = SetPointPos.Get();
        currframe = FkPos.Get();
        ok = true; // prove otherwise

        if ( KDL::Frame() == spframe ) // if the setpoint has not been initialized
                spframe = currframe;      // then replace it with the current frame
(which had better be initialized)

        // x-y bounds
        if( (spframe.p.x()<Bounds.get()[0]) || (spframe.p.x()>Bounds.get()[1]) ||
(spframe.p.y()<Bounds.get()[2]) || (spframe.p.y()>Bounds.get()[3]) ||
(spframe.p.z()<Bounds.get()[4]) || (spframe.p.z()>Bounds.get()[5]) )
        {
                methPG_reset();
                if (!fired)
                {
                        ok = false;
                        limitevent();
                        printf(" WS LIMIT! reset.\n");
                }
        }
        if( (currframe.p.x()<Bounds.get()[0]) || (currframe.p.x()>Bounds.get()[1]) ||
(currframe.p.y()<Bounds.get()[2]) || (currframe.p.y()>Bounds.get()[3]) ||
(currframe.p.z()<Bounds.get()[4]) || (currframe.p.z()>Bounds.get()[5]) )
        {
                methPG_reset();
                if (!fired)
                {
                        ok = false;
                        limitevent();
                        printf(" WS LIMIT! reset.\n");
                }
                fired = true;
        }

        // internal boxes
        for( int i = 0; i<numBoxes; i++)
        {
                if( (spframe.p.x()>Boxes.get()[6*i]) &&
(spframe.p.x()<Boxes.get()[6*i+1]) && (spframe.p.y()>Boxes.get()[6*i+2]) &&
(spframe.p.y()<Boxes.get()[6*i+3]) && (spframe.p.z()>Boxes.get()[6*i+4]) &&
(spframe.p.z()<Boxes.get()[6*i+5]) )
                {
                        methPG_reset();
                        if (!fired)
                        {
                                ok = false;
                                limitevent();
                                printf(" WS LIMIT! reset.\n");
                        }
                }
                if( (currframe.p.x()>Boxes.get()[6*i]) &&
(currframe.p.x()<Boxes.get()[6*i+1]) && (currframe.p.y()>Boxes.get()[6*i+2]) &&
(currframe.p.y()<Boxes.get()[6*i+3]) && (currframe.p.z()>Boxes.get()[6*i+4]) &&
(currframe.p.z()<Boxes.get()[6*i+5]) )
                {
                        methPG_reset();
                        if (!fired)
                        {
                                ok = false;
                                limitevent();
                                printf(" WS LIMIT! reset.\n");
                        }
                        fired = true;
                }
        }

        if (ok)
                LimitedPos.Set(spframe);
        else
                LimitedPos.Set(currframe);

    }



    void SSL::WsLimiterCart::stopHook()
    {

    }
```

```
void SSL::WsLimiterCart::cleanupHook()
{
  // Clean things up

}
```

Appendix D: Configuration Files

Configuration files for Ranger Mark I. The XML file, presented first, lists the software components to be loaded, the interconnections to be established, and the configuration parameter files (CPF) to be loaded. The CPF files follow in alphabetical order.

config/configuration_nbv.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>


      <!-- This file instructs Deployer to load ElmoArm2 and connects it to
            the full suite of stuff for position-mode Cartesian control.

            Execute as: deployer-gnulinux -s /path/to/configuration.xml
      -->



      <!-- Where to find the shared object files -->
      <simple name="Import"
type="string"><value>/home/ndamore/ssl/rtsx/projects/orocos/bin/</value></simple>
      <simple name="Import" type="string"><value>/usr/local/lib/</value></simple>



      <!-- ROBOT -->
      <struct name="Robot" type="SSL::ElmoArm2">

            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="Activity">
                  <simple name="Period" type="double"><value>0.00648</value></simple>
                  <simple name="Priority" type="short"><value>5</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>1</value></simple>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="DriveValue" type="string">
                        <value>DriveValue</value>
                  </simple>
                  <simple name="EncoderReading" type="string">
                        <value>EncoderReading</value>
                  </simple>
                  <simple name="Temperature" type="string">
                        <value>JointTemperature</value>
                  </simple>
            </struct>

            <!-- Configure properties. -->
            <simple name="PropertyFile" type="string">

      <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/elmoarm.cpf</value>
            </simple>
      </struct>




      <!-- ETHERNET INTERFACE -->
      <struct name="etherface" type="SSL::EthernetInterface">

            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="PeriodicActivity">
                  <simple name="Period" type="double"><value>0.025</value></simple>
                  <simple name="Priority" type="short"><value>0</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>1</value></simple>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="IncomingMessage" type="string">
                        <value>IncomingMessage</value>
                  </simple>
                  <simple name="OutgoingMessage" type="string">
                        <value>OutgoingMessage</value>
                  </simple>
            </struct>

            <!-- Configure properties. -->
            <simple name="PropertyFile" type="string">

      <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/ethernet.cpf</value>
            </simple>

      </struct>




      <!-- MESSAGE INTERPRETER -->
      <struct name="interpreter" type="SSL::MessageHandler">

            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="PeriodicActivity">
                  <simple name="Period" type="double"><value>0.025</value></simple>
                  <simple name="Priority" type="short"><value>0</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>1</value></simple>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="IncomingMessage" type="string">
                        <value>IncomingMessage</value>
                  </simple>
                  <simple name="OutgoingMessage" type="string">
                        <value>OutgoingMessage</value>
                  </simple>
                  <simple name="CartesianPose" type="string">
                        <value>CartesianPos</value>
                  </simple>
                  <simple name="SetPointPose" type="string">
                        <value>SetpointPos</value>
                  </simple>
                  <simple name="JointPose" type="string">
                        <value>EncoderReading</value>
                  </simple>
                  <simple name="JointTemperature" type="string">
                        <value>JointTemperature</value>
```

```
                </simple>
                <simple name="DriveValue" type="string">
                      <value>DriveValue</value>
                </simple>
                <simple name="Logger" type="string">
                      <value>logs</value>
                </simple>
          </struct>

          <struct name="Peers" type="PropertyBag">
                <simple type="string"><value>PathGenerator</value></simple>
                <simple type="string"><value>JointGenerator</value></simple>
                <simple type="string"><value>Robot</value></simple>
                <simple type="string"><value>WsCartLimiter</value></simple>
                <simple type="string"><value>JointLimiter</value></simple>
                <simple type="string"><value>cSwitch</value></simple>
                <simple type="string"><value>InvKin</value></simple>
                <simple type="string"><value>MultiMan</value></simple>
                <simple type="string"><value>MultiManJnt</value></simple>
          </struct>

    </struct>



    <!-- Logger -->
    <struct name="logger" type="SSL::FileLogger">

          <!-- Set it up as a periodic activity.  -->
          <struct name="Activity" type="PeriodicActivity">
                <simple name="Period" type="double"><value>0.05</value></simple>
                <simple name="Priority" type="short"><value>0</value></simple>
                <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
          </struct>
          <simple name="AutoConf" type="boolean"><value>1</value></simple>
          <simple name="AutoStart" type="boolean"><value>1</value></simple>

          <!-- Configure the ports. -->
          <struct name="Ports" type="PropertyBag">
                <simple name="Incoming" type="string">
                      <value>logs</value>
                </simple>
          </struct>

          <!-- Configure properties. -->
          <simple name="PropertyFile" type="string">

    <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/logger.cpf</value>
          </simple>
    </struct>



    <!-- FORWARD KINEMATICS -->
    <struct name="myfk" type="SSL::ForwardKinematics">

          <!-- Set it up as a periodic activity.  -->
          <struct name="Activity" type="PeriodicActivity">
                <simple name="Period" type="double"><value>0.01944</value></simple>
                <simple name="Priority" type="short"><value>0</value></simple>
                <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
          </struct>
          <simple name="AutoConf" type="boolean"><value>1</value></simple>
          <simple name="AutoStart" type="boolean"><value>1</value></simple>

          <!-- Configure the ports. -->
          <struct name="Ports" type="PropertyBag">
                <simple name="JointPoses" type="string">
                      <value>EncoderReading</value>
                </simple>
                <simple name="CartesianPose" type="string">
                      <value>CartesianPos</value>
                </simple>
          </struct>

          <!-- Configure properties. -->
          <simple name="PropertyFile" type="string">

    <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/kinematics.cpf</value>
          </simple>

    </struct>




    <!-- MULTIPOINT MANAGER, CARTESIAN -->
    <struct name="MultiMan" type="SSL::MultipointManager">
          <!-- Set it up as a periodic activity.  -->
          <struct name="Activity" type="PeriodicActivity">
                <simple name="Period" type="double"><value>0.01</value></simple>
                <simple name="Priority" type="short"><value>0</value></simple>
                <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
          </struct>
          <simple name="AutoConf" type="boolean"><value>1</value></simple>
          <simple name="AutoStart" type="boolean"><value>1</value></simple>

          <struct name="Peers" type="PropertyBag">
                <simple type="string"><value>PathGenerator</value></simple>
          </struct>

          <!-- Configure the ports. -->
          <struct name="Ports" type="PropertyBag">
                <simple name="CurrentPose" type="string">
                      <value>CartesianPos</value>
                </simple>
          </struct>
    </struct>




    <!-- TRAJECTORY GENERATOR, CARTESIAN -->
    <struct name="PathGenerator" type="OCL::CartesianGeneratorPos">

          <!-- Set it up as a periodic activity.  -->
          <struct name="Activity" type="PeriodicActivity">
                <simple name="Period" type="double"><value>0.01944</value></simple>
                <simple name="Priority" type="short"><value>0</value></simple>
                <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
          </struct>
          <simple name="AutoConf" type="boolean"><value>1</value></simple>
          <simple name="AutoStart" type="boolean"><value>0</value></simple>

          <!-- Configure the ports. -->
          <struct name="Ports" type="PropertyBag">
                <simple name="CartesianSensorPosition" type="string">
                      <value>CartesianPos</value>
                </simple>
                <simple name="CartesianDesiredPosition" type="string">
                      <value>SetpointPos</value>
                </simple>
          </struct>

          <!-- Configure properties. -->
          <simple name="PropertyFile" type="string">

    <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/trajgen.cpf</value>
```

```
            </simple>


      </struct>




      <!-- WORKSPACE LIMITER -->
      <struct name="WsCartLimiter" type="SSL::WsLimiterCart">

            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="PeriodicActivity">
                  <simple name="Period" type="double"><value>0.01944</value></simple>
                  <simple name="Priority" type="short"><value>0</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>1</value></simple>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="SetPointPos" type="string">
                        <value>SetpointPos</value>
                  </simple>
                  <simple name="FkPos" type="string">
                        <value>CartesianPos</value>
                  </simple>
                  <simple name="LimitedPos" type="string">
                        <value>AllowedFrame</value>
                  </simple>
            </struct>

            <!-- Configure properties. -->
            <simple name="PropertyFile" type="string">

      <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/wslimits.cpf</value>
            </simple>

            <struct name="Peers" type="PropertyBag">
                  <simple type="string"><value>PathGenerator</value></simple>
            </struct>

      </struct>




      <!-- INVERSE KINEMATICS -->
      <struct name="InvKin" type="SSL::InversePosKinematics">

            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="PeriodicActivity">
                  <simple name="Period" type="double"><value>0.01944</value></simple>
                  <simple name="Priority" type="short"><value>0</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>0</value></simple>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="CurrentJointPose" type="string">
                        <value>EncoderReading</value>
                  </simple>
                  <simple name="DesiredFrame" type="string">
                        <value>AllowedFrame</value>
                  </simple>
                  <simple name="CurrentFrame" type="string">
```

```
                        <value>CartesianPos</value>
                  </simple>
                  <simple name="NewJointPose" type="string">
                        <value>CartDriveValue</value>
                  </simple>
            </struct>

            <!-- Configure properties. -->
            <simple name="PropertyFile" type="string">

      <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/kinematics.cpf</value>
            </simple>

            <struct name="Peers" type="PropertyBag">
                  <simple type="string"><value>JointLimiter</value></simple>
            </struct>

      </struct>



      <!-- MULTIPOINT MANAGER, JOINT SPACE -->
      <struct name="MultiManJnt" type="SSL::MultipointManagerJnt">
            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="PeriodicActivity">
                  <simple name="Period" type="double"><value>0.01</value></simple>
                  <simple name="Priority" type="short"><value>0</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>1</value></simple>

            <struct name="Peers" type="PropertyBag">
                  <simple type="string"><value>JointGenerator</value></simple>
            </struct>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="CurrentPose" type="string">
                        <value>EncoderReading</value>
                  </simple>
            </struct>
      </struct>



      <!-- TRAJECTORY GENERATOR, JOINT -->
      <struct name = "JointGenerator" type="OCL::nAxesGeneratorPos">
            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="PeriodicActivity">
                  <simple name="Period" type="double"><value>0.01944</value></simple>
                  <simple name="Priority" type="short"><value>0</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>0</value></simple>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="nAxesSensorPosition" type="string">
                        <value>EncoderReading</value>
                  </simple>
                  <simple name="nAxesDesiredPosition" type="string">
                        <value>JointDriveValue</value>
                  </simple>
            </struct>

            <!-- Configure properties. -->
            <simple name="PropertyFile" type="string">
      <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/naxestrajgen.cpf</value
>
            </simple>
      </struct>



      <!-- CONTROL SWITCH -->
      <struct name="cSwitch" type="SSL::ControlSwitch">

            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="PeriodicActivity">
                  <simple name="Period" type="double"><value>0.01944</value></simple>
                  <simple name="Priority" type="short"><value>0</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>1</value></simple>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="ControlOutput" type="string">
                        <value>DriveValueRequested</value>
                  </simple>
                  <simple name="CartSystem" type="string">
                        <value>CartDriveValue</value>
                  </simple>
                  <simple name="JointSystem" type="string">
                        <value>JointDriveValue</value>
                  </simple>
            </struct>

            <struct name="Peers" type="PropertyBag">
                  <simple type="string"><value>InvKin</value></simple>
                  <simple type="string"><value>PathGenerator</value></simple>
                  <simple type="string"><value>JointGenerator</value></simple>
            </struct>
      </struct>



      <!-- JOINT LIMITER -->
      <struct name="JointLimiter" type="SSL::JointLimiter">

            <!-- Set it up as a periodic activity.  -->
            <struct name="Activity" type="PeriodicActivity">
                  <simple name="Period" type="double"><value>0.01944</value></simple>
                  <simple name="Priority" type="short"><value>0</value></simple>
                  <simple name="Scheduler"
type="string"><value>ORO_SCHED_OTHER</value></simple>
            </struct>
            <simple name="AutoConf" type="boolean"><value>1</value></simple>
            <simple name="AutoStart" type="boolean"><value>1</value></simple>

            <!-- Configure the ports. -->
            <struct name="Ports" type="PropertyBag">
                  <simple name="DriveValue" type="string">
                        <value>DriveValue</value>
                  </simple>
                  <simple name="DriveValueRequested" type="string">
                        <value>DriveValueRequested</value>
                  </simple>
                  <simple name="EncoderReading" type="string">
                        <value>EncoderReading</value>
                  </simple>
            </struct>

            <!-- Configure properties. -->
            <simple name="PropertyFile" type="string">
      <value>/home/ndamore/ssl/rtsx/projects/orocos/config/nbv/jointlimiter.cpf</value
>
            </simple>

      </struct>

</properties>
```

config/elmoarm.cpf

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>

  <!-- Total number of servos (joints) in the kinematic chain. -->
  <simple name="NumberOfServos" type="long"><description>Number of
servos</description><value>6</value></simple>

  <!-- CAN interface device -->
  <simple name="CANdevice" type="string"><description>CAN
interface</description><value>/dev/pcan0</value></simple>

  <!-- Servo drive unit mode (UM).  1=torque, 2=speed, 5=angle. -->
  <simple name="ControlMode" type="long"><description>Control
Mode</description><value>5</value></simple>

  <!-- Encoder counts per joint revolution. -->
  <struct name="EncoderCountsPerRev" type="array">
     <description>Encoder counts per joint revolution</description>
     <simple type="double"><value>817741</value></simple>
     <simple type="double"><value>817741</value></simple>  <!--  +/- ~0.14%  -->
     <simple type="double"><value>-414246</value></simple> <!--  +/- ~0.06%  -->
     <simple type="double"><value>414246</value></simple>
     <simple type="double"><value>-414246</value></simple>
     <simple type="double"><value>414246</value></simple>
  </struct>

  <!-- Home position -->
  <struct name="home" type="array">
     <description>home position</description>
     <simple type="double"><value>1.5708</value></simple>
     <simple type="double"><value>0.0</value></simple>
     <simple type="double"><value>1.5708</value></simple>
     <simple type="double"><value>0.0</value></simple>
     <simple type="double"><value>1.5708</value></simple>
     <simple type="double"><value>0.0</value></simple>
  </struct>

  <!-- The Node-ID's to which the servo drives will respond.
       SimArm will ignore these values. -->
  <struct name="NodeIDarr" type="array">
     <description>Joint node IDs</description>
     <simple type="double"><value>101</value></simple>
     <simple type="double"><value>102</value></simple>
     <simple type="double"><value>103</value></simple>
     <simple type="double"><value>104</value></simple>
     <simple type="double"><value>105</value></simple>
     <simple type="double"><value>106</value></simple>
  </struct>

</properties>
```

config/ethernet.cpf

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>

  <!-- Port number -->
  <simple name="port"
type="long"><description>port</description><value>49151</value></simple>

</properties>
```

config/jointlimiter.cpf

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <!-- Total number of servos (joints) in the kinematic chain. -->
  <simple name="NumberOfServos" type="long"><description>Number of
servos</description><value>6</value></simple>

  <!-- Servo drive unit mode (UM).  1=torque, 2=speed, 5=angle. -->
  <simple name="ControlMode" type="long"><description>Control
Mode</description><value>5</value></simple>

  <!-- Upper Limits -->
  <struct name="UpLim" type="array">
     <description>Joint upper limits</description>
     <simple type="double"><value>4.713</value></simple>
     <simple type="double"><value>3.926</value></simple>
     <simple type="double"><value>1.600</value></simple>
     <simple type="double"><value>3.142</value></simple>
     <simple type="double"><value>3.926</value></simple>
     <simple type="double"><value>3.142</value></simple>
  </struct>

  <!-- Lower Limits -->
  <struct name="LowLim" type="array">
     <description>Joint lower limits</description>
     <simple type="double"><value>-4.713</value></simple>
     <simple type="double"><value>-0.785</value></simple>
     <simple type="double"><value>-2.350</value></simple>
     <simple type="double"><value>-3.142</value></simple>
     <simple type="double"><value>+0.785</value></simple>
     <simple type="double"><value>-3.142</value></simple>
  </struct>

  <!-- Maximum Rates -->
  <struct name="MaxRates" type="array">
     <description>Joint lower limits</description>
     <simple type="double"><value>0.2</value></simple>
     <simple type="double"><value>0.2</value></simple>
     <simple type="double"><value>0.2</value></simple>
     <simple type="double"><value>0.2</value></simple>
     <simple type="double"><value>0.2</value></simple>
     <simple type="double"><value>0.2</value></simple>
  </struct>

</properties>
```

config/kinematics.cpf

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <!-- Total number of servos (joints) in the kinematic chain. -->
  <simple name="numJoints" type="long"><description>Number of
joints</description><value>6</value></simple>

  <!-- D-H Parameters following John Craig's (1989) convention.
       {type(0=revolute,1=prismatic), a, alpha, d, theta} per joint.
       Joint variables should be set to zero here. -->
  <struct name="DHparams" type="array">
     <description>D-H Values</description>
     <simple type="double"><value>0.0</value></simple>     <!-- First joint,
0.0=revolute -->
        <simple type="double"><value>0.0</value></simple>     <!-- a_0 -->
        <simple type="double"><value>0.0</value></simple>     <!-- alpha_0 -->
        <simple type="double"><value>0.250</value></simple>    <!-- d_1 -->
        <simple type="double"><value>0.0</value></simple>     <!-- theta_1 -->
     <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>1.5708</value></simple>
        <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.0</value></simple>
     <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.558</value></simple>
        <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.0</value></simple>
     <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.152</value></simple>
        <simple type="double"><value>-1.5708</value></simple>
        <simple type="double"><value>0.538</value></simple>
        <simple type="double"><value>0.0</value></simple>
     <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>1.5708</value></simple>
        <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.0</value></simple>
     <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>1.5708</value></simple>
        <simple type="double"><value>0.0</value></simple>
        <simple type="double"><value>0.0</value></simple>
  </struct>

  <!-- Description via the same D-H convention of the end effector
        frame relative to the last joint frame. -->
  <struct name="EndEffDH" type="array">
        <description>End Effector Frame</description>
        <simple type="double"><value>0.0</value></simple>     <!-- a -->
        <simple type="double"><value>0.0</value></simple>     <!-- alpha -->
        <simple type="double"><value>0.264</value></simple>    <!-- d -->
        <simple type="double"><value>0.0</value></simple>     <!-- theta -->
  </struct>

</properties>
```

config/logger.cpf

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>

  <!-- Total number of logs. -->
  <simple name="numLogs" type="long"><description>Number of log
files</description><value>5</value></simple>

</properties>
```

config/naxestrajgen.cpf

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <simple name="num_axes" type="long"><description>Number of
servos</description><value>6</value></simple>
  <struct name="max_vel" type="array">
     <description>Maximum rotational rates</description>
     <simple type="double"><value>0.07</value></simple>
     <simple type="double"><value>0.08</value></simple>
     <simple type="double"><value>0.10</value></simple>
     <simple type="double"><value>0.10</value></simple>
     <simple type="double"><value>0.10</value></simple>
     <simple type="double"><value>0.10</value></simple>
  </struct>
  <struct name="max_acc" type="array">
     <description>Maximum accelerations</description>
     <simple type="double"><value>0.50</value></simple>
     <simple type="double"><value>0.50</value></simple>
     <simple type="double"><value>0.50</value></simple>
     <simple type="double"><value>0.50</value></simple>
     <simple type="double"><value>0.50</value></simple>
     <simple type="double"><value>0.50</value></simple>
  </struct>
</properties>
```

config/trajgen.cpf

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <struct name="max_vel" type="array">
     <description>Maximum translational/rotational rates</description>
     <simple type="double"><value>0.03</value></simple>
     <simple type="double"><value>0.03</value></simple>
     <simple type="double"><value>0.03</value></simple>
     <simple type="double"><value>0.05</value></simple>
     <simple type="double"><value>0.05</value></simple>
     <simple type="double"><value>0.05</value></simple>
  </struct>
  <struct name="max_acc" type="array">
     <description>Maximum accelerations</description>
     <simple type="double"><value>0.01</value></simple>
     <simple type="double"><value>0.01</value></simple>
     <simple type="double"><value>0.01</value></simple>
     <simple type="double"><value>0.05</value></simple>
     <simple type="double"><value>0.05</value></simple>
     <simple type="double"><value>0.05</value></simple>
  </struct>
</properties>
```

config/wslimits.cpf

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>

  <!-- Servo drive unit mode (UM).  1=torque, 2=speed, 5=angle. -->
  <simple name="ControlMode" type="long"><description>Control
Mode</description><value>5</value></simple>

  <!-- Overall x-y-z workspace bounds -->
  <struct name="xyzBounds" type="array">
     <description>x-y-z bounds</description>
     <simple type="double"><value>-0.8</value></simple>  <!-- x_min -->
     <simple type="double"><value>2.0</value></simple>   <!-- x_max -->
     <simple type="double"><value>-2.0</value></simple>  <!-- y_min -->
     <simple type="double"><value>2.0</value></simple>   <!-- y_max -->
     <simple type="double"><value>-1.0</value></simple>  <!-- z_min -->
     <simple type="double"><value>2.0</value></simple>   <!-- z_max -->
  </struct>

  <!-- Ceilings -->
  <struct name="Boxes" type="array">
     <description>internal off-limits regions</description>
     <simple type="double"><value>-0.2</value></simple>
     <simple type="double"><value>+0.2</value></simple>
     <simple type="double"><value>-0.2</value></simple>
     <simple type="double"><value>+0.2</value></simple>
     <simple type="double"><value>-0.1</value></simple>
     <simple type="double"><value>+0.4</value></simple>

  </struct>
</properties>
```

Bibliography

[1] P. Fitzpatrick, G. Metta and L. Natale, "Towards long-lived robot genes," *Robotics and Autonomous Systems*, vol. 56, 2008, pp. 29-45.

[2] I. Nesnas, et al., "CLARAty: Challenges and Steps Toward Reusable Robotic Software," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, 2006, pp. 23-29.

[3] I. Nesnas, "The CLARAty Project: Coping with Hardware and Software Heterogeneity," in *Software Engineering for Experimental Robotics*, 1st ed. D. Brugali, Ed., Berlin: Springer, 2007.

[4] I. Nesnas, "CLARAty: A Collaborative Software for Advancing Robot Technologies," *NASA Science and Technology Conference*, June, 2007.

[5] E. Coste-Maniere and R. Simmons, "Architecture, the Backbone of Robotic Systems," in *Proc. IEEE International Conference on Robotics & Automation*, 2000.

[6] D. Schmitz, et al., "CHIMERA: A Real-time Programming Environment For Manipulator Control," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 1989, pp. 846-852.

[7] D. Stewart and P. Khosla, "The Chimera Methodology: Designing Dynamically Reconfigurable Real-Time Software using Port-Based Objects," *Proceedings of WORDS '94. The First Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 46-53, 1995.

[8] D. Stewart and P. Khosla. "Chimera Home Page." Internet: http://www.cs.cmu.edu/~aml/chimera/chimera.html, [Jun 21, 2010].

[9] "Player Project." Internet: http://playerstage.sourceforge.net/, [Jun 23, 2010].

[10] "ROS/Introduction – ROS Wiki." Internet: http://www.ros.org/wiki/ROS/Introduction, May 11, 2010 [July 8, 2010].

[11] M. Quigley, et al., "ROS: an open-source Robot Operating System." Internet: www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf, [July 8, 2010].

[12] W. Meeusen, "Orocos RTT and ROS integrated | Willow Garage." Internet: http://www.willowgarage.com/blog/2009/06/10/orocos-rtt-and-ros-integrated, June 10, 2009, [July 9, 2010].

[13] “History | The Orocos Project.” Internet: http://orocos.org/orocos/history, [Jun 18, 2010].

[14] P. Soetens, “The Orocos Component Builder’s Manual.” Internet: http://people.mech.kuleuven.be/~orocos/pub/stable/documentation/rtt/current/doc-xml/orocos-components-manual.html, [Jun 18, 2010].

[15] “Bug 670 – TaskBrowser application crashes when using .connect and browsing.” Internet: https://www.fmtc.be/bugzilla/orocos/show_bug.cgi?id=670, Jun 12, 2009, [Jun 25, 2010].

[16] A. Ellsberry, “Development and evaluation of a flexible distributed control architecture,” unpublished draft, M.S. thesis, University of Maryland, College Park, MD, 2010.

[17] “CAN in Automation (CiA): CAN history.” Internet: http://www.can-cia.de/index.php?id=161, [Jun 21, 2010].

[18] “CAN in Automation (CiA) – CAN Product Guide.” Internet: http://www.can-cia.org/pg/can/additional/about_can1.html, [Jun 25, 2010].

[19] “Elmo SimplIQ Servo Drives-Command Reference Manual.” Internet: http://www.elmomc.com/support/manuals/MAN-SIMCR.pdf, [Dec 31, 2009].

[20] J. Craig, “Introduction to Robotics: Mechanics and Control,” 3rd ed., Upper Saddle, New Jersey: 2005.

[21] J.C. Lagarias, et al., “Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions,” *Society for Industrial and Applied Mathematics Journal of Optimization*, vol. 9, no. 1, 1998, pp. 112-147.

[22] S. Roderick and W. Smith, “Ranger Static Performance Measurements,” Space Systems Lab Document DT21-0039, January 2006.

[23] “Point-to-Point and Static Performance Characteristics – Evaluation,” American National Standards Institute, ANSI/RIA R15.05-1-1990 (R1999).

[24] E. Sabelli, D. Akin, and C. Carignan, “Selecting Impedance Parameters for the Ranger 8-DOF Dexterous Space Manipulator,” *AIAA Infotech@Aerospace Conference and Exhibit*, May 2007.

[25] M. W. Spong, S. Hutchinson and M. Vidyasagar, “Path and Trajectory Planning,” in *Robot Modeling and Control*, 1 ed., Hoboken: Wiley, 2006, ch. 5, pp. 163-202.

[26] T. Lozano-Perez, “A Simple Motion-Planning Algorithm for General Robot Manipulators,” *IEEE Journal of Robotics and Automation*, vol. RA-3, no. 3, June, 1987.

[27] G. Marani, et al., “A real-time approach for singularity avoidance in Resolved Motion Rate Control of Robotic Manipulators,” in *Proc. IEEE International Conference on Robotics & Automation*, 2002.

[28] N. Scott, “A line-based obstacle avoidance technique for dexterous manipulator operations,” M.S. thesis, University of Maryland, College Park, MD, 2007.

[29] M. W. Spong, S. Hutchinson and M. Vidyasagar, “Multivariable Control,” in *Robot Modeling and Control*, 1 ed., Hoboken: Wiley, 2006, ch. 8, pp. 289-318.

[30] D. Akin, “Maryland Day - April 24, 2010.” Internet: http://spacecraft.ssl.umd.edu/SSL.photos/SSLevent.photos/2010/100424.MdDay/index.html, [Aug 9, 2010].