August 3, 2001

To the Graduate School:

This dissertation entitled "New Object-Oriented and PC-Based Approaches to Robot Control Software" and written by Markus S. Loffler is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy with a major in Electrical Engineering.

Dr. Darren Dawson, Dissertation Advisor

We have reviewed this dissertation
and recommend its acceptance:

Dr. Nader Jalili

Dr. Ian D. Walker

Dr. Johnson Y. S. Luh

Accepted for the Graduate School:

NEW OBJECT-ORIENTED AND PC-BASED APPROACHES TO ROBOT CONTROL SOFTWARE

A Dissertation

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Electrical Engineering

---

by

Markus Loffler

August 2001

Advisor: Dr. Darren M. Dawson

ABSTRACT

This doctoral dissertation presents three different robot control platforms: The *QRobot* system, the *QMotor Robotic Toolkit*, and the *Robotic Platform*. All three platforms introduce novel software architectures that have the following common objectives: i) reaching a new level of flexibility, ii) establishing a new level of ease-of-use, and iii) achieving a "one-box" solution, which means that the platform is integrated on a single PC, using a single operating system and a single programming language.

This research demonstrates that PC technology is very capable to implement this "one-box" robot control platform. Along with a real-time operating system, the PC allows for the implementation of a homogeneous object-oriented architecture, which fosters ease-of-use and flexibility of a robot control system. While the *QRobot* system and the *QMotor Robotic Toolkit* only focus on certain aspects of the above stated issues, the *Robotic Platform* finally meets all objectives by creating a very slim and optimized object-oriented design, which implements all components in the C++ programming language (including servo control and 3D simulation), utilizing a single PC running the QNX 6 operating system.

VITA

Markus Loffler was born in 1971, in Munich, Germany. He received his diploma in Electrical Engineering from the Technische Universität München (Technical University of Munich). After finishing his dissertation for diploma at Clemson University, South Carolina, he joined the Robotics and Mechatronics Group at Clemson to work on his Ph.D. degree. Besides being involved in various software projects for PC based real-time control and sensor integration, his research focuses on new object-oriented control concepts for robot control platforms.

ACKNOWLEDGEMENTS

First, I would like to thank everybody who motivated me to go abroad and face the challenge of my Ph.D. studies. Especially I'd like to thank my parents, my brother, and my friends Dominik, Hubsi, Wolfi, and Alfons.

I would like to thank my advisor Dr. Darren Dawson and my initial co-advisor Dr. Chris Rahn for their great support of my research efforts, and everybody of the Controls and Robotics Group in Clemson, who made my stay very enjoyable and productive. This includes Nick for long night sessions when fixing my PC, Erkan, for his help as the absolute "Puma expert", and Mr. Hai for his endless cookie supplies. Special thanks to my Robotic Platform team, Vilas, who was essential for the success of this project, Vijay, and Ankur. Also thanks to Jason, Andi, and Bret for fixing the WAM robot 101 times after I crashed it 100 times, Mr. "QMotor manual" Matt Feemster, Matt Steel for his expertise and the great maintanable software he wrote, and also Warren, Sid, Pradeep, Zighao, Aman, Sriram, and Dennis. Additionally, I'd like to thank all the people from QSSL for their great support with regard to their QNX operating system.

I'm very grateful to have been able to work in a wonderful international environment here in Clemson. I'd like to thank all the great people that I've met who are not necessarily work-related, but yet have given me great support and motivation by making my stay in Clemson an interesting learning experience. Especially I'd like to thank my girlfriend Virginia, Jason & Dasha, the Austrians Christian and Gerald, Susanita, Syed, Karen & Matthias, Irenee, and Holgi.

TABLE OF CONTENTS

Table of Contents (Continued)

LIST OF FIGURES

Page

List of Figures (Continued)

List of Figures (Continued)

LIST OF TABLES

Page

CHAPTER 1

INTRODUCTION

The design of software for robot control systems is very demanding and complex since the building blocks of robot control software cover a wide range of disciplines found in robotics and software development. Consequently, it is desirable to create a common generic software platform that can be reused by researchers for different applications. The diversity of robotic research areas along with the complex requirements of hardware and software for robotic systems have always presented a challenge for system developers. Many past robot control platforms were complex, expensive, and not very user friendly. Even though several of the previous platforms were designed to provide an open architecture system, very few of the previous platforms have been reused.

This research focuses on three main goals. The first goal is reaching a new level of flexibility and extensibility. Specifically, the use of object-oriented concepts in robot control software is investigated. The goal is a system that is truly open throughout all components. This research especially focuses on a flexible and extensible design for implementing the servo control loop, which has not been addressed sufficiently in previous research. The requirement of flexibility in the servo control loop leads to the necessity of including control parameter tuning, data logging and data plotting as well as establishing deterministic real-time behavior into the robot control system.

The second goal of this dissertation is establishing a new level of ease-of-use. It is especially investigated how an object-oriented programming interface and an advanced

integration of the graphical user interface (GUI) into the robot control platform lead to an architecture that helps to simplify and speed up the development of robotic applications. Ease-of-use also includes installation and (re-)configuration of the system, debugging, testing, and extending the system for new components.

Finally, this research moves towards achieving a "one-box" solution, which means that the robot control platform is integrated on a single PC, a single operating system and a single programming language. This issue is a design goal but also a requirement, since only a one-box solution allows implementing a simple homogeneous architecture, which allows object-oriented paradigms to completely unfold. This approach is different to many inhomogeneous and distributed robot control platforms introduced by past research. Also, a one-box solution fosters ease-of-use, since users and programmers do not have to familiarize themselves with miscellaneous hardware and software components and their communication requirements (e.g., the configuration of a local network). Due to advances in PC hardware and software in the last ten years, the PC has become a very capable platform to implement a "one-box" robot control platform, which is inexpensive, widely known, and provides a great variety of hardware and software components.

This dissertation describes the path towards the above-described goals. Along this path, three different software packages for robot control have been created. First, chapter two describes the development of the robot control system *QRobot*, which was motivated by the lack of flexibility in implementing new servo control strategies for Puma robots, due to the closed architecture of the Mark II controller. The focus of the *QRobot* system was to demonstrate the feasibility of PC platform for integrating real-time servo control

loops with high-level components (e.g., the trajectory planning and the GUI). However, the design of *QRobot* was limited due to the integration of inhomogeneous software components. To overcome these limitations, chapter three introduces the *QMotor Robotic Toolkit* (*QMotor RTK*), which is based on a purely object-oriented design. Also, the *QMotor RTK* presented the new feature of control tuning and data logging/plotting capabilities, since it was built on top of the QMotor control environment [1]. Yet, the *QMotor RTK* did not include 3D simulation and functionality for kinematics and dynamics of manipulators.

Only recent technological advances finally made it possible to meet all research goals and implement them into an advanced robot control platform: Chapter four presents the design and implementation of the *Robotic Platform*. The "one-box" design goal could not entirely be implemented for the *QRobot* system and the *QMotor RTK*, since the operating system utilized (QNX 4) did not allow for hardware accelerated 3D graphics, as required for the 3D robot simulator. With the introduction of QNX 6 (QNX Real-Time Platform), this last component could be integrated into the homogeneous design. Also, the thread support provided by QNX 6 and advances in compiler technology with regard to templates (which made a real-time matrix library possible) allowed to create the very slim and optimized object-oriented design of the *Robotic Platform*.

CHAPTER 2

TELEROBOTIC DECONTAMINATION AND DECOMISSIONING WITH QROBOT, A PC-BASED ROBOT CONTROL SYSTEM

Introduction

The U.S. Department of Energy (DOE) is facing the decontamination and decommissioning of a high number of surplus facilities. These facilities often contain radioactive or other hazardous material. Current technologies are often labor intensive, time consuming, expensive, or they unnecessarily expose workers to the hazardous material. The DOE is looking for new and innovative technologies that allow D&D operations to be faster, safer, and more cost-effective. Telerobotic systems provide a good solution to this problem. They allow robots to be remotely controlled from an operator console and provide visual feedback to the operator. In basic systems, an operator controls the robot directly (*e.g.* with a joystick) and receives video feedback [2]. Performing a remote disassembly is a complicated, often repetitive task, which requires skilled operators. Therefore, much of the ongoing research focuses increasingly on the development of semi-autonomous systems. These systems perform higher level tasks, such as removing a bolt, triggered by the operator. Furthermore, virtual reality (VR) based operator interfaces are desired to simplify interaction with the system.

This chapter describes how the QRobot joint level control [3] was extended to a complete semi-autonomous robot control system for D&D operations. QRobot is the

robotic part of the "Semi-Autonomous Decommissioning of Hazardous Waste Project" [4]. It is a purely PC based system that integrates the following components:

- A joint level control for Puma manipulators.
- A trajectory generator with a high level programming interface.
- A 3D OpenGL-based hardware-accelerated robot simulator.
- A video based and a VR based operator interface.
- Teleobservation programs.
- Interfacing of different sensors.
- Control of different robotic end-effectors.

The chapter concludes with a demonstration of the system's capability by the experimental study of the disassembly of an electric motor.

Motivation for Developing a PC Based System

The different components of an advanced semi-autonomous telerobotic system have different hardware and software requirements:

- The joint level control task and the trajectory generation of the robots require hard real-time performance.
- The graphical user interface needs to integrate VR and video techniques. Hardware accelerated 3D video is required for the VR interface.
- Networking capabilities are required in order to locate the robot control hardware remotely from the operator console. Networking is also required if a heterogeneous multiprocessor architecture is used to divide the work among multiple computers (***e.g.*** video capture on one PC, robot control on another PC, *etc.*)

The above requirements usually lead to the integration of proprietary solutions and expensive hardware platforms. As an example, consider an RCCL based system. The software consists of a robot control library (RCCL [5]) running on a Sun workstation, Moper (a replacement for VAL, running on the LSI 11/2 processor in the Mark II), and firmware joint level control running on digital servo boards inside the Mark II [6]. Additional hardware required includes an SBUS to VME bus adapter and a VME card cage (with various parallel I/O and timer cards. The closed architecture of the Mark II controller prevents the implementation of new, state-of-the-art control algorithms [3], as well as the integration of sensors such as cameras into the control loop. The heterogeneous architecture of this type of system leads to a higher complexity of integration and higher costs.

QRobot is entirely PC-based. The entire computational functionality of the system, including the joint level control, is implemented exclusively as PC software. Neither a dual processor architecture (like PC/digital signal processing boards solutions) nor special controller hardware (such as the Mark II's digital servo boards) is necessary.

This system has the following advantages:

- The system is cost-effective, because PCs and their components are less expensive than proprietary controllers or traditional Unix workstations.
- The system has a simpler architecture, since the additional effort to integrate completely different hardware components (such as VME cards with an SBUS computer) is not required.
- The system is more flexible. To modify or extend the system, only a change to PC source code is necessary.

- The PC is a widely known and technically advancing technology. Many powerful software packages as well as a great variety of interface boards are available for the PC.

There are two developments that allowed the integration of the PC with the different hardware and software requirements stated above. First, the advent of high-speed PC CPUs provides computing power similar to or exceeding Unix workstations or special purpose computers such as DSP boards [7]. Second, hard real-time operating systems, available for PCs, are able to execute real-time tasks (such as joint level control and trajectory generation) as well as non real-time tasks (such as networking and user interface tasks) on one PC [8].

Overview of the Disassembly System

System Components

Figure 1 and Figure 2 show the software and hardware components that are distributed across three PCs. The *VR Operator Interface PC* runs Windows NT, while the *Main Control PC* and the *WebCam PC* run QNX, a real-time operating system. The *VR Operator Interface* and the *Robot Simulator* are integrated into one Windows NT program. This program displays a kinematic three-dimensional simulation of the robot and allows the operator to start disassembly operations by clicking in the 3D scene. The *Video Operator Interface* provides similar functionality, but uses video images as the operator feedback. It contains the actual disassembly program. The communication with the virtual operator interface (to trigger disassembly operations) uses Internet Domain TCP/IP sockets.

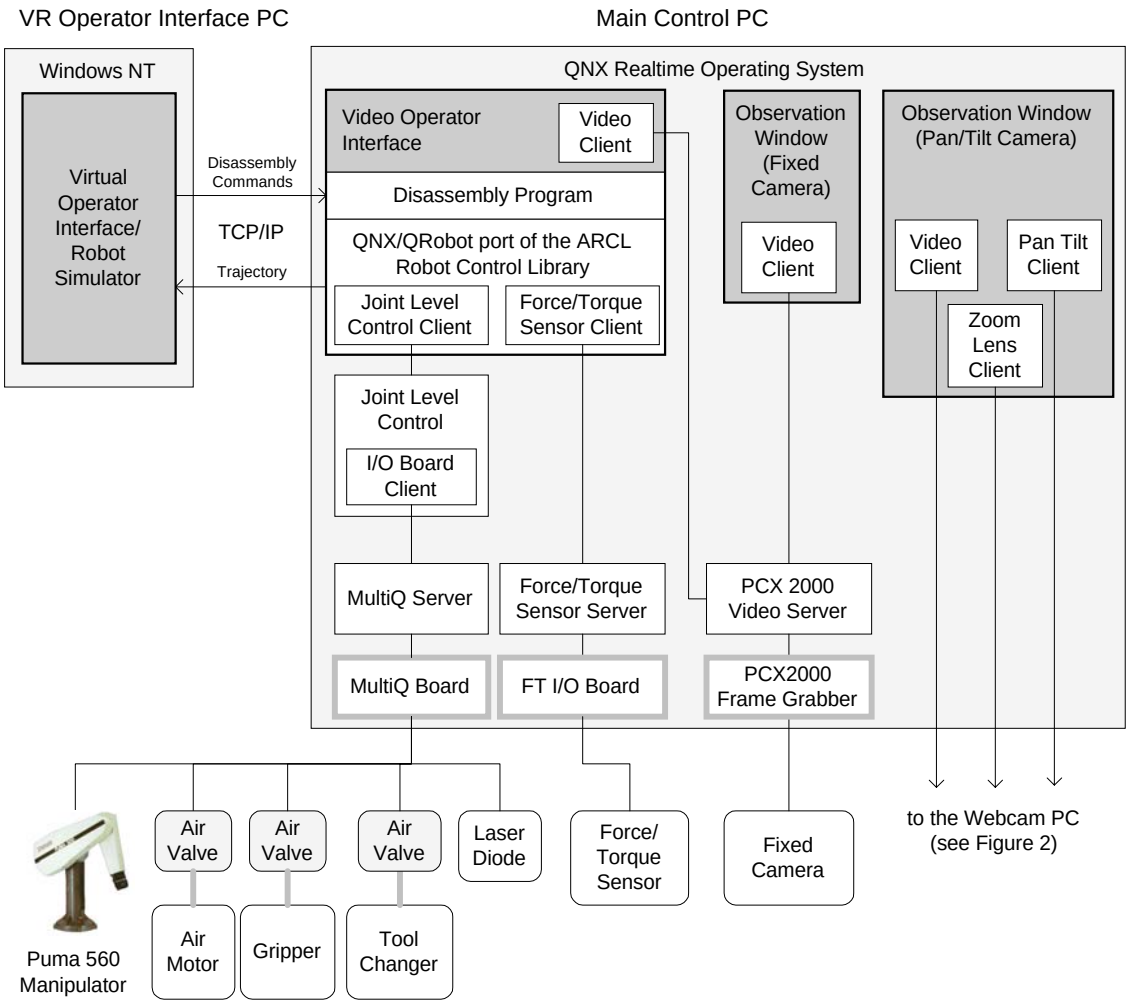


Figure 1. The Virtual Operator Interface PC and the Robot Control PC

The *Disassembly Program* issues high level commands to *ARCL*, a robot control library that serves as the programming interface and as the trajectory generator. ARCL generates a stream of setpoints that are fed into the *Joint Level Control*. *Observation Windows* provide visual feedback of the D&D operation. They show a continuously updated image from one of the video cameras. Multiple observation windows can be started and connected to different video cameras.

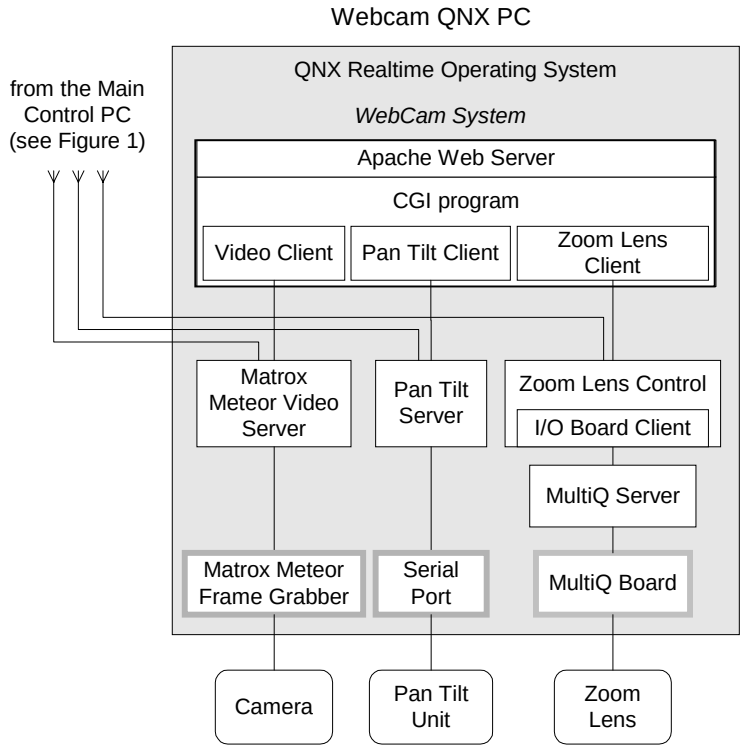


Figure 2. The WebCam PC

The *WebCam System* allows video feedback over the World Wide Web by using a standard web browser. The camera, the pan-tilt unit and the zoom lens of the WebCam system is also accessible from the observation windows running on the Main Control PC. Special programs, called *Servers*, are responsible for accessing the PC boards and for implementing control algorithms. Figure 2 shows some examples of servers: the pan tilt server, the zoom lens control, and the MultiQ server. Applications use *Clients* to communicate with servers.

Different PC boards are used in the system: Quanser's MultiQ board and ServoToGo's S8 board for digital and analog I/O, the PXC2000 board and the Matrox Meteor board for video capturing, and a custom board for interfacing the force/torque sensor. The main hardware component is a *Puma 560 Manipulator*. A hardware retrofit interfaces the power amplifiers, encoders and potentiometers of the Puma directly to an I/O board. The

Force/Torque Sensor and the *Toolchanger* are mounted on the Puma's wrist. A tool rack provides three tools for the disassembly: A *Gripper*, an *Air Motor* for bolt removal, and a *Laser Diode* to simulate torch cuts. The vision system consists of two cameras. One is fixed and connected to the Main Control PC (Figure 1), the other is mounted on a pan/tilt unit and uses a zoom lens. This second camera is connected to the WebCam PC (Figure 2).

The Multitasking and Communication Architecture

The systems functionality is split into many cooperating tasks. For these tasks to work seamlessly together, the operating system must fulfill certain requirements. It must provide priority based deterministic CPU scheduling to ensure that high priority real-time tasks (*e.g.* the joint level control) are not delayed by low priority non real-time tasks (*e.g.* graphical user interfaces). It must provide robust interprocess communication (IPC) mechanisms so that the cooperating tasks can synchronize and communicate.

The real-time microkernel based operating system QNX, developed by QSSL [9], meets all of these requirements. Unlike real-time extensions such as RT-Linux or Hyperkernel for Windows NT, QNX is a true microkernel real-time operating system. One benefit of this is that the whole spectrum of operating system functions, including file access and networking, can be used in real-time tasks.

Client/Server Architecture. The system utilizes two types of servers. Hardware servers are used to access hardware. Control servers implement a control algorithm. Both types of servers are separate programs that usually cycle at a fixed rate. To exchange data with a server (*e.g.* to send setpoints to a control or to read analog inputs), a program is linked

with the appropriate client library. The client library uses shared memory or message passing to communicate with the server.

Message passing is a QNX IPC mechanism that is very flexible because it is network transparent. Network transparency allows sending messages between clients and servers that are located on the same PC or on different PCs. Reconfiguring the location of clients and servers does not require recompilation of code. For example, the video client can connect to the video server of the fixed camera, running on the same PC, or to the video server of the camera mounted on the pan/tilt unit, running on the WebCam PC. This mechanism allows great flexibility in distributing the resources of the system.

Another advantage of the client/server concept is that multiple clients can use the same server. In this way, resources can be accessed from multiple tasks. For example, different clients can use the images captured by the same video server: the operator interface, the observation windows and the webcam. Since QNX message passing provides synchronization implicitly, collisions of multiple requests from clients are automatically avoided since client requests are automatically serialized.

Finally, the interface between client and server adds a level of abstraction to the hardware interfacing. To use different standard I/O boards, for example, different servers are implemented, but the same generic client can be used. The communication protocol between client and server doesn't change. To use a different board, a different server must be started, but no client code needs to be changed or recompiled.

Timing. The timer server is a special server. It provides the clock for all components, which guarantees synchronous behavior of the different tasks. Timer clients and timer

servers share a sequence counter in a shared memory segment that is used to detect a lagging client.

Priority Based Deterministic Multitasking. QNX allows running different processes at different priorities. In the architecture of this system timer servers, which create the system clock and trigger all actions, run at highest priority (see Table 1). Below that priority are hardware servers and control programs. All other tasks have lower priorities.

This scheduling discipline guarantees certain safety features:

- Varying system load will never delay the timer server. Processes falling behind the timer are always detected.
- Hardware servers and control programs are never delayed by the user interfaces or other non-real time system processes. This feature allows having the real-time control and the user interfaces to be run on the same machine.
- A user interface does not crash the execution of the control.

Priority	Process
29	Timer Server
28	Hardware Servers
27	Control Programs
26	QNX CPU Scheduler
25 - 20	Root owned or setuid processes can request to run at these priorities. Many device drivers run at these priorities.
<= 19	Non root processes can run at these priorities
10	Default process priority

Table 1. Task Priorities

Hardware Components

The Puma 560 Retrofit

The standard controller for Puma manipulators is the VAL-II based Unimation Mark II. The VAL software runs on a DEC LSI-11/2 computer. It provides trajectory generation and a simple, text based command interface over the serial port. Six digital servo boards, containing 6503 microprocessors, perform the joint level control [6, 10]. RCCL systems replace the functionality of the obsolete VAL high level control, but the joint level control is still done by the digital servo boards. Performing the control calculation by PC software instead of the digital servo boards allows the implementation of arbitrary user-defined joint level control and the integration of sensor information into the joint level control. To achieve this, a retrofit of the Puma hardware is necessary.

The hardware retrofit bypasses the LSI-11/2 trajectory generation and the digital servo boards and interfaces the amplifiers, encoders and potentiometers of the Puma 560 to a PC I/O board. The TRC boards and card cable sets, produced by Trident Robotics and Research Inc. [11], are specifically developed for this purpose. The TRC boards provide an almost “plug and play” solution [3]. The TRC setup consists of three parts: The TRC006 is a simple ISA bus interface card for the PC. The TRC004 board contains the actual A/D converters, encoders and digital lines. Finally, the TRC041 cable card set removes the need for point-to-point soldering on the Mark II backplane. Using the TRC boards provides a simple solution, but it is still a proprietary solution. That is, the user is dependent on one source of hardware and software – Trident Robotics Research. The TRC boards were specifically designed for use with Puma robots, and so they are not

adaptable for use with other control applications. For example, the A/D channels are very slow, and could not be used in a 1KHz control loop.

In order to make the system more flexible and less dependant on one vendor's hardware, the next step was to replace the TRC boards with generic interface boards. The MultiQ board, produced by Quanser Consulting [12], was selected to replace the TRC boards. It was necessary to develop an additional simple interface board to connect the MultiQ board to the amplifier circuits of the Puma. It contains preamplifiers and filtering circuitry for the noisy potentiometer readings. The TRC041 cable card set was initially used with the MultiQ board solution, but it was later replaced by an in-house developed cable card set, in order that the system might be completely vendor independent.

The MultiQ boards do not support latching of digital inputs, a feature that the TRC004 board does provide for use with the Puma 560 encoder index pulses, required during the robot calibration procedure. To solve this problem, the MultiQ hardware server simulates the latching in software.

To demonstrate that the architecture is flexible enough to easily accommodate other motion control interface boards, another controller was retrofitted with a ServoToGo (STG) S8 interface board instead of a MultiQ board. The interface board for the STG board only differs in the wiring from the one for the MultiQ board. A problem occurred when using the encoders: The encoder channels of the MultiQ board operate in the reverse direction than those of the STG board. To compensate, the encoder values are reversed in the STG hardware server.

End-Effector Hardware

The D&D tasks require various tools and sensors. To accommodate these tasks, an ATI Industrial Automation Gamma 30/100 Force/Torque (FT) sensor is mounted at the end of the robot arm. Although it could be used for force-based control and compliance trajectory generation, currently it is only utilized to trigger an emergency stop when the end-effector encounters forces above a certain threshold. The FT sensor comes with an ISA bus controller.

The toolchanger is mounted on the FT sensor. It is a Light 5 Robotic Tool Changer, also from ATI. It contains 10 electric and 6 pneumatic pass-through ports, for electrical and pneumatic connections on the end-effector. The custom-built tool rack provides space for three tools: For bolt removal, the MMR-002 air motor, from Micro-Motor Inc., is used. The gator grip socket tool, mounted on the axis of the air motor, is a universal socket that automatically adjusts to varying bolt sizes. The gator grip's design allows bolt removal operations to proceed even with several millimeters of positioning error. The second tool is a standard pneumatic gripper. Finally, the laser diode is used to simulate the operation of a torch cut. Electrically controlled air valves actuate the tool changer, the gripper and the air motor. Digital output lines of the MultiQ board control all tools.

Observation System

The observation system consists of two Pulnix TMC-7 cameras. One is connected to a Matrox Meteor PCI bus frame grabber, the other uses an Imagination PXC200 PCI bus frame grabber. One camera is mounted in a fixed direction above the workspace. The other camera is mounted on a Directed Perception Pan/Tilt Unit (PTU) model PTU-46-

17.5. The PTU is connected to the PTU controller (a micro controller based constant acceleration open loop control), which is in turn connected to the PC via an RS232 serial port. The PTU mounted camera also uses a zoom lens. The motors and potentiometers of the zoom lens are connected to a custom interface board (containing amplifiers and voltage dividers), which is then connected to a MultiQ for A/D and D/A.

Software Components

Clients and Servers

All clients and servers are written in C++. Besides the joint level control, which will be described later, and the zoom lens control, all servers access hardware directly. The MultiQ server, for the MultiQ board, and the STG server, for the STG S8 board, perform digital and analog I/O at a fixed frequency. A generic I/O board client is used to communicate with either of these servers. The Matrox Meteor Server and the Imagination PXC Server are responsible for capturing video frames. A generic video client is responsible to send image requests to these servers and to receive the image data. As message passing is used for communication between client and server, the client can be located on a different PC than the server. The force/torque sensor server talks to the ISA force/torque sensor controller board. It polls the force and torque values continuously and provides them to the force/torque client in a shared memory space. The pan/tilt unit server controls the pan/tilt unit over the RS232 serial port. It receives messages from the pan/tilt client that contain the desired angles and issues move commands over the serial port to the controller. The zoom lens server receives the desired zoom factor from the zoom lens client and uses a proportional position control to set the focal length of the

lens. The gains of the control are set very high and the output is clipped, so it is almost a bang-bang approach for fast response.

Overview of The Robot Control Software

The joint level control, the ARCL trajectory generator, and the task level program represent three levels of robot control. Figure 3 shows these levels and their specific implementation. All three levels are implemented on the same PC, which leads to a simpler architecture. Furthermore, sensor information can be used on all levels.

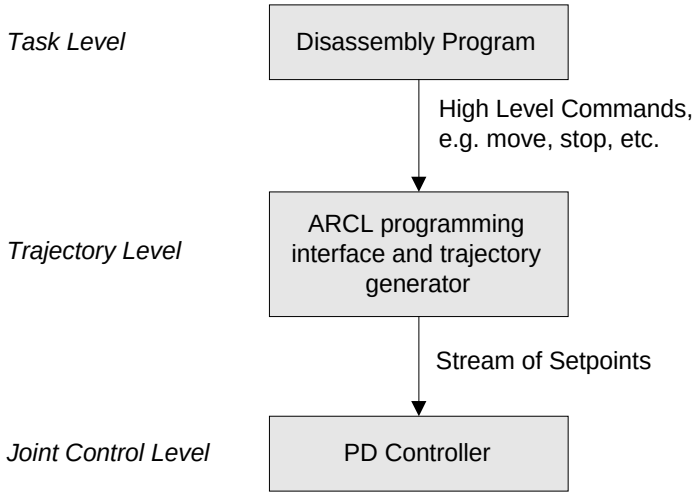


Figure 3. Levels of Robot Control

Joint Level Control

The Puma 560 retrofit allows the implementation of the joint level control program as PC software. This is a very flexible solution, since the control algorithm can be modified directly by changing and recompiling the control program. The ever-increasing computing power of PCs allows the implementation of more complex control algorithms.

In addition, it is now possible to include sensor information in the joint level control loop, which allows the implementation force-based control or even visual servoing.

The joint level control used in the D&D system was developed using QMotor [7], which is an environment for PC based control program development and implementation. The control program implements a PD controller with static and coulomb friction compensation for all joints and gravity compensation for the second and third joint. Joint velocities are manufactured via a backward difference method and low pass filtered [3]. The joint level control program works as a server and receives the stream of setpoints via message passing from the joint level control client, which is part of ARCL. The rate of the setpoints can be lower than the control frequency, because the trajectory is low-pass filtered in the server. The control can be switched to a zero gravity mode. In this mode, the control just compensates for the gravity on the robot links instead of servoing to desired setpoints [13]. The robot can be freely moved around in this mode, which is used to teach end effector positions and orientations with the teachpendant program.

Trajectory Generation and Robot Programming Interface

To achieve the goal of an entirely PC-based system, a high-level robot control API and trajectory generator package for the PC is required. As there is no such package available for QNX, the quickest solution is a port from a different platform. One of the most sophisticated and well-known high level robot control libraries is RCCL. John Lloyd, one of the developers of RCCL, had unsuccessfully tried to port RCCL to QNX. For this reason, RCCL was not considered for use in this project.

The Advanced Robot Control Library (ARCL), a robot control library developed by Peter Corke at CSIRO [14], seemed to be more suitable for a QNX port, because its modular architecture separates platform dependent and platform independent parts. ARCL was developed for Unix workstations. Porting ARCL to QNX and integrating it with the QRobot system required significant effort. This effort included making the C source code C++ syntax compliant, debugging the existing ARCL source code, developing the platform specific part of ARCL, and embedding the ARCL modules into the real-time environment of QNX and the client/server architecture.

ARCL provides similar functionality to RCCL, although in a much more limited fashion. The main problem with ARCL was that some functions were either missing, were not implemented completely or contained bugs. The following changes were made to ensure proper operation:

- A problem with the specification of joint coordinates as target positions was fixed.
- A bug in the stop command was fixed.
- The disabling of the arm power in case of inverse kinematic errors was added as a safety feature.

The main challenge, however, was writing the AMI (ARCL machine interface), for QNX. The AMI is the platform dependent module that contains functions for multitasking, timing and hardware interfacing of the manipulator. Two problems occurred when developing the AMI for QNX. First, the architecture of the AMI requires that both the trajectory generator task and the user program task share the same address space. This can be either implemented by using shared memory between these tasks or by using threads. Since ARCL was not designed to take advantage of shared memory, the

development of a special memory manager would have been the only solution. Threads, the other approach, are multiple instances of a process that share the same address space. QNX supports threads, but in a limited fashion, and not all library functions are "thread-safe". Tests showed that the thread-based solution, which is much easier to implement than the shared memory manager, was adequate.

The other main problem concerned the management of semaphores, a technique of process synchronization. While ARCL assumes semaphores with just two states (blocked and non-blocked), QNX semaphores actually utilize a counter to allow multiple processes to wait on the same semaphore (which is the normal mechanism). Some small changes in ARCL were necessary to ensure proper operation. In addition, ARCL does not destroy the semaphores at program termination. QNX semaphores are also not automatically destroyed at program termination, so they accumulate until no more semaphores are available. To solve this problem, a semaphore manager was added to the AMI to keep track of the semaphores in use and to destroy them as the program terminates.

To integrate ARCL into the QRobot system, the QNX/QRobot AMI was constructed to contain the following functions:

- A client to the joint level control server is used to send the trajectory to the joint level control, which acts as a server.
- Functionality was added to connect to the robot simulator running on the Windows NT PC. Sockets were used as the communication mechanism. A 100Mbps full duplex point-to-point fast Ethernet link proved to be fast enough to ensure that the robot simulator does not fall behind the trajectory generation. The protocol is simple: The data sent contains the stream of setpoints generated by the trajectory generator. There

are two modes of operation, test mode and standard mode. Test mode allows running robot control programs with the simulator alone, without accessing hardware. This is useful for debugging robot programs without the risk of damaging the robot. In standard mode, the trajectory is sent to both the robot simulator and to the joint level control, so it is possible to compare the movement of the real robot with the 3D simulation. A communication channel in the other direction, from the robot simulator to ARCL, was added to allow the virtual operator interface (which is part of the robot simulator) to send disassembly commands to the task level program.

- Force and torque information is used to trigger an emergency stop to minimize damage from collisions between the end-effector and the workspace.

Rather than using ARCL's limited functionality to control tools (there is only a function to control the gripper), a C++ class was developed for each tool. The tool classes use I/O board clients to control the tools.

Although the port of ARCL achieved satisfactory results, it is not an ideal solution. The use of threads is not 100% safe, since not all QNX library functions are "thread-safe" although we never identified any crashes that could be attributed to threads. If a user chooses to use a non thread-safe function in one of his programs, the behavior of the system is undefined, and hence could crash the user's program.

Robotic Utility Programs

Besides the specific D&D programs, a set of utility programs was developed. These programs were inspired by their RCCL counterparts.

PotVal: The PotVal utility performs the initial calibration procedure of the Puma 560 that relates joint potentiometer readings to encoder readings.

PumaCal: The PumaCal utility performs encoder calibration after power up of the manipulator. It determines the current position of the robot by using potentiometers and index pulses. This utility is similar to RCCL's pumacal utility [5], but performs the calibration in half the time.

Teachpendant: To "learn" the end-effector positions and orientations used in the D&D operation, a teachpendant program was developed, see Figure 4.

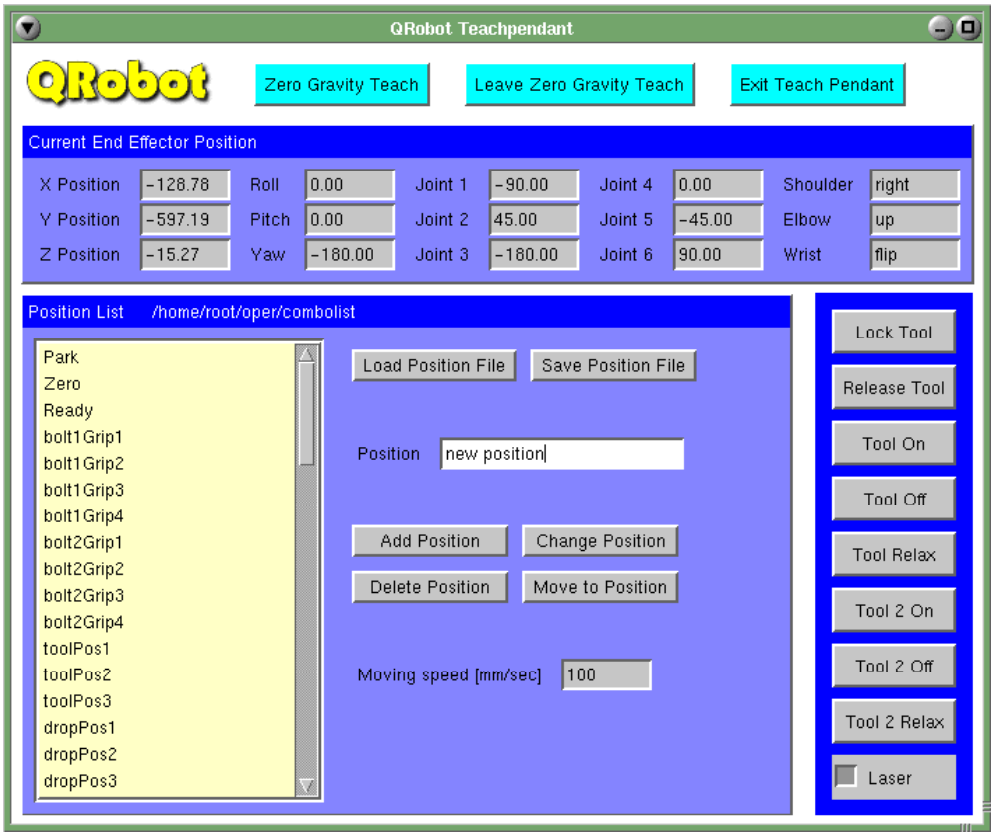


Figure 4. The QRobot Teachpendant

The teachpendant uses the zero-gravity mode of the joint level controller, which allows the user to easily push the robot around in the workspace. At the top of the window, the current position is continuously displayed in Cartesian and joint coordinates. Once a position and orientation is found, it can be stored under a given name in a position list. It is possible to leave the teach mode at any time and move the robot to previously taught positions. The position list can be stored in an ASCII file for later use in the teachpendant, or for use from an ARCL program.

The Disassembly Program

A motor is used to demonstrate a simple disassembly. The objective is to remove the cap of the motor. Figure 5 shows the steps performed by the system:

1. Remove the first bolt: The gator grip is used to unscrew the bolt. As it usually stays in the housing, the operator has the additional option to remove the bolt with the gripper and drop it into a container.
2. Remove the second bolt in the same fashion as described in 1.
3. Perform a torch cut. In the experiment, this is simulated by a laser diode.
4. Remove the cap with the gripper and put it on the table.

The disassembly program is written in C++. For each disassembly step, via points are determined with the teachpendant program and saved to a file. The disassembly program reads this file and creates transformations and position equations for each via point. The position equations are the input to the ARCL “move” function calls. Each disassembly step consists of picking up the right tool from the tool rack, performing the operation, and returning the tool back to the rack. Some special functions are defined to allow the operator to intervene in case the system failed to complete a step. These functions include

manually getting or returning a tool and manually locking or releasing a tool. A callback function of the trajectory generator is used to update a progress bar, which informs the operator how much of the task is complete.

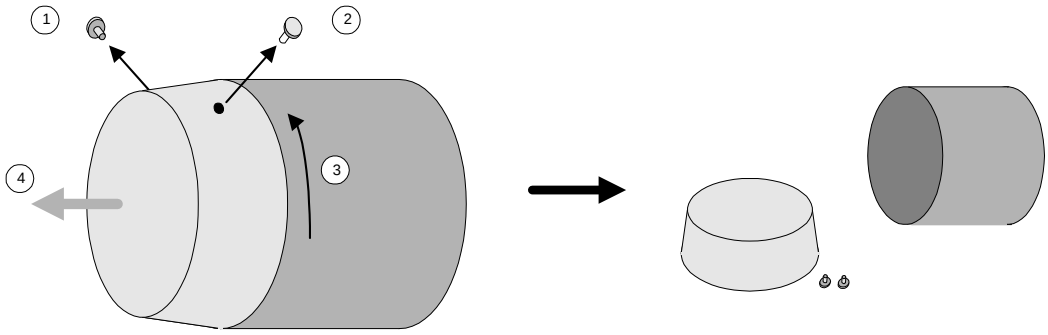


Figure 5. Steps to Disassemble the Motor

The Operator Programs

Four operator interface programs offer different control and feedback functions. Observation windows and the video operator interface run on the same PC as the robot control, but at a lower priority. They use Photon, the graphical user interface for QNX. Photon provides functionality similar to the X Window System and Xt. To accelerate GUI development under Photon, a C++ class library (QWidgets++) was developed.

Observation Windows. The observation window (see Figure 6) provides live visual feedback from a video camera. When using the camera mounted on the PTU, the observation window offers additional functionality: Clicking into the image defines the new center and moves the PTU accordingly. The buttons at the bottom control the zoom lens of the camera.

It is possible to start multiple observation windows and connect them to the same or different cameras. As the observation windows use message passing based client/server communication, the camera servers can be distributed over multiple PCs. The disadvantage of message passing is reduced speed in the image transfer. Depending on the PC performance, image size, color depth, and network traffic, the observation window displays 1-5 frames per second.

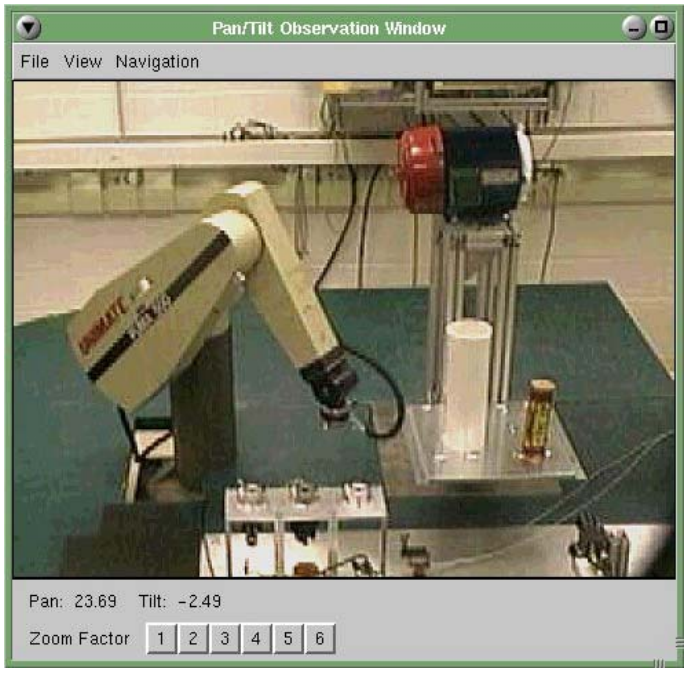


Figure 6. Observation Window

Web Camera. The WebCam is a World Wide Web based visual feedback, with similar functionality to the observation windows. The Apache web server starts a CGI program whenever the web page is accessed. The request for the web page contains the desired camera position and the desired zoom factor as parameters. The CGI program moves the PTU to the desired position, sets the zoom factor, and captures an image. This image is converted to JPEG format, and a web page is dynamically created to show the image. The

client/server architecture allows multiple observation windows and any number of web browsers to request images at the same time. The requests are serialized and queued in FIFO order. The advantage of the WebCam is the accessibility from any Internet connected computer. The disadvantage is the lack of a continuous and fast update of the image.

Video Operator Interface. The video operator interface provides even unskilled operators an easy to use interface to control the disassembly tasks. The idea is to use the video image to select disassembly operations. The operator moves the mouse cursor over a certain part that he wants to disassemble. The operator interface then displays a pop-up menu with a list of disassembly options. For example, when the operator moves the mouse over the motor end cap, the end cap is highlighted, and a menu pops up with the menu item "Remove Cap" (see Figure 7). After the operator selected an operation, the program begins to perform the task and shows progress information in a dialog. As the disassembly is being performed, the operator is able to supervise the operation in the observation windows. In case the disassembly of a part was unsuccessful, the operation can be repeated. The operator intervention menu offers a set of intervention functions to directly pick up or return a tool or to directly control the toolchanger.

The image-based selection of disassembly operations is convenient for the operator, but it also requires that the system knows where the parts of the object are located in the image. The Image Processing group at Clemson University is investigating the use of advanced image processing and 3D-object virtualization to automatically identify and locate these parts for the disassembly task [4]. This research is not addressed in this

dissertation. To demonstrate the basic concept of the operator interface, the coordinates are manually determined in the current system.

Virtual Operator Interface/Robot Simulator. The video operator interface works fine with a workpiece such as the motor and the overhead camera. However, using different camera perspectives or more complex workpieces can result in hidden parts that can not be viewed and selected by the operator. For instance, the front perspective of the motor would not show the second bolt at the back of the motor. Virtual Reality based operator interfaces overcome this problem. In those interfaces, the operator is able to navigate within the virtual scene and view parts from different angles. The virtual operator interface is integrated into the robot simulator.

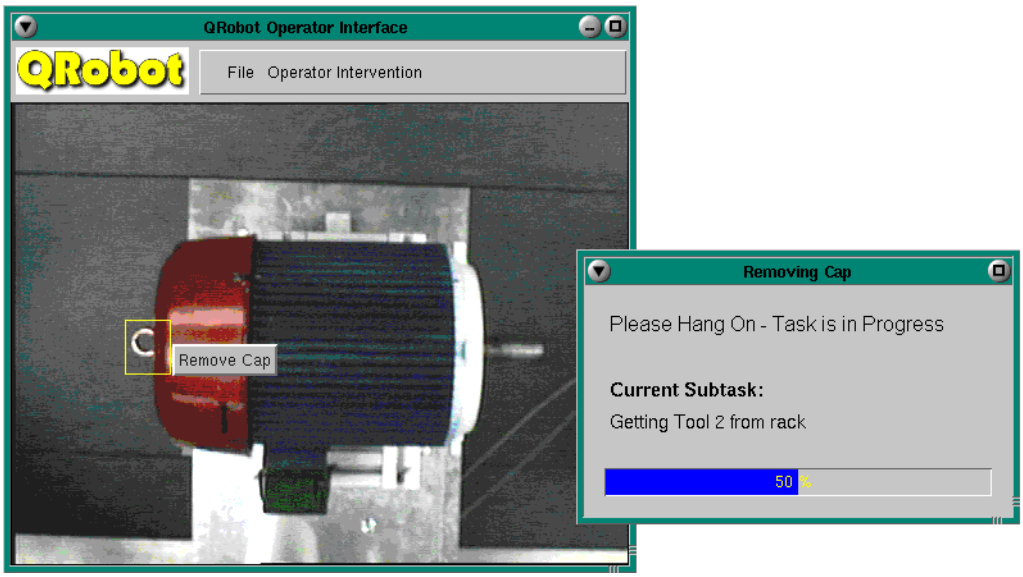


Figure 7. Video Operator Interface

Figure 8 shows a screenshot of the robot simulator. The simulator allows testing robot control programs without running the actual hardware and risking damage in case of programming errors. Since there are no hardware accelerated 3D graphics libraries

available for QNX, the program runs under Windows NT and uses OpenGL, a standard 3D graphics library [15].

The 3D scene consists of two Puma robots, the toolrack and the workpiece. The main window is split into three parts that show the scene from different perspectives. In each sub-window, the operator can navigate by using the mouse, selecting and defining custom positions or selecting the end-effector view. The latter option simulates the view of a camera mounted on the end-effector. The level of detail in the display can be reduced to accelerate the display.

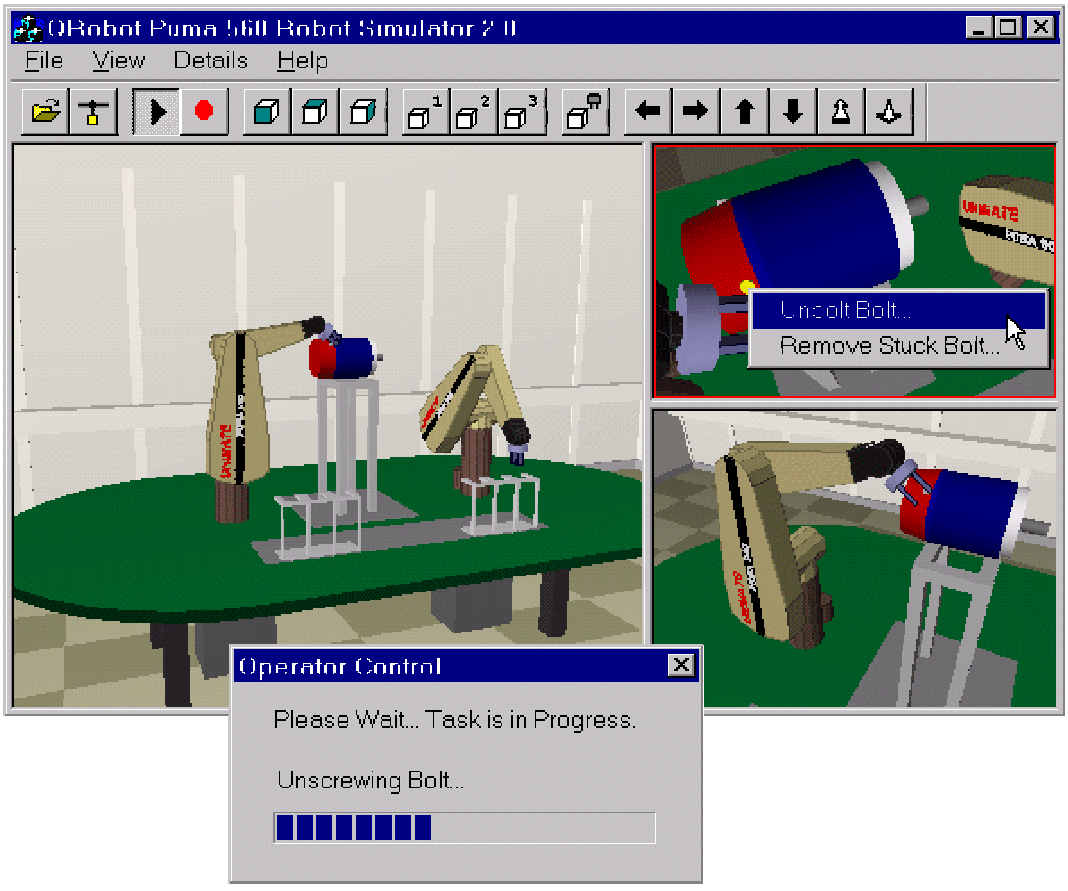


Figure 8. The Virtual Operator Interface/Robot Simulator

While the disassembly program is running, ARCL forwards the trajectory information to the robot simulator. The VR operator interface is part of the robot simulator. It requires that the video based operator interface be running, since the video based operator interface is linked to the actual disassembly program. The convenient side effect is that both operator interfaces work hand-in-hand: A disassembly task can be started from either.

A special technique of 3D programming, called object picking, gives the operator functionality similar to the video based operator interface. Moving the mouse cursor over parts of the motor highlights these parts. Clicking on the parts displays menus with disassembly options. Once the operator selects a disassembly option, the software encodes this operation into a command word, and sends it to the video based operator interface, which initiates the operation. This data transport uses the same TCP/IP connection that is used to send the trajectory information. Progress reports are also sent back to update the progress dialog box as shown in Figure 8.

The disadvantage of the current version of the robot simulator/operator interface is that the scene is hard-coded by function calls within the robot simulator. Changing the scene is not trivial and requires extensive programming effort. However, this compromise establishes a working system at this stage of the project. To address this problem, ongoing research of the third group involved in this project, the Virtual Reality Group at Clemson University, is investigating automatic virtualization of the D&D workspace.

Experimental Results

There are three aspects of the experiment. The first is the usability and reliability of the system used by an untrained operator. The experiment showed that the video based operator interface and the VR operator interface provide an intuitive way to control the D&D operation. Only a few mouse clicks are necessary to guide the complete disassembly. Problems occurred when the operator needed to recover from a system failure or a handling error. Generally, the capabilities of aborting operations and returning to the initial state are limited. Often, the robot, the tools or workpieces had to be moved back manually to initial positions.

The second aspect is the reliability of the control system and the mechanical part. The system was successful in repeating the disassembly multiple times without any problems. The joint level control was capable of precisely moving different tools of different weights. Figure 9 shows the desired position trajectories for link 2 and 3 for the bolt removal task in the top graphs. The graphs below are the tracking errors. The middle graphs show the tracking errors for a test run without the bolt actually being present. The actual bolt removal happens from t = 20 sec to t = 30 sec. Comparing the tracking errors in the middle and bottom graphs shows that the effect of the force created from removing the bolt does not effect the control significantly.

Finally, the most interesting part is the stability of the user interface programs and the real-time control programs running together. The system showed high stability in this issue. For example, it is possible to open many observation windows while the D&D task is in progress. The observation windows slowed the user interfaces down, but they did not influence the control tasks and the disassembly task.

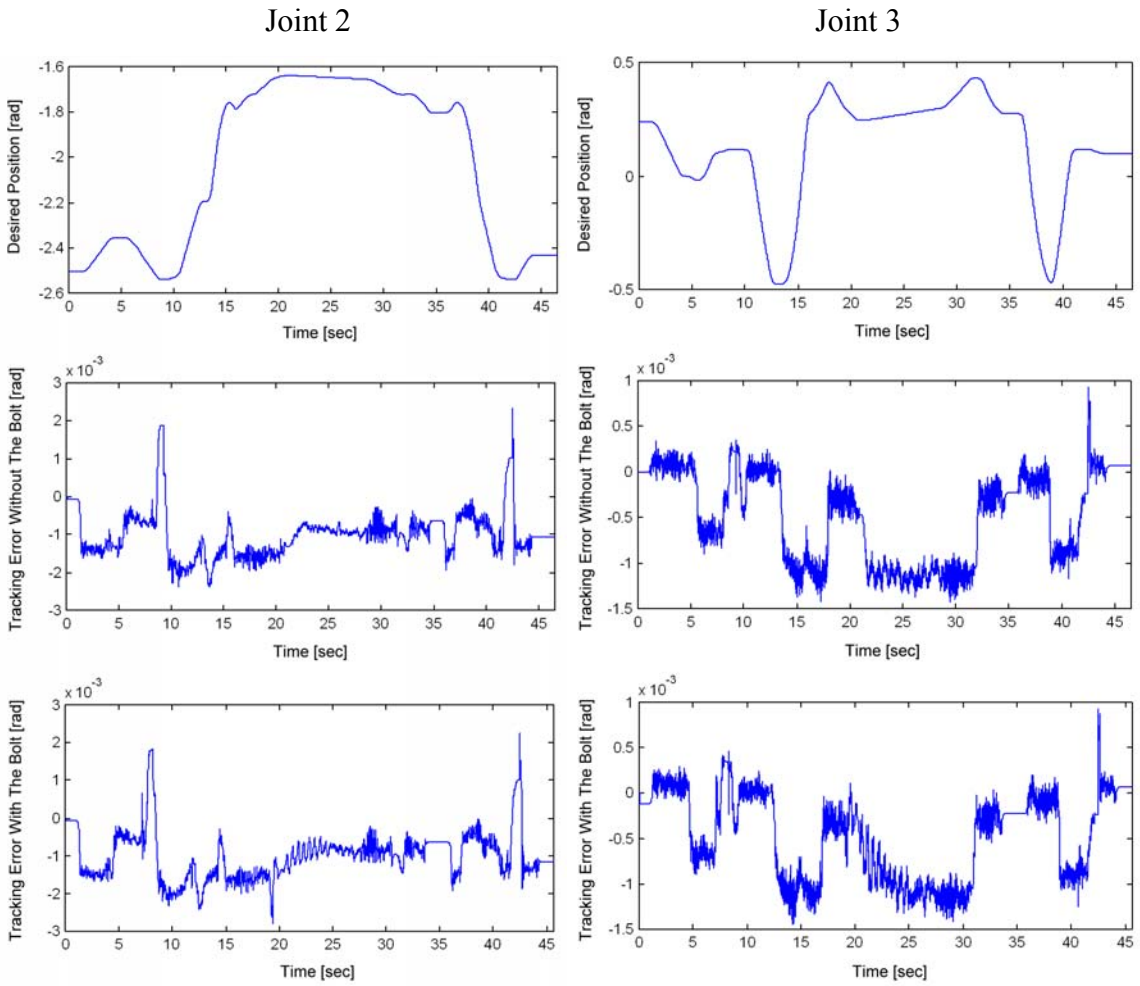


Figure 9. Trajectories and Tracking Errors for the Bolt Removal

Conclusions

The QRobot system described in this chapter demonstrates the feasibility of using a PC for the various tasks required in telerobotic semiautonomous D&D operations. It was shown how conceptually different tasks can be integrated on a single cost-effective hardware platform. The client/server concept, using a modern real-time operating system as its platform, provides a flexible way of communication for these tasks. Priority based scheduling allows complex real-time control programs to coexist with low-priority

graphical user interfaces. The operator interfaces and the teleobservation programs provide an easy-to-use telerobotic operator environment.

The limitations of the system originate in its components and its architecture. ARCL as a high-level control library is not a satisfactory, robust solution. The port of ARCL to QNX uses threads, which are not completely supported by the operating system. ARCL itself is limited in functionality and flexibility. Although QRobot is entirely PC based, it still uses two operating systems: QNX (for all functions except the 3D robot simulator) and Windows NT (for the 3D robot simulator). This introduces higher complexity and costs. It would be desirable to have all components running under the same operating system. Additionally, the system was developed by integrating different components. All components have been specifically modified to work together, which leads to a static architecture. For example, extending the system to a different robot type would result in modifications of ARCL, the joint level control, the robot simulator and the communication between them, which is a big disadvantage. The biggest problem is the limited flexibility in adapting the system to different applications. Basically, all parts of QRobot are developed specifically for the D&D example of the motor disassembly. Adapting the system to a different application requires extensive knowledge of its internal workings and modification of many parts, such as the operator interfaces, the robot simulator and the disassembly program.

CHAPTER 3

OBJECT-ORIENTED TECHNIQUES IN ROBOT MANIPULATOR CONTROL SOFTWARE DEVELOPMENT

Introduction

Challenges and Needs in Robot Manipulator Control Software Development

Software development for the control of robotic manipulators is a complex task. That is, even for simple pick and place operations, the software developer needs to incorporate hardware interfacing, concurrency, real-time programming, servo control programming, and trajectory generation into the software platform. In more advanced applications, manipulators operate based on sensor and visual feedback (*e.g.*, to implement visual servoing and force based control). Finally, many modern systems provide advanced operator consoles based on visual feedback, virtual reality, and a graphical user interface. To minimize the development effort, it is desired to build on an existing software platform. However, the diversity of robotic research areas, applications, and robotic hardware has not fostered the development of a commonly used platform.

The necessity for such a platform can be understood by looking at our previous work regarding the utilization of robotic manipulators for decommissioning tasks[3][4][16]. Our disassembly system first utilized the robot control library RCCL [25] and a Puma 560 robot. This structure was not flexible enough to implement more complex controller types, as required in disassembly operations, because the servo level was implemented on a proprietary Mark II controller. To increase flexibility, we developed a servo control

program executing on the QNX 4 real-time operating system [9] along with an interface to the robot control library RCCL. Subsequently, due to the limitations of this system, we started over again and ported ARCL [14] to QNX 4. Although at all stages of the project, the result was a working platform [16], each platform had several disadvantages. First, a lot of development time had to be spent in areas that were not really part of the research issue: Porting libraries, writing interfaces between different software packages, studying of code written by others, *etc.* Second, the system was very proprietary to the application and the hardware environment. For example, later in this project we wanted to integrate the WAM [17] into our disassembly system. This modification would require a lot of effort because it would involve the development of a complete new servo control program for the WAM as well as modifications to ARCL and all GUI programs. Finally, the system did not provide any means for control tuning and data logging of the servo control program.

The experience of this disassembly resulted in the formulation of the following requirements for a reusable software platform:

- *Flexibility*. The platform should be easily extensible for new components, especially for new manipulators. Modifications and extensions of the platform should be possible on all levels of the system (*i.e.*, task level, trajectory generation level, and servo control level).
- *Real-Time Support*. The platform should provide support for real-time operations. In addition, the user should be able to debug the real-time code, log and plot control signals, and tune the controller.

- *Modularity*. The platform should be structured into components that can be easily added and reconfigured. Also, researchers are often interested in just one special component of the platform (*e.g.*, they are interested in improving the servo control algorithm). Hence, modularity allows the researcher to focus on his interest without learning the internals of the rest of the platform.

Previous Systems and Research

To implement a robotic system, developers often utilize: i) robot control languages, ii) common programming languages like C/C++, iii) graphical control environments, and iv) robotic libraries for common programming languages.

Robot control languages provide a set of commands for implementing the control application. They are usually provided by the manipulator vendor and custom tailored to the specific manipulator type. Additionally, they are often based on proprietary hardware (*i.e.*, special purpose processors). Many of these languages do not allow modification (*e.g.*, implementing new control strategies) and extension (*e.g.*, interfacing to new system components such as sensors, visual feedback, *etc*). Hence, the scope of proprietary robot control languages is limited.The most direct method for developing a software platform is to implement a new solution from scratch, in a common programming language such as C. The advantage of this approach is that one is able to design the system in a way that fits exactly his needs and requirements. However, there are many disadvantages to this approach. Specifically, due to the complexity of the problem, development is very time consuming, error-prone, and requires a high level of skill.

To cut down on development time, solutions are available that simplify servo level control development. For example, QMotor [1] is a graphical control environment that requires the use of C/C++ to implement the control algorithm, but takes care of the programming issues related to timing, control tuning, data logging, and plotting. There are also several software platforms available that are based on MATLAB/Simulink, allowing the developer to create block diagrams instead of implementing the control as a C/C++ program. Real-Time Linux Target (RTLT) [18], Real-Time Windows Target [19], and WinCon [12] are examples of this concept. However, even though it is possible to implement a manipulator control system as a block diagram, the required functionality very often leads to complex block diagrams. In addition, the use of Simulink limits hardware related functionality and greatly increases the computational burden on the real-time platform (*i.e.*, as opposed to developing a C/C++ program).

Unfortunately, the development of a manipulator software platform, even when supported by the above-described environments, is an extensive task. That is why software libraries have been developed that provide data types and functions for robotic applications. The most well known example is RCCL [25]. The robot control library ARCL [14] is less complex and less powerful than RCCL, but follows the same concept. However, there is no straightforward way to modify the servo control level in RCCL and ARCL (*e.g.*, for Puma 560 robots, the servo control runs on a proprietary Mark II controller); therefore; it is not straightforward to implement new control strategies. Also, the large amount of code and complexity of RCCL and ARCL make them very difficult to understand and modify. RCCL and ARCL are good examples of procedural programming reaching its limits. That is, both libraries use programming constructs (*e.g.*,

function pointers) that emulate object-oriented concepts. However, since the implementation language (C) is not object-oriented, these constructs are difficult to understand and modify.

All of the solutions described above have one common problem. Specifically, if new functionality is needed or if new hardware is required, one must modify the source code (or the block diagrams, respectively). Modification of the systems internals is very error-prone. To overcome this problem, there have been object-oriented approaches to robot control libraries. For example, RIPE [20], developed at Sandia National Laboratories, defines an intuitive hierarchy of classes for robotic hardware. However, RIPE does not address the use of object-oriented concepts at the servo level. MMROC+ [21] uses an object-oriented design for error handling and simplification of the communication between processes. OSCAR [24] is an extensive library that addresses many issues of object-oriented design for robotic systems. It focuses mainly on the operational software layer (the layer between the user interface and the servo control). However, it is also very complex and requires multiple computing platforms.

Design Concepts

This section introduces the concepts behind the design of the QMotor RTK. These design concepts are utilized to meet the requirements stated above.

Object-Oriented Design

The procedural programming approach is based on two major concepts:

1. Data representation (*e.g.*, representation of the current position error of a manipulator).
2. Functions that operate on this data (*e.g.*, a function that calculates the required torques from the position error).

The above two concepts exist in the object-oriented approach as well. However, while procedural programming treats them separately, the object-oriented design ties them together. That is, they are grouped together in a construct called a *class*. There can be any number of classes in the system, identified by class names. For example, a PumaControl class would contain all of the data related to the control of a Puma robot (*e.g.*, current position, desired position, output torques, *etc.*) and all functions that are related to the control (*e.g.*, calculate the control algorithm, enable the arm power, *etc.*). To design an object-oriented system, the software engineer must carefully group data and functions in classes. Considering a software platform for robotic applications, this choice is intuitive. For example, classes represent physical objects such as the manipulator. Additionally, there are classes that represent functional objects (*e.g.*, the trajectory generator) and classes for GUI components. Consequently, the use of classes leads to a very intuitive modeling of the system.

There are several useful programming techniques utilized in object-oriented programming: i) data abstraction, ii) encapsulation, iii) polymorphism, and iv) inheritance [22]. Among other benefits, these programming styles have the following advantages:

- To use a class, an *object* of the class has to be created. There can be multiple objects that have the same class. Therefore, the representation of multiple physical objects

(*e.g.*, to control two manipulators of the same kind) is simply achieved by creating multiple objects of the same class.

- *Polymorphism* is the ability to provide the same interface to objects with differing implementations. Polymorphism is very useful for developing generic programs (*e.g.*, a trajectory generator can use the same generic interface for different manipulators).
- The use of classes leads to an open system that allows extension of the system via the design of new classes. Specifically, *inheritance* can be utilized. That is, any class can be defined to reuse generic data and functions from another class.

The idea of *inheritance* is now examined in detail. Once one starts to design classes for a manipulator control system, similarities between these classes become apparent. A class for a Puma 560 robot and a class for the WAM contain common functionality (*e.g.*, they both utilize a servo control algorithm, receive a desired trajectory, determine the current position by encoders, *etc.*).

A simple approach to develop both classes would be to first develop the class for the Puma 560 robot and then either rewrite the code for the WAM or copy the Puma 560 code and modify it (see Figure 10a). However, this approach leads to additional development effort; and hence; a higher probability of new errors. In addition, if the common functionality changes (*e.g.*, due to bug fixes or improvements), then changes need to be applied to all of the copies.

To avoid these disadvantages, the inheritance feature of object-oriented programming can be utilized. To use inheritance, a base class ManipulatorControl is defined. This base class contains the common functionality as described above. Then, the more specific classes for the Puma 560 and the WAM manipulator are *derived* from this base class.

Deriving means that they take over the functionality and data from the base class. Additionally, they are also able to redefine parts of this functionality and/or add new functionality and data. The classes of the Puma 560 robot and the WAM are then called *derived classes* (see Figure 10b). Once the base classes have been developed, they do not need to be re-compiled when a new derived class is added. That is, one does not need to change any source code of the base class to add a derived class. On the other hand, a modification of the common functionality in the base class is automatically reflected in all derived classes. Hence, inheritance greatly supports code reuse.

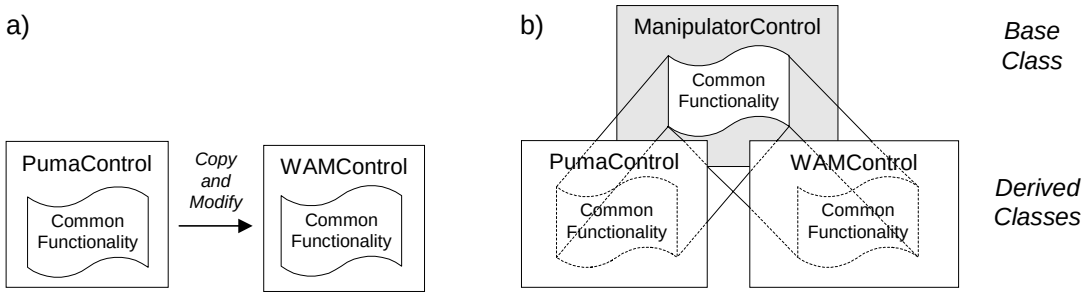


Figure 10. Code Reuse through a) Code Duplication, and b) Object-Oriented Programming

To further illustrate, we use class hierarchy diagrams to show the relationship between classes. In the example in Figure 11, the classes PumaControl and WAMControl are derived from the base class ManipulatorControl.

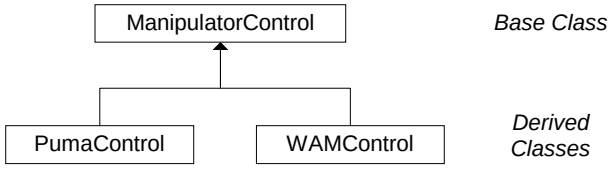


Figure 11. Example of a Simple Class Hierarchy

In a class hierarchy, one could also view base classes as "common" or more "generic" classes while derived classes are more "specific". One major task in object-oriented design is to separate common functionality from specific functionality.

A Lightweight Modular Bottom-Up Design

Using an object-oriented system design is just the first step in supporting code reuse. Whether the code will really be reused for many applications is highly dependent on its simplicity and its design. The smaller and less complex a robot control platform is, the simpler it is for system developers to learn and reuse it.

In previous work related to robot control software, a huge part of the software was dedicated to establishing real-time and distributed computation using multiple processors, architectures, and operating systems. Such an architecture leads to larger platforms that are more complex and heterogeneous. The technological progress in PC hardware and operating systems made heterogeneous architectures superfluous for many applications. Hence, this research proposes a design that is less complex for two reasons:

1. The design is homogeneous since all components are developed with the same programming language and executed on the same processor.
2. The design has very little overhead with regard to real-time programming and communication because these features are provided by a real-time operating system.

Previous platforms also attempted to include a wide range of robotic functionality. This approach attributes to additional complexity as well, and it often misses the desired outcome, because the spectrum of robotic research areas and applications are so broad that a robotic platform is never able to include all of them (*i.e.*, a specific application

often needs to modify the platform when adding new functionality). This research claims that it is easier for developers to build on a lightweight and solid base of low-level functionality than to add or modify a full-scale system. Hence, the design described here is a bottom-up approach that starts out by providing a flexible servo control level and then adding higher level components (*e.g.*, a joint level trajectory generator and a joint level teachpendant) on top of it. The important characteristics of this design are that it is modular and scalable. That is, the components of each level are independent from the other components (*i.e.*, a developer can use and modify the system at any level). Figure 12 shows some examples of different configurations of the QMotor RTK (all RTK components are indicated by white boxes while components added by the user are indicated by gray boxes).

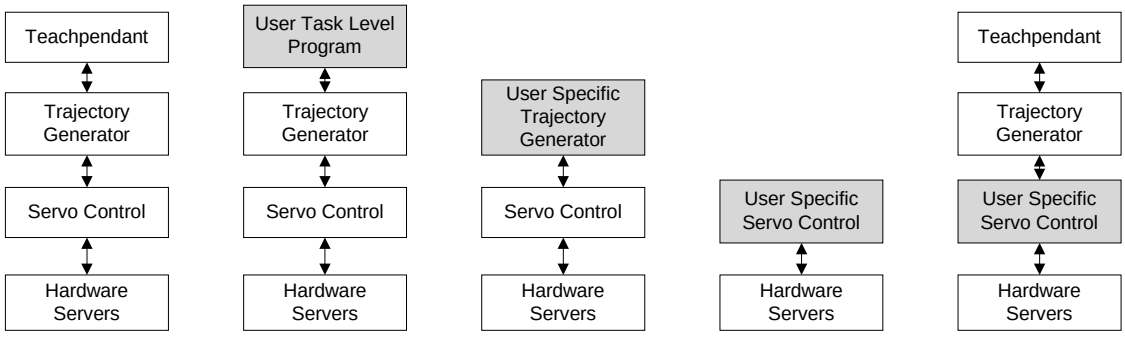


Figure 12. Example Configurations of the QMotor RTK

Run-Time Issues

For general-purpose applications, object-oriented programming has become more and more popular over the last two decades (mainly due to its code reusability and ability to handle complex systems). In real-time systems, however, the use of object-oriented

programming has caught on more slowly. This is due to the belief that object-oriented languages are inefficient and that they have unpredictable temporal characteristics [23]. Neither of these concerns can be attributed specifically to object-oriented programming. However, one must take care of the specific run-time requirements that appear in a manipulator control program (*e.g.*, real-time behavior, concurrency and communication). While the previous two sections addressed the logical design of the software, these issues are concerned with how the software resulting from the logical design is executed on a physical machine.

Concurrency

An important run-time requirement of a software platform for robotic applications is concurrency. The concurrency requirement can be understood by examining how a task level program, a trajectory generator, and a servo control program interact. The task level program is usually asynchronous, waiting for user input or the completion of operations (*e.g.*, when the manipulator reaches a target position). The trajectory generator usually runs at a fixed frequency. The servo control task also runs at a fixed frequency that is usually higher than the one of the trajectory generator. From the above scenario, it is clear that the task level program, the trajectory generator, and the servo control task must run concurrently. The above scenario describes a simple system with only one manipulator. More complex systems with multiple manipulators or other additional components might require even more concurrent tasks.

In the past, multi-processor systems have often been used to achieve concurrency (*e.g.*, different processor types and/or multiple computing platforms have been used in

[24] and [25]). Fortunately, increasing processing power has made the multi-platform approach unnecessary for most applications [7], and thereby, motivated the use of a single processor. There are two concepts to achieve concurrent behavior on a single processor system: i) execute one program that spawns multiple threads, or ii) execute multiple programs. Either of these concepts or a combination of both can be used. As an example, ARCL [14] uses threads. The advantage of threads is that the user does not need to take care of starting/terminating multiple programs. Also, threads are able to use the same common address space (*i.e.*, data can be easily shared between multiple threads). However, reconfiguration of a thread-based system requires recompilation. Recompilation is not necessary if multiple programs are used. That is, reconfiguration is simply accomplished by starting and terminating different programs.

The QMotor RTK design is based on the concept of using multiple programs due to two reasons. First, its modular design, as described in the last section, is better supported by this concept. Second, the real-time operating system QNX 4 does not support threads sufficiently.

Communication

It is necessary to establish communication between the concurrently executing tasks. A simple but effective and well supported QNX 4 communication mechanism is client/server message passing. In a client/server architecture, the server waits for messages from clients. Once it receives a message, it is processed and a reply is sent back to the client. This is very similar to a normal C/C++ function call. The difference is that the function is “called” by the client, but executed on the server. Client/server message

passing facilitates the implementation of generic components. That is, different servers can have a particular interpretation of the same message that is sent by a generic client. For example, a message to initiate an automatic calibration procedure would be processed by a Puma 560 control program, but discarded by an IMI control program, because the IMI robot [26] does not support automatic calibration.

Real-Time Performance

Another important requirement of a software platform for robotic applications is hard real-time performance. In this context, real-time performance means that certain tasks (e.g., a position control loop) are able to execute at a certain frequency without falling behind. It is a misunderstanding that high processing speed ensures real-time behavior. A programmer must meet three requirements to achieve real-time:

The processing time of a task is limited to a worst case. For example, dynamic memory allocation can take an unpredictable amount of time. Therefore, dynamic memory allocation (as performed by the operator "new") must be avoided in a real-time control loop.

Even if the processing time within a task is limited, this task can be interrupted and delayed by another concurrent task or an interrupt. Hence, the software designer must be able to control how processes interrupt each other. To accomplish this, real-time operating systems allow the programmer to set process priorities and scheduling algorithms.

The processing speed must be high enough to execute the worst case within the given time limit (i.e., in the context of a servo control loop, the sampling time of the control loop).

For best performance, the QMotor RTK utilizes the programming language C++. Concerns about the overhead created by a C++ compiler compared to C are not an issue. This overhead is minimal and can be neglected compared to the execution time of the control algorithms (see [*27*] for detailed information about C++ overhead).

The QMotor RTK System

Overview

The QMotor RTK system is structured as a combination of ready-to-execute programs and C++ libraries. The system is implemented on QNX 4, which is a very reliable real-time operating system. To avoid addressing timing, data logging, and plotting, the real-time control environment QMotor [1] is used as the base for the RTK. QMotor allows object-oriented control implementation in which control programs can be implemented as C++ classes. The RTK takes advantage of this concept and builds on the QMotor classes.

The QMotor RTK works only at the joint level, (*i.e.*, forward/inverse kinematics and Cartesian trajectory generation are not included). The RTK contains joint level position control programs for the WAM, the Puma 560, and the IMI manipulator. Also included is a generic joint level trajectory generator and GUI based teachpendant. Additionally, various utility programs are part of the RTK. The object-oriented approach is used in the control development as well as for the GUI components.

Figure 13 shows a typical QMotor RTK configuration. Each box represents a separate program. Lines represent message paths between the programs. The example system contains the teachpendant, the trajectory generator, the WAM servo control, and the WAM control panel. A ServoToGo S8 motion control board provides the hardware interface to the manipulator. To reconfigure the system, one only has to start different programs. For example, for the replacement of the WAM by a Puma 560 robot, one would start the program "`pumacontrol`" instead of "`wamcontrol`".

Figure 14 shows the complete class hierarchy of the QMotor RTK. Some classes have already been introduced; the others will be described later.

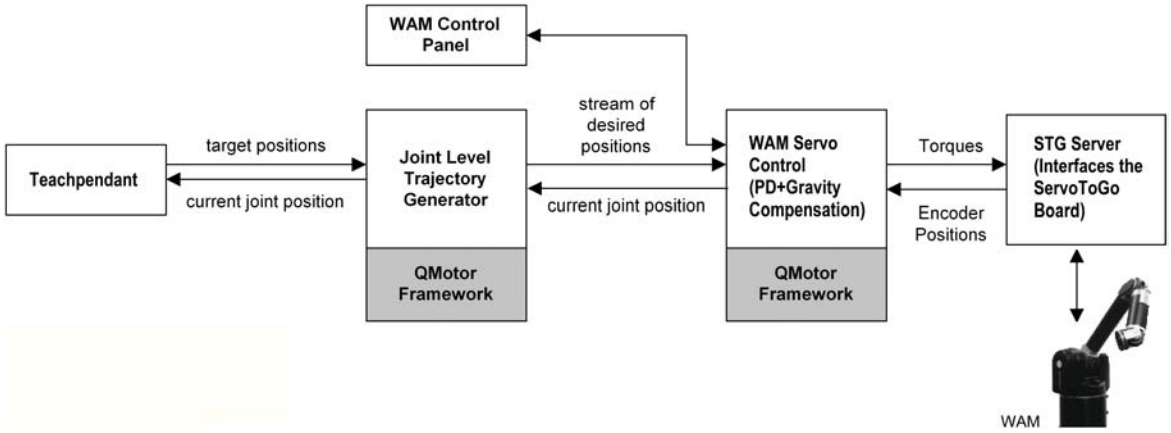


Figure 13. A Typical QMotor RTK Configuration

Data Logging, Plotting, and Control Tuning in an Object-Oriented Environment

Since the system design discussed in this chapter starts by implementing the servo control loop, one needs to implement a cycling control loop. Furthermore, it is desired to offer functionality with regard to tuning the control algorithm and gathering data during the control run. These tasks are not trivial to implement. They add large overhead to the programming effort. Thus, it is helpful to reuse existing software to save development

time. The QMotor environment is well suited for all real-time components of the RTK. QMotor is a general real-time environment for the development of any kind of control program. It contains three components: Hardware servers, the control program library and the QMotor GUI.

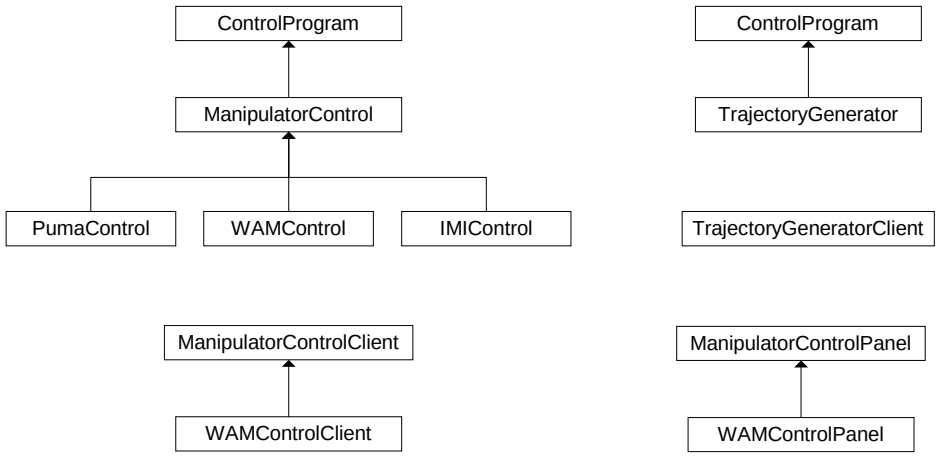


Figure 14. The QMotor RTK Class Hierarchy

The *Hardware Servers* provide a generic interface to motion control boards and other hardware components. Currently, the QMotor RTK utilizes hardware servers for the MultiQ and the ServoToGo S8 motion control boards.

The *Control Program Library* is utilized to implement the control algorithm as a C/C++ program. A base class, called `ControlProgram`, contains the framework for implementing control programs. The user derives the control application specific class from the `ControlProgram` class (*e.g.*, `ManipulatorControl`) and fills in the necessary functionality to implement the control algorithm. This functionality is contained in five functions that are left blank in the base class `ControlProgram` (see Figure 15):

- `enterControl()`: Called when the control program is loaded
- `startControl()`: Called every time the control execution is started
- `control()`: Called regularly at the control frequency
- `stopControl()`: Called when the control execution is stopped
- `exitControl()`: Called when the control program terminates
- `handleMessage()`: This function allows the control program to perform as a server, since `handleMessage()` is called when a message from another task (*i.e.*, the client task) arrives.

The class hierarchy shown in Figure 14 illustrates how further classes are derived from the `ControlProgram` class.

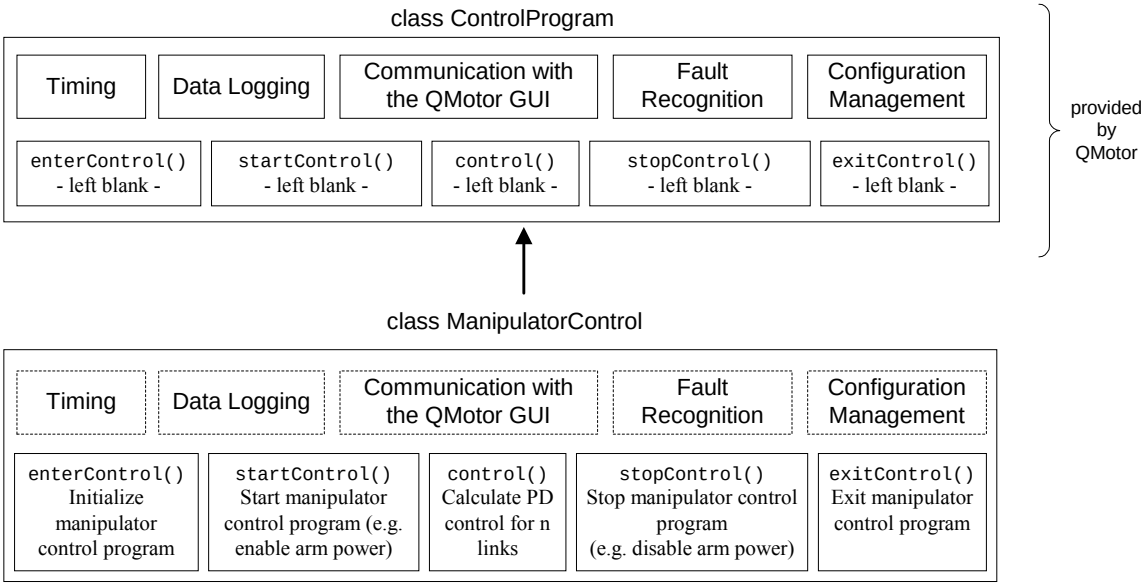


Figure 15. Deriving the `ManipulatorControl` Class from the `ControlProgram` Class

The *QMotor GUI* is used for selecting logging options, for plotting signals, and for control tuning. C++ variables of the control program can be registered as control

parameters to give the user the ability to change them from the GUI environment. In the same fashion, other C++ variables can be registered as log variables, thereby, making them available to be logged and plotted in the GUI. Figure 16 shows the QMotor main window and a plot window.

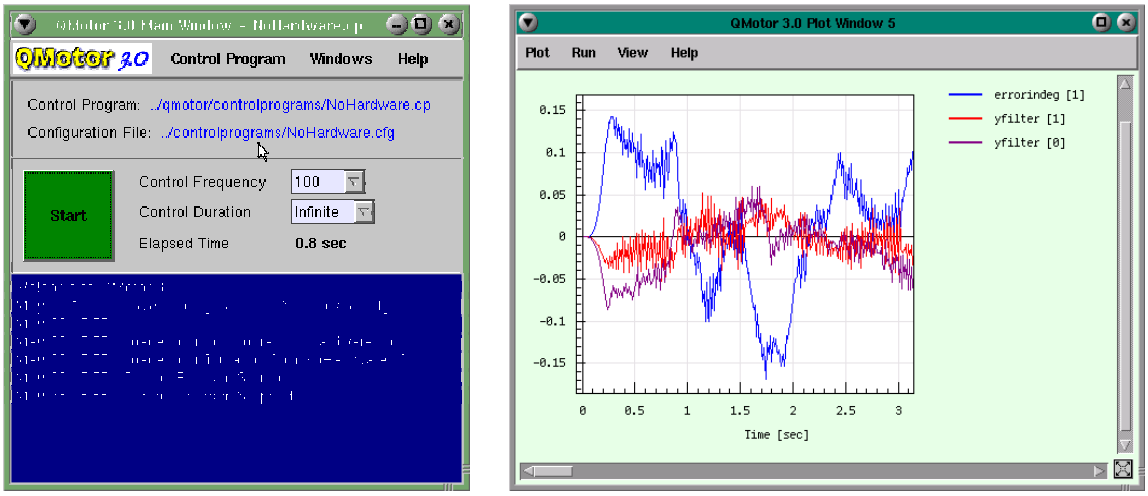


Figure 16. The QMotor Main Window and a Plot Window

Design of the `ManipulatorControl` Class

The lowest level of the QMotor RTK is the servo control level. This level consists of an independent PD joint tracking controller and the interface between the computer and the robot via a motion control board. The servo control level is implemented for three different manipulators: The Puma 560 robot, the WAM, and the IMI robot. As mentioned earlier, the first step in object-oriented design is to distinguish between common functionality/data and specific functionality/data. This concept is illustrated for the servo control level in Table 2 and Table 3.

Common Functionality	Specific Puma Functionality
• Communication with the I/O board • Setting output torques by setting voltages of the D/A converters • Position readings through encoders • Enabling/disabling arm power by setting digital outputs • PD position control • Determining velocities by backwards difference and filtering • Communication with client tasks (*e.g.*, to receive a desired trajectory) • Switching between control modes (*e.g.*, zero gravity mode/position control mode) • Safety checks for joint and torque limits • Generation of a simple test mode trajectory	• Automatic encoder calibration • Motor angles to joint angles transformation (to include coupling effects) • Gravity compensation
	Specific WAM Functionality
	• Automatic encoder calibration • Motor angles to joint angles transformation (to include coupling effects) • Joint torques to motor torques transformation • Gravity compensation • Torque ripple compensation
	Specific IMI Functionality
	• Disable arm power functions (There is no software control over the arm power)

Table 2. Common and Specific Functionality for the Manipulator Control

Common Data	Specific Puma Data
• Joint position and velocity for n joints • Control gains • Control modes • Joint and torque limits • Variables for I/O board communication • Other control parameters	• Potentiometer values
	Specific WAM Data
	• Torque ripple data
	Specific IMI Data

Table 3. Common and Specific Data for the Manipulator Control

All common functionality (Table 2, left column) and data (Table 3, left column) build the base class `ManipulatorControl`. This class, which is derived from the `ControlProgram` class, implements all the QMotor functions of the `ControlProgram` class that were left empty (*i.e.*, `enterControl()`, `exitControl()`, `startControl()`, `stopControl()`, `control()`, and `handleMessage()`). Figure 17 depicts how these functions are implemented. Note that the `handleMessage()` function is not shown in the flowcharts. All of the functions that are listed in the flowcharts of Figure 17 (*e.g.*, `control`(), `checkJointLimits()`, *etc.*) are *virtual functions*. That means that a derived class is able to redefine their functionality. Even if such a function is called from the base class, the redefined function will be used. This feature of inheritance is used by derived classes to implement specific functionality.

The Derived Classes of the ManipulatorControl Class

Some functions of the base class ManipulatorControl contain basic functionality; some are left empty (*e.g.*, the `doCalibration()` function is responsible for the automatic calibration procedure, and hence, is highly manipulator dependent). In the derived classes for the Puma 560 robot, the WAM and the IMI robot, certain functions are now filled in with new or modified functionality, as listed in Table 2 and Table 3. Since the major part of the work is done in the base class `ManipulatorControl`, the derived classes are significantly smaller and simpler.

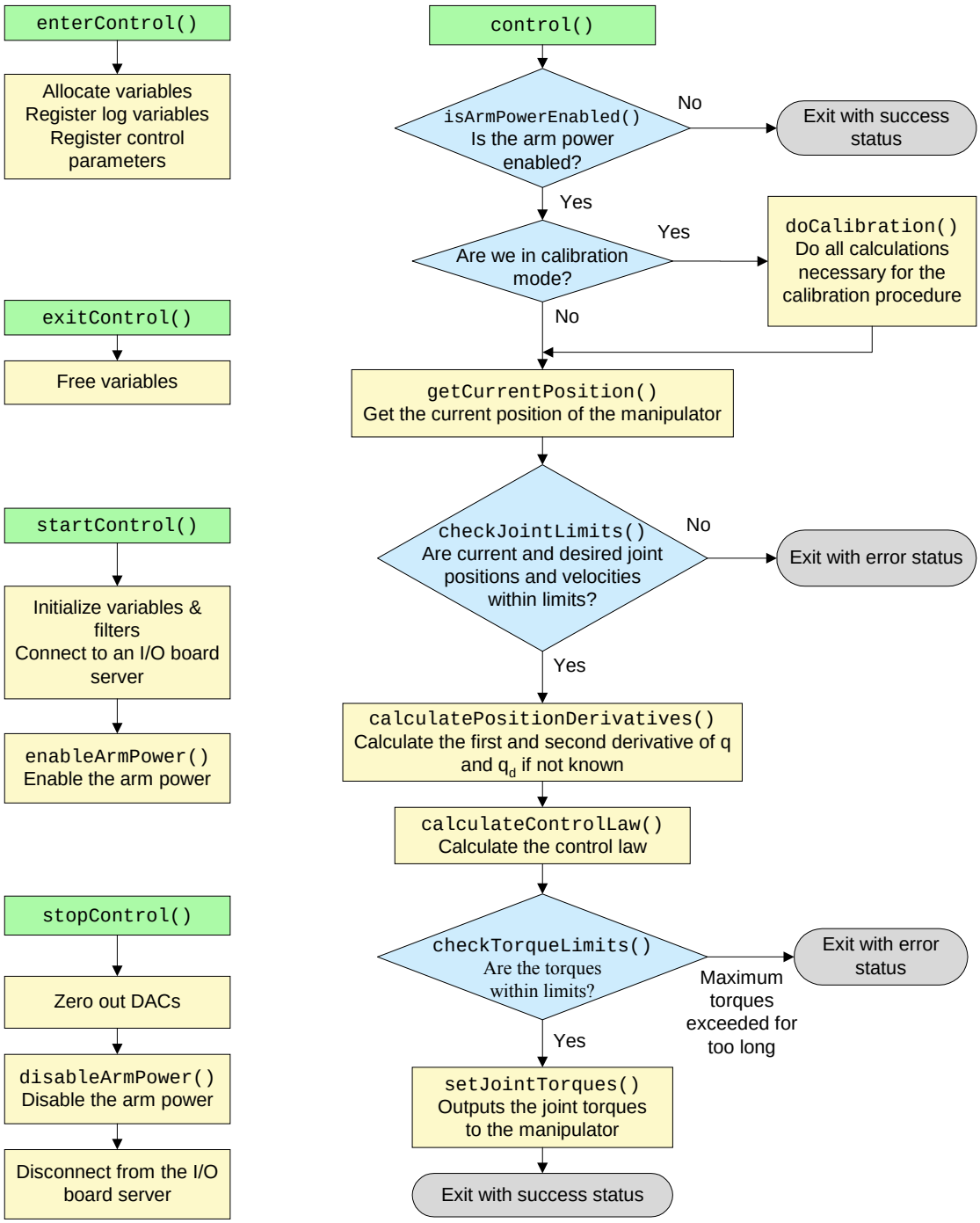


Figure 17. Flowchart of the Functions enterControl(), exitControl(), startControl(), stopControl(), and control()

The Class `PumaControl`

The following extensions are made in the `PumaControl` class:

- Variables and functions for the automatic encoder calibration procedure are added. This procedure determines the absolute position of the Puma by first getting a rough estimate from potentiometer readings and then performing the calibration by searching for the next index pulse.
- The function `getCurrentPosition()` is modified to take the coupling of joints 4, 5 and 6 into account.
- Gravity compensation is added. Gravity compensation calculates the torques resulting from the manipulator's weight and adds these to the output torque for compensation [13].

The Class `WAMControl`

The following extensions are made in the `WAMControl` class:

- Variables and functions for the automatic encoder calibration procedure are added.
- The functions `getCurrentPosition()` and `setControlTorque()` are modified to take the coupling of joints 2/3 and joints 5/6 into account.
- Gravity compensation is added.
- Torque ripple compensation is added.

The automatic calibration procedure of the RTK determines the absolute position of the WAM by moving joint by joint to its joint limits. The joint limit is detected by the position error exceeding a certain threshold. Then, a weighted sum of the encoder values at the minimum and the maximum joint limit determines the zero position of the WAM.

This procedure is a somewhat lengthy operation; however; it is necessary because the WAM does not contain hardware to determine its absolute position (*e.g.*, potentiometers).

For the gravity compensation, the WAM is modeled as three point masses. Two of these point masses are located at the center of mass of each link, and the third is located at the end of the robot arm. Lagrange's equation of the manipulator is simplified by the static conditions of the manipulator holding the position (*i.e.*, the joint velocities and the kinetic energy are zero). This simplified equation can be used to calculate the required torques [17]. To determine the mass parameters of the equation, a calibration procedure is implemented as a separate program. This program, called "gravity calibration utility", moves the WAM with the position control (without gravity compensation) to three predefined positions and measures the average torque to hold the WAM at this position. From those torque values, the mass parameters can be calculated.

The torque ripple compensation feature is responsible for compensating for the torque ripple of the electric motors of the WAM. Torque ripple is the resulting varying force on the rotor from the interaction of the motor magnets with the coils and iron cores in the stator. For many geared robots, the torque ripple is unnoticeable at the end effector. However, the low friction cable design of the WAM transmits the ripple to the end effector. This becomes especially apparent when a person manually pushes the WAM around in zero gravity mode. To compensate for torque ripple, first the so-called "torque ripple footprint" of each motor is determined. To do so, the QMotor RTK contains a program called "torque ripple calibration utility". This utility moves each motor one revolution and gathers the torque that is required to hold the motor at the current motor position (The motor position is defined by the number of encoder steps from the index

pulse). This procedure is performed for the forward and the backward direction. To compensate for torque ripple, the `WAMControl` class overrides the `setControlTorque()` function to add the compensation. The current motor position is determined and the average torque for the forward and backward directions from the calibration data is added to the control signal.

The Class `IMIControl`

The only modification in the IMIControl class concerns the arm power functions. As the IMI does not have arm power control by software, the arm power functionality is removed in the derived class IMIControl.

The Manipulator Control Client Classes

The manipulator control program (*i.e.*, either for the WAM, the Puma, or the IMI) performs as a server since it receives and processes messages from other programs. A client program sets control modes and creates the desired trajectory by sending messages to the manipulator control program. To simplify the sending of these messages, a separate class `ManipulatorControlClient` is created that wraps the sending of messages into functions (*e.g.*, a function `setDesiredJointPosition()` sends a message along with the new setpoint to the manipulator control program). The programmer can now easily utilize this class to communicate with the manipulator control program (see Figure 18).

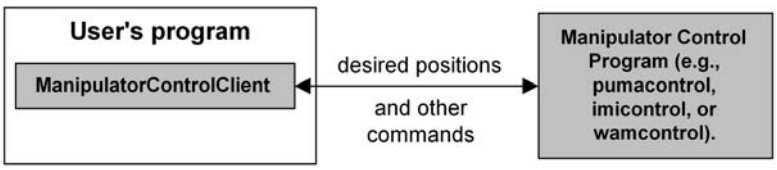


Figure 18. Message Passing between the Manipulator Control Client and the Manipulator Control

As the `ManipulatorControlClient` class is generic, this class works with the Puma control, the WAM control, or the IMI control program. To use specific functions of the WAM control program, the `WAMControlClient` class is derived from the `ManipulatorControlClient` class. The `WAMControlClient` adds functions to enable and disable torque ripple compensation. Note that the programmer can still utilize the `ManipulatorControlClient` class to communicate with the WAM control program. In this case, the WAM specific functions are not available but the resulting program is manipulator independent. There is no Puma specific client class because no additional client functions are necessary to address the Puma specific functionality. The same is true for the IMI.

The Trajectory Generator

The trajectory generator is also a QMotor control program; hence; the class `TrajectoryGenerator` is derived from the `ControlProgram` class to utilize timing, data logging, fault recognition, and communication with the QMotor GUI. The trajectory generator operates at the joint level. It uses the `ManipulatorControlClient` class to communicate with a manipulator control program. As this class is generic, the trajectory generator can be used with any manipulator supported by the RTK. The trajectory generator receives target positions from a client program and then calculates a smooth

trajectory to the target positions including acceleration and deceleration. The client can send multiple target positions asynchronously. The positions are stored into a queue and are processed in first-in first-out (FIFO) order. If there are multiple positions in the queue, two path segments are blended to ensure a smooth trajectory that does not stop the manipulator. The algorithm is based on the one described in Paul's book [28].

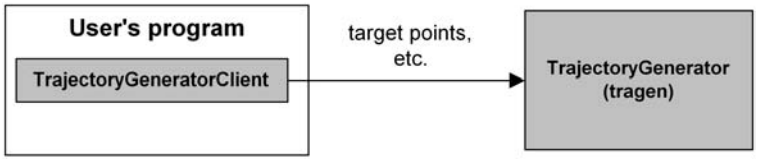


Figure 19. Message Passing between the `TrajectoryGeneratorClient` and the Trajectory Generator

Again, message passing is used as the communication medium between client and server. A `TrajectoryGeneratorClient` class simplifies sending the messages (see Figure 19).

The GUI Components

The design of GUI components is very important with regard to simplifying the use of the manipulator control system. A real-time operating system like QNX 4 allows GUI programs to coexist with high priority control programs. The RTK contains four GUI programs: the manipulator control panel, the WAM control panel, the manual-move utility, and the teachpendant.

QWidgets++ [29] is an object-oriented library for GUI programming under QNX. QWidgets++ was selected for the GUI programs of the RTK because it facilitates a pure object-oriented design. Specifically, GUI elements (*e.g.*, buttons, windows, *etc.*), also

called widgets, are represented by C++ classes. The manipulator control panels demonstrate the use of inheritance at the GUI level. The manipulator control panel (see Figure 20) is a generic control panel that works with all manipulators. The WAM control panel has two additional buttons to control torque ripple compensation and manual calibration (see Figure 21). To avoid duplicating the common features of both control panels, a base class `ManipulatorControlPanel` is created that implements the common features of the control panel. The class `WAMControlPanel` is derived from the class `ManipulatorControlPanel` to add the additional buttons to the control panel window.

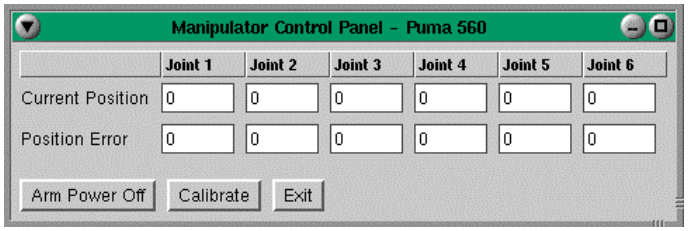


Figure 20. The Generic Manipulator Control Panel

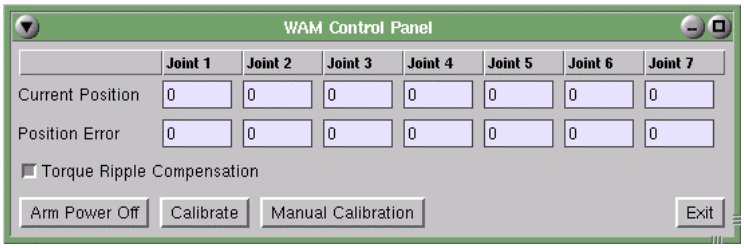


Figure 21. The WAM Control Panel

The manual-move utility (see Figure 22) is a simple program to test the servo control. It contains a slider for each joint. The user can move the sliders with the mouse and the manipulator follows immediately. This program illustrates how setpoints can be sent

asynchronously to the servo control. To avoid jerky movement, the frequency of the low-pass filter that processes incoming setpoints is set to a low value.

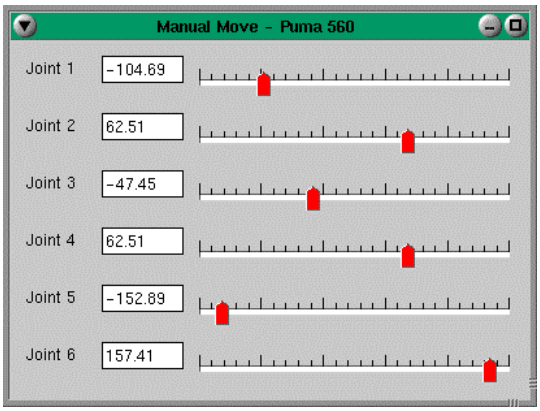


Figure 22. The Manual-Move Utility

The teachpendant (see Figure 23) uses the zero gravity mode of the manipulator to allow the user to push the manipulator around in the workspace.

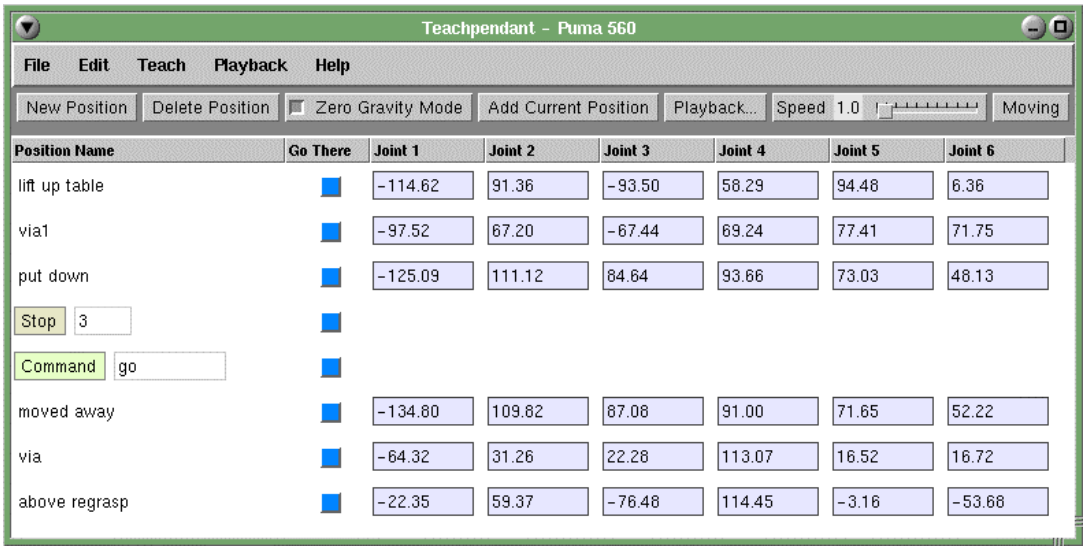


Figure 23. The Teachpendant

Once the user has moved the manipulator to a desired target position, this position can be added to a list of points. The teachpendant also utilizes the trajectory generator to move the manipulator back to the taught positions. It is also possible to cycle the manipulator through all or some of the taught positions. Additionally, the teachpendant is able to control the Barrett Hand, an advanced three-finger gripper. Hence, complete pick and place operations can be programmed with the teachpendant.

Modifying the System Using Inheritance

The previous sections explained how object-oriented techniques accelerate the addition of new components to the QMotor RTK. This section illustrates in greater detail how inheritance can be used during the addition of a new control algorithm. Specifically, in this simple example, the controller is extended from the PD controller to a PID controller. Figure 24 shows the function that calculates the PD control.

```
void ManipulatorControl::calculatePositionControl()
{
  // PD control plus acceleration feedforward
  for (int i = 0; i < d_numJoints; i++)
  {
    d_controlTorque[i] +=
      d_kp[i] * d_positionErrorRad[i]
      + d_kd[i] * (d_desiredVelocityRad[i] - d_velocityRad[i])
      + d_feedforwardAccelerationGain[i] *
        d_desiredAccelerationRad[i];
  }
}
```

Figure 24. The PD Control Calculation in the Base Class

```
class WAMPIDControl : public WAMControl                          [a]
{
  // ----- Constructors -----
public:
  WAMPIDControl (int argc, char *argv[])
    : WAMControl(argc, argv) {}
  ~WAMPIDControl () {};

  // ----- Manipulators -----
  virtual void calculatePositionControl();

  double d_ki[7];  // Integral Gain
  double d_prevPositionErrorRad[7]; // Position error of the
                                    // previous control cycle
  double d_positionErrorInt[7];     // Integrated position error
};


void WAMPIDControl::calculatePositionControl()
{
  // Call the base class to do the PD control
  ManipulatorControl::calculatePositionControl();                [b]

  // Then add the integral term
  for (int i = 0; i < d_numJoints; i++)                          [c]
  {
        d_positionErrorInt[i] += 0.5 * d_controlPeriod
              * (d_positionErrorRad[i] +
              d_prevPositionErrorRad[i]);
        d_prevPositionErrorRad[i] = d_positionErrorRad[i];
        d_controlTorque[i] += d_ki[i] * d_positionErrorInt[i];
  }
}
```

Figure 25. The Derived Class `WAMPIDControl`

The function is contained in the `ManipulatorControl` class. To implement the new controller, the new class `WAMPIDControl` is derived from the class `WAMControl` (see Figure 25, [a]). This class only redefines the function `calculatePositionControl()`. It calls the `calculatePositionControl()`

function from the base class and uses the algorithm for the PD control from there (see Figure 25, [b]). Then, the integral term is added (see Figure 25, [c]). Note that the function `calculatePositionControl()` of the base class and the derived class are distinguished by their class scope. That is, the prefix used for this function is "`ManipulatorControl::`" for the base class and "`WAMPIDControl::`" for the derived class.

Conclusions

This chapter presented an object-oriented design for a software platform for robotic applications. Because of the complexity of large-scale software environments, the QMotor RTK was designed to be a lightweight modular platform. In addition to the logical object-oriented design, the runtime design was included in this work to meet requirements with regard to real-time behavior, modularity, communication and concurrency. The RTK is a homogeneous, object-oriented system that is purely implemented as PC software. It utilizes a bottom-up design that is open and extensible from the servo-level to the task level.

The QMotor RTK reuses code for implementing different manipulator control programs and GUI programs. Specifically, base classes and classes for the Puma 560 robot, the WAM, and the IMI robot have been developed. Figure 26 relates the code size of the common and specific RTK components to the total code size. It illustrates that the implementation of new manipulators require a significantly smaller coding effort once the common base class is implemented. Note that a smaller coding effort also means a smaller source of coding errors. All new manipulator classes can refer to the well-tested

base classes. The WAM control class, for example, had been developed without the manipulator present. Large parts of the platform had already been tested with the Puma 560 robot. After the WAM arrived in the laboratories of Clemson University, the control program was debugged and tuned within three days. The IMI control program was implemented, debugged, and tuned in a single day.

We have also illustrated how object-oriented principles can be utilized to extend the system for new control algorithms shown with the example of a PID control. This example illustrates one of the primary advantages of the RTK design: Coding effort of extensions is significantly smaller compared to implementation from scratch. Additionally, there is no need to modify source code when extensions are needed.

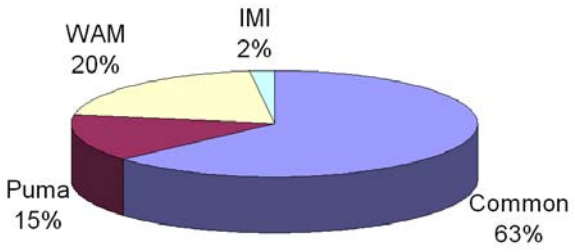


Figure 26. Code Size Ratios for the Supported Manipulators

CHAPTER 4

DESIGN AND IMPLEMENTATION OF THE ROBOTIC PLATFORM

Introduction

Robot control systems are very demanding with regard to software and hardware performance because their building blocks cover a wide range of disciplines found in robotics and software development (see Figure 27). Hence, it is desirable to create a common generic platform that can be reused by researchers for different applications. Considering the variety of robotic applications and research areas, this is a challenging task.

Hardware Interfacing	Real-Time Programming	Concurrency
Trajectory Generation	Real-Time Data Logging And Plotting	Interprocess Communication
Object-Oriented System Design	Complex System Design	Robotic Mathematical Functions
Utility Programs (Calibration, etc.)	Support for Different Robots, Sensors, and Tools	Networking
Hardware-Accelerated 3D Computer Graphics	Graphical User Interface	Programming Interface

Figure 27. Building Blocks of a Modern Robot Control System

Due to the lack of flexibility and performance of proprietary vendor-supplied robot control languages, previous research focused on building robot control libraries on top of a commonly used programming language (*e.g.,* "C") that was executed on a Unix workstation. RCCL [25] and ARCL [14] are examples of such libraries. Even though a

new level of flexibility and performance was achieved by using a common programming language, many robot control platforms developed in the 80's and early 90's were inherently complex due to the limitations of software packages and hardware components of that time. That is, most operating systems did not support real-time programming (fostering projects like RCI [30] and Chimera [31]). In addition, procedural programming languages like "C" tend to reach their limits with regard to reusability for complex projects; furthermore, the limited performance of hardware components forced system developers to utilize distributed architectures that integrated a mix of proprietary hardware and software.

Over the last ten years, many innovations in the computing area have occurred. Specifically, the advent of object-oriented software design [22] facilitated the management of more complex projects while also fostering code reuse and flexibility. For example, the robot control libraries RIPE [20], MMROC+ [21], OSCAR [24], and ZERO++ [32] utilized object-oriented techniques in robot programming. We have also witnessed the proliferation of real-time Unix-like operating systems for the PC [9], which facilitate the replacement of proprietary hardware components for real-time control [7]. In the hardware sector, we have witnessed the advent of high-speed low-cost PCs, fast 3D graphics video boards, and inexpensive motion control cards. Consequently, the PC platform now provides versatile functionality, and hence, makes complex software architectures and proprietary hardware components superfluous in most cases. The QMotor Robotic Toolkit (QMotor RTK) [33], for example, integrates real-time manipulator control and the graphical user interface (GUI) all on a single PC platform.

Despite the extensive functionality of the PC platform, much of the research in robot control software utilizes distributed and inhomogeneous architectures [20][24][32]. Besides the obvious advantages of distributed systems (*e.g.,* greater extensibility and more computational power), there are several disadvantages. Specifically, a distributed architecture requires a sophisticated communication framework, which increases the complexity of the software significantly. Additionally, deterministic real-time communication over network connections often requires expensive proprietary software and hardware. Specifically, the integration of multiple cooperating robots presents a challenge to distributed architectures. For example, it is often desired to modify the trajectory of one manipulator depending on certain signals of a cooperating manipulator (*e.g.*, the feedback of a force/torque sensor). In a distributed architecture, an additional effort must be spent on passing these signals between the components. Passing these signals and guaranteeing the required deadlines might even be impossible, depending on the flexibility of the system's components and the communication infrastructure.

Generally, the overall hardware cost of distributed systems is higher and users have to familiarize themselves with different hardware architectures and operating systems. Even though many platforms developed in the last couple of years attempted to be flexible, reconfigurable, and open, these platforms are seldom used and extended. Apparently, engineers consider it faster and easier to develop their applications from scratch. Indeed, from our own experience, the learning curve of installing, learning, and modifying robot control platforms of the past is steep.

Given the above remarks, the Robotic Platform is the first platform that has been designed to integrate servo control loops, trajectory generation, task level programs, GUI

programs, and 3D simulation in a homogeneous software architecture. That is, only one hardware platform (the PC), only one operating system (the QNX Real-Time Platform [9]), and only one programming language (C++ [27]) are used. This type of architecture has the following advantages:

Simplicity. A homogeneous non-distributed architecture is much smaller and simpler than a distributed inhomogeneous architecture. It is easier to configure, easier to understand, and easier to extend. Simplicity is critical with regard to motivating code reuse of the platform for different applications.

Flexibility at all Levels. All components of the platform are open for extensions and modifications. Many past platforms have utilized an open architecture at some levels, but other levels had been implemented on proprietary hardware such that they could not be modified.

High Integration. Since all components run on the same platform, a high integration is achieved, which allows for a simpler and more efficient cooperation between components. That is, communication between the components has little overhead and is often implemented by just a function call. Also, GUI components and 3D simulation are integrated with functional components.

Powerful Tools And Technologies – The Basis for the Robotic Platform

To reduce development effort and complexity, the Robotic Platform is based on general-purpose tools and technologies.

PC Technology. While in the past only expensive UNIX workstations provided the processing power necessary to control robotic systems, the PC has caught up or even

exceeded the performance of workstations [7]. Compared to UNIX workstations, a PC based system allows for a greater variety of hardware and software components. Additionally, these components and the PC itself are usually cheaper than their UNIX counterparts.

The QNX Real-Time Platform. The QNX Real-Time Platform (RTP) by QSSL [9] consists of the QNX 6/Neutrino operating system and additional components for development and multimedia. QNX 6 is an advanced real-time operating system that provides a modern microkernel-based architecture, a POSIX compliant programming interface, self-hosted development, 3D graphics capabilities and an easy device driver architecture. The RTP is also very cost-effective as it is free for non-commercial use and runs on low-cost standard PCs.

Object-Oriented Programming in C++. With regard to developing robot control software, object-oriented programming has several benefits over procedural programming. First, it provides language constructs that allow for a much easier programming interface. For example, a matrix multiplication can be expressed by a simple "`*`", similar to MATLAB programming. Second, object-oriented programming allows for a system architecture that is very flexible but yet simple. That is, the components (classes) of the system can have a built-in default behavior and default settings. The programmer can utilize this default behavior to reduce the code size or override it for specific applications. Finally, object-oriented programming supports generic programming, which facilitates the development of components that are independent from a specific implementation (*e.g.*, a generic class "`Manipulator`" that works with different manipulator types). All of the above benefits are based on the

general concepts of object-oriented programming: i) abstraction, ii) encapsulation, iii) polymorphism, and iv) inheritance [27, 33]. The language of choice is C++, as it provides the whole spectrum of object-oriented concepts while maintaining high performance [27].

Open Inventor. Open Inventor [34], developed by Silicon Graphics, is an object-oriented C++ library for creating and animating 3D graphics. Open Inventor minimizes development effort, as it is able to load 3D models that are created in the Virtual Reality Modeling Language (VRML) format. A variety of software packages are available that facilitate the construction of 3D VRML models that represent robotic components. The Robotic Platform also utilizes the functionality of Open Inventor to animate these components.

The QMotor System. Implementation of control strategies requires the capability to establish a deterministic real-time control loop, to log data, to tune control parameters, and to plot signals. For this purpose, the graphical control environment QMotor [1] is used for the Robotic Platform.

The Design and Implementation of the Robotic Platform

Design Overview

Each component of the Robotic Platform (*e.g.,* manipulators, the trajectory generator, *etc.*) is modeled by a C++ class. A C++ class definition combines the data and the functions related to that component. For example, the class "`Puma560`" contains the data of a Puma 560 robot (*e.g.,* the current joint position) as well as functions related to the Puma (*e.g.,* enabling of the arm power). Hence, the design of the Robotic Platform results from grouping data and functions in a number of classes in a meaningful and intuitive

way. A class can use parts of the functionality and the data of another class (called the *base class*) by deriving this class from the base class. This process is called inheritance, and it attributes heavily to code reuse and eliminates redundancy in the system. To extend the system, the user creates new classes. Usually, new classes will be derived from one of the already existing classes to minimize coding effort. The classes of the Robotic Platform include GUI components and a 3D model for graphical simulation. These types of components were traditionally found in separate programs (*e.g.*, see the RCCL robot simulator [25]). However, by including them in the same class, we can achieve a tight integration of the user interface, 3D modeling, and other functional parts. Additionally, object-oriented concepts for system extensions can also be used for GUI components and 3D modeling.

To illustrate how classes are derived from each other, class hierarchy diagrams are used. The main class hierarchy diagram of the Robotic Platform is shown in Figure 28.

Each arrow is drawn from the derived class (the more specific class) to the parent class (the more generic class). The classes of the Robotic Platform can be separated into the following categories:

Core Classes. The classes `RoboticObject`, `FunctionalObject`, and `PhysicalObject` build the basis of all robotic objects. The classes `RoboticPlatform` and `ObjectManager` contain functionality for overall management of robot control programs.

Generic Robotic Classes. Derived from the core classes are a number of generic robotic classes. These classes cannot be instantiated. Rather, these classes serve as base classes that implement common functionality while also presenting a generic interface to the

programmer (*i.e.*, these classes can be used to create programs that are independent from the specific hardware or the specific algorithm).

Specific Robotic Classes. Derived from the generic robotic classes are classes that implement a specific piece of hardware (*e.g.*, the class `Puma560` implements the Puma 560 robot) or a specific functional component (*e.g.*, the class `DefaultPositionControl` implements a proportional integral derivative (PID) position control).

The ControlProgram Class. This class is part of the QMotor system. All Classes that require a real-time control loop are derived from the `ControlProgram` class.

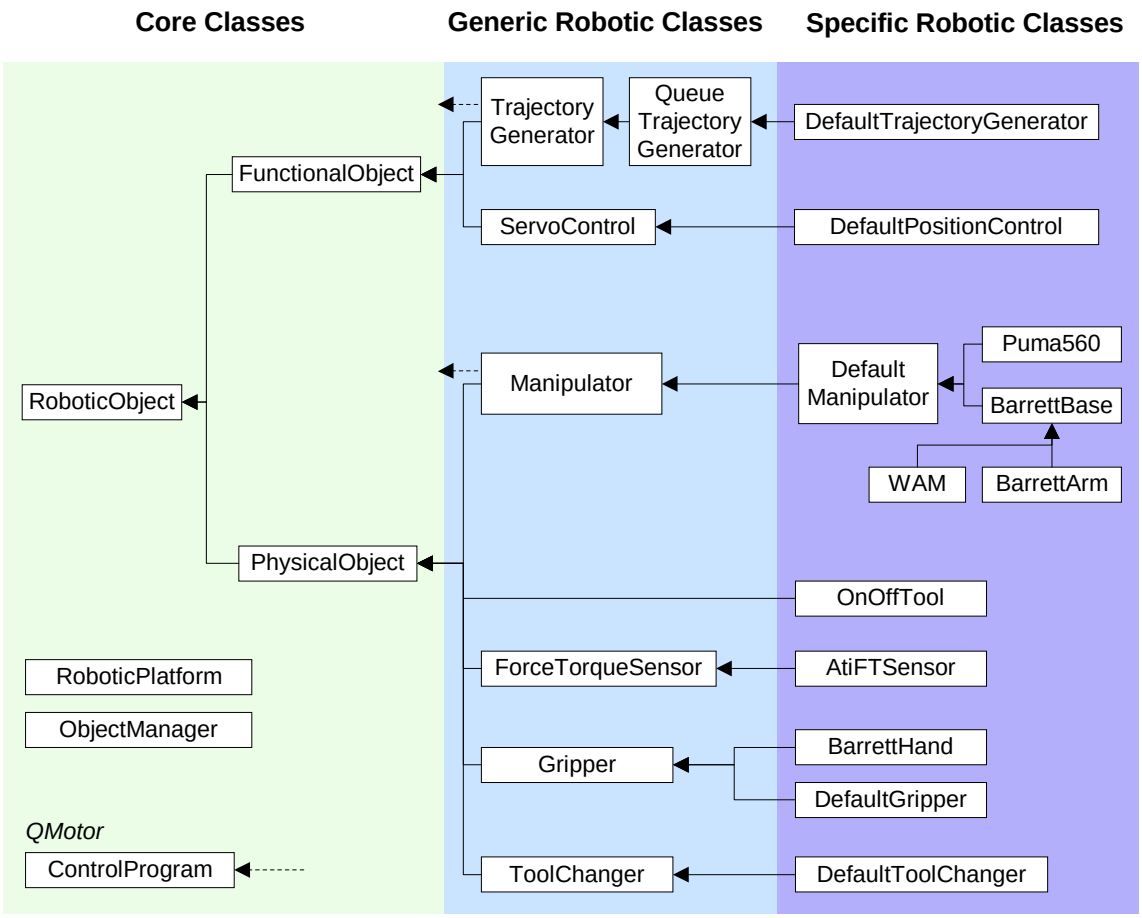


Figure 28. Class Hierarchy of the Robotic Platform

In addition to the classes shown in Figure 28, the Robotic Platform provides the classes of the math library, the manipulator model classes, and several utility classes. These classes and their class hierarchy will be described later in this chapter.

In a robot control program, the user instantiates objects from classes. When instantiating an object, memory for the object is reserved, and the object initializes itself. The user can create as many objects as desired from the same class. For example, it is straightforward to operate two Puma robots by simply creating two objects of the class `Puma560`. As soon as objects are created, the user can employ their functionality. The object manager maintains a list of all currently existing objects. With the object manager, it is possible to initiate functionality on multiple objects (*e.g.,* to shutdown all objects). The Scene Viewer is the default GUI of the Robotic Platform. It contains windows to view the 3D scene of the robotic work cell and a list of all objects. The overall run-time architecture is shown in Figure 29.

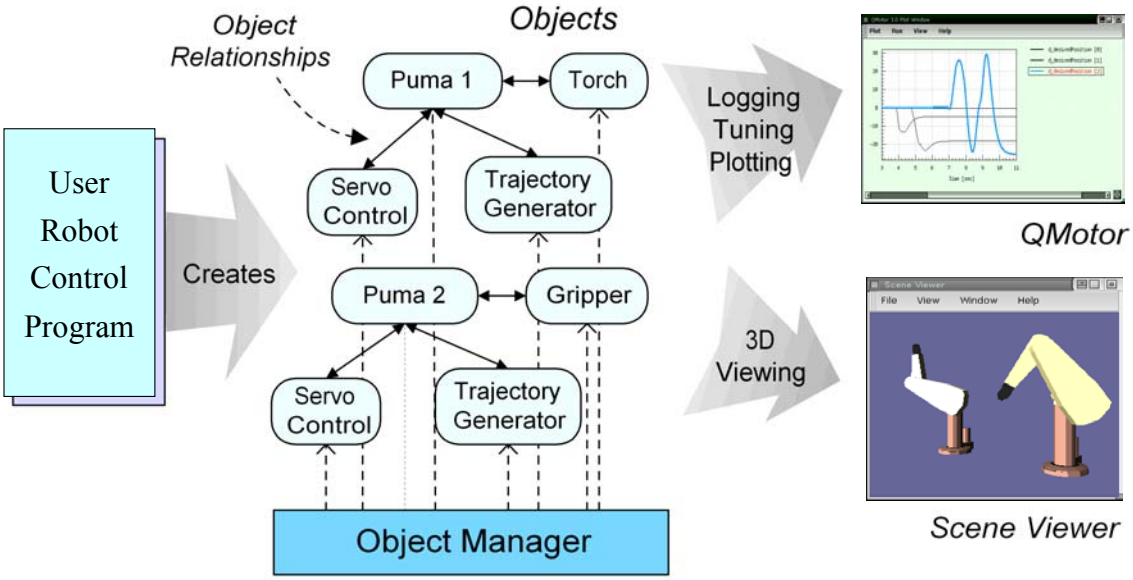


Figure 29. Run-Time Architecture of the Robotic Platform

In a robotic system, different components are related to each other. To reflect this fact, object relationships are established between objects. For example, objects can specify their physical connection to each other. Object relationships are implemented by C++ pointers to the related object. The object relationships in an example scenario are shown in Figure 30.

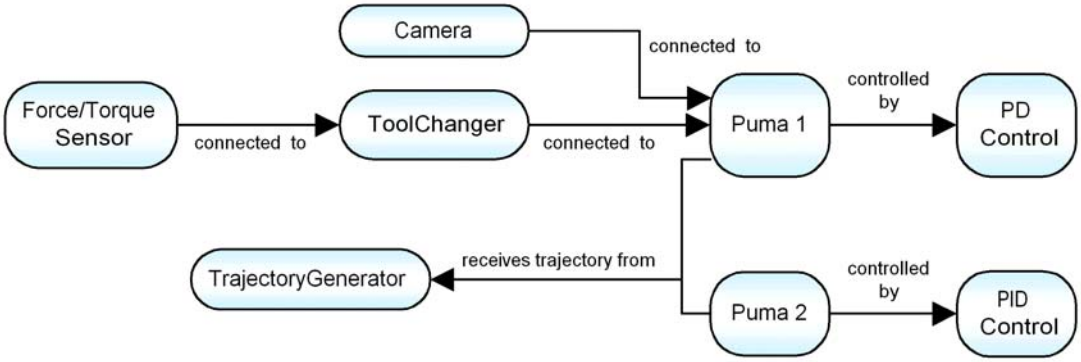


Figure 30. Object Relationships in an Example Scenario

The Core Classes

The class `RoboticObject` is the base class for all robotic classes. It defines a generic interface (*i.e.,* a set of functions that can be used with all robotic classes of the Robotic Platform). For example, a program can apply the `startShutdown()` function to any robotic object to initiate the shutdown of the object. To summarize, the following generic functionality is defined in the class `RoboticObject` (see also Figure 31):

Error Handling. Every object must indicate its error status.

Interactive Commands. Each object can define a set of interactive commands (*e.g.,* "Open Gripper") that the user can select in the object pop-up menu of the Scene Viewer.

Configuration Management. Each object can use the global configuration file to set itself up.

Shutdown Behavior. Each object is able to shutdown itself.

GUI Control Panel. Each object is able to create a control panel.

Message Handler. Each object has a message handler that can interpret custom messages.

Thread Management. Each object indicates if additional threads are required for its operation and provides functions that execute those additional threads.

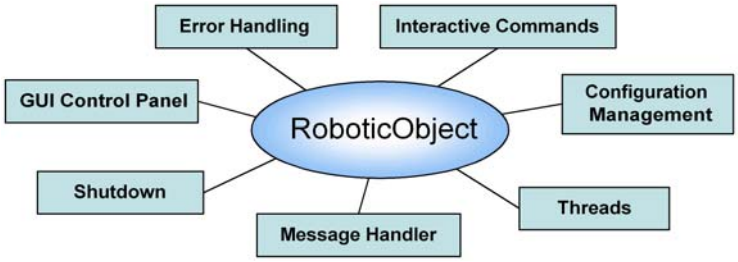


Figure 31. The Class RoboticObject

Note that the actual functionality is usually implemented in the derived class. However, the class `RoboticObject` also implements simple default functionality. This feature supports code reuse and simplicity by giving all classes derived from the class `RoboticObject` the choice to either take over this default functionality and/or implement new functionality.

The class `PhysicalObject` is derived from the class `RoboticObject`. It is the base class for all classes that represent physical objects (*e.g.*, manipulators, sensors, grippers, *etc.*). It defines a generic interface for these classes as illustrated in Figure 32. Specifically, the following generic functionality is defined in `PhysicalObject`:

3D Visualization. Every physical object can create its Open Inventor 3D model. The Scene Viewer loops through all physical objects to create the entire 3D scene.

Object Connections. A physical object can specify another object as a mounting location. By using this object relationship, the Scene Viewer draws objects at the right location (*e.g.*, the gripper being mounted on the end-effector of the manipulator).

Position and Orientation. The position and orientation specify the absolute location of the object in the scene (or the mounting location, if an object connection is specified).

Simulation Mode. Every physical object can be locked into simulation mode. That is, the object does not perform any hardware I/O, instead, its behavior is simulated.

The class `FunctionalObject` currently does not contain any functionality. It is only added as a symmetric counterpart to the class `PhysicalObject`. Functional robotic classes like the class `TrajectoryGenerator` are derived from the class `FunctionalObject`.

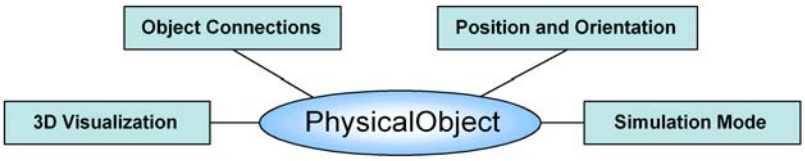


Figure 32. The Class PhysicalObject

Classes Related to the Control of Manipulators

The central components of any robotic work cell are manipulators. The class `Manipulator` is a generic class that defines common functionality of manipulators with any number of joints. Derived from the class `Manipulator` is the class

`DefaultManipulator`, which contains the default implementation for open-architecture manipulators. Open-architecture manipulators provide access to the current joint position and the control torque/force of the manipulator and hence, allow for custom servo control algorithms. Derived from the class `DefaultManipulator` are the classes that implement specific manipulator types. Currently, two manipulators are supported: the Puma 560 robot and the Barrett Whole Arm Manipulator (WAM) in both the 4-link and 7-link configuration. More information about the specific control implementation of these robot manipulators can be found in [33].

The class `DefaultManipulator` reads the current joint position and outputs the control signal continuously in a QMotor control loop. The actual calculation of the servo control algorithm is contained in a separate servo control object. The class of the servo control object must be derived from the class `ServoControl`, which defines the interface of a servo control. The default servo control is defined in the class `DefaultPositionControl`, which implements a PID position control with friction compensation. Manipulator classes like `Puma560` or `WAM` automatically instantiate an object of the class `DefaultPositionControl` for the convenience of the programmer. However, the programmer can switch to a different servo control anytime.

For the simulation of the manipulators, their dynamic model is required. Additionally, for Cartesian motion, forward/inverse kinematics and the calculation of the Jacobian matrix are needed. All these functions are located in the `ManipulatorModel` classes. The class hierarchy of the `ManipulatorModel` classes is displayed in Figure 33.

The trajectory generation is performed in separate classes. The base class `TrajectoryGenerator` defines the interface of a generic trajectory generator. A

trajectory generator is any object that creates a continuous stream of setpoints and provides this stream to a manipulator. The manipulator calls the `getCurrentSetpoint()` function of the trajectory generator to determine the current desired position. It is possible to switch between multiple trajectory generators. The class `QueueTrajectoryGenerator`, which is derived from the class `TrajectoryGenerator`, is a generic interface of a trajectory generator that creates the trajectory along via and target points. The class `DefaultTrajectoryGenerator`, which is derived from `QueueTrajectoryGenerator`, is the specific implementation of a trajectory generator that interpolates both in joint space and Cartesian space, including path blending between two motion segments at the via points.

To summarize, the manipulator classes only provide an interface to the manipulator itself. They do not include servo control and trajectory generation. These are performed in separate objects that are connected to the manipulator object. Figure 34 illustrates this relationship in an example scenario.

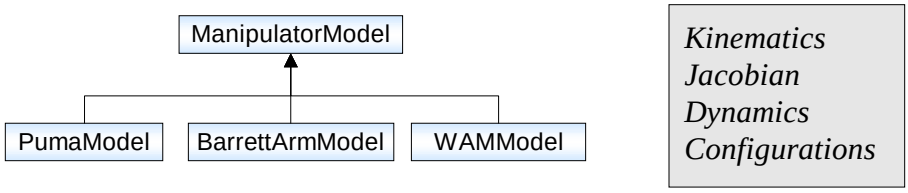


Figure 33. The ManipulatorModel Classes

Figure 34. Object Setup for the Servo Control and the Trajectory Generation of a Manipulator

The End-Effector Classes

In a typical robotic work cell, different end-effectors are connected to a manipulator. Consequently, the Robotic Platform provides several robotic classes that refer to end-effectors, as given below:

Gripper Classes. The class `Gripper` is the generic interface class of grippers. It defines the functions `open()`, `close()`, and `relax()`. The derived class `DefaultGripper` contains the default implementation, which utilizes two digital output lines to control the gripper, one digital line to open the gripper, and one to close it. The class `BarrettHand` is a specific class to operate the BarrettHand [17], an advanced three-finger gripper.

Force/Torque Sensor Classes. The base class `ForceTorqueSensor` defines the interface of a force/torque sensor. That is, it defines functions to read forces and torques. The derived class `AtiFTSensor` is the implementation of the ATI Gamma 30/100 Force/Torque sensor.

Toolchanger Classes. The class `ToolChanger` is the generic interface class of a toolchanger. It defines the functions `lock()`, `unlock()`, and `relax()`. The class `DefaultToolChanger` is the default implementation of a toolchanger, which uses two digital output lines to control the lock and unlock function of the toolchanger.

Configuration Management

The Robotic Platform utilizes a global configuration file, which is parsed by the object manager and the objects themselves to determine the system's configuration. This file is called `rp.cfg` by default. The format of the configuration file is as follows. For each

object, the configuration file lists the object name in brackets, the class name of the object, and some additional settings (see Figure 35). Table 4 lists available object settings. Derived classes are able to define additional settings.

Name of Setting	Description
`class <className>`	Specifies the class name of the object
`qmotorConfig <configFileName>`	Specifies a specific QMotor configuration file.
`position <x,y,z>` `orientation <r,p,y>`	Specifies the position and orientation of the object
`simulationMode on` `simulationMode off`	If "on" is specified, use simulation mode for this object

`display off`
`display solid`
`display wireframe`
	Specifies if and how the object is displayed in the Scene Viewer

Table 4. Object Settings of the Configuration File

```
[leader]
class Puma560
qmotorConfig leader.cfg

[secondrobot]
class WAM
position 300 0 0
simulationMode on
display solid

[gripper]
class BarrettHand
port /dev/ser1
```

Figure 35. An Example Global Configuration File

The Object Manager

The class `ObjectManager` implements the object manager. Every time a new object is instantiated in the user's robot control program, the object registers itself with the object manager. Similarly, every time an object is destroyed, it is removed from the object list of the object manager. The object manager contains functionality to loop through this list to perform operations on multiple objects. For example, the Scene Viewer retrieves a list of all objects that are derived from the class `PhysicalObject` to render each of them, and thereby, is able to render the entire 3D scene.

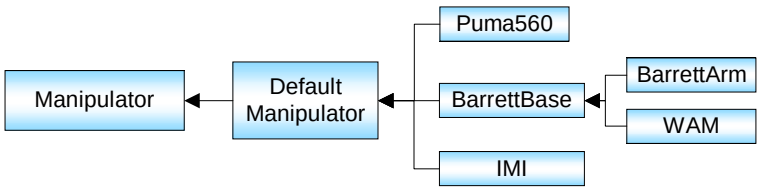


Figure 36. The Generic Class Manipulator and its Derived Classes

The functionality of the object manager is also necessary to allow for generic code. Generic code operates any object (*e.g.*, a manipulator object of class `Puma`) through the appropriate interface class (*e.g.*, the class `Manipulator`) by using C++ virtual functions. Hence, generic code does not need to be changed when an object of a different class is used (*e.g.*, the class `WAM`), as long as this object is derived from the same interface class. Generic code is very useful for code-reuse (*e.g.*, only a single generic trajectory generator must be written which can be used with different manipulator types). The following excerpt of generic code is manipulator independent code that works with the Puma robot,

the `WAM`, or any robot that is added in the future (see also the excerpt of the class hierarchy in Figure 36).

```
Manipulator *manipulator;    // Any manipulator
ObjectManager om;            // The object manager

manipulator = om.createDerivedObject<Manipulator>("leader");
  // Creates either a Puma or a WAM object, depending on
  // what is specified in the global configuration file
  // under the name "leader"

// Now, we can do generic operations
manipulator->enableArmPower();
cout << "Current End-Effector Coordinate Frame: "
     << robot->getEndEffectorTransform();
```

The above code first calls the function `createDerivedObject()` to create an object of the classes `Puma560`, `BarrettArm`, or `WAM`. Then, it operates this object via a pointer to the generic base class (*i.e.,* `Manipulator *`). In order to create the desired object, the `createDerivedObject()` function looks for the object name in the global configuration file (see Figure 35). Then, it reads the class name of the object from the configuration file and creates an object of this class[1]. Hence, to switch to a different manipulator type, only the class name in the global configuration file has to be changed when using a generic program.

---

[1] To be able to create an object dynamically from its name, the framework of the Robotic Platform creates a type database, which contains list of all classes defined in the robot control program.

The Concurrency/Communication Model

An inherent characteristic of robotic systems is concurrency. That is, while it is often sufficient for many software systems to run as a single task, robotic systems require components like the servo control to be executed concurrently with other components (*e.g.*, the trajectory generator). The Robotic Platform runs all concurrent tasks on the same PC.

The predecessor of the Robotic Platform, the QMotor RTK [33], executes multiple programs to achieve concurrency on a single processor. While this concept attributes to modularity, it is inconvenient to manage the startup and termination of multiple programs. Hence, an application that uses the Robotic Platform is compiled and linked to a single program instead. This program spawns multiple threads if concurrent execution is required. Once the program terminates, all threads are automatically terminated. Figure 37 shows an example user program.

At program start, only thread 1 is executing. At the initialization of the Robotic Platform library, a new thread is created that executes the 3D Scene Viewer. Then, the user utilizes a new object of a manipulator class. The instantiation of this manipulator object automatically spawns a third thread for the servo control loop. Hence, the first thread can go ahead and specify a desired trajectory for the manipulator, while the servo control loop and the Scene Viewer run in the background. To ensure real-time behavior of time critical tasks, the threads run at different priorities (*e.g.*, the servo control loop runs at the high priority 27).

Since threads access the same global address space, this address space can be used for communication between the threads. However, it is important to synchronize the access to avoid corruption of data structures. To allow for synchronized communication between the threads, message passing (as provided by the classes `Client` and `Server`) and standard thread synchronization mechanisms are used (as implemented in the classes `Barrier` and `ReaderWriterLock`).

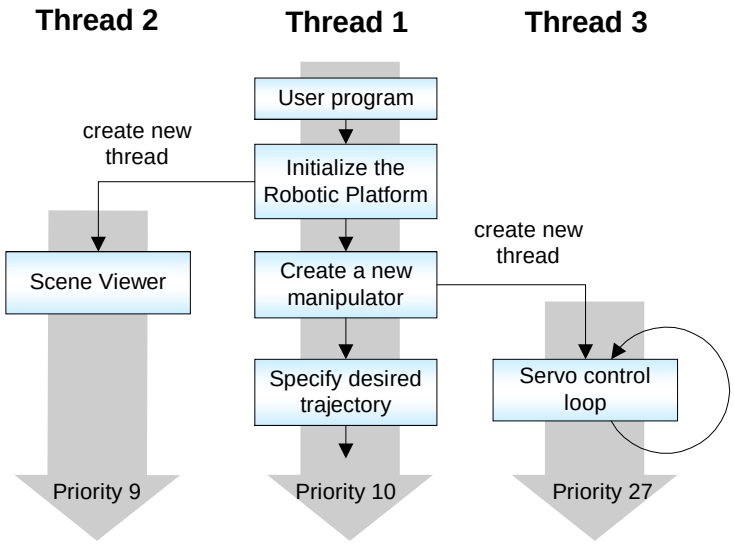


Figure 37. Creating New Threads for Concurrency

Real-Time, Plotting, and Control Tuning Capabilities with QMotor

QMotor [1] is a complete environment for implementing and tuning control strategies. QMotor consists of: i) a client/server architecture for hardware access, ii) a C++ library to create control programs, and iii) the QMotor GUI, which allows for control parameter tuning, data logging and plotting. To communicate with hardware, QMotor uses hardware servers that run in the background and perform hardware I/O at a fixed rate. Servers for

different I/O boards are available (*e.g.*, the ServoToGo board, the MultiQ board, and the ATI force/torque sensor interface board). The use of hardware servers provides an abstract client/server communication interface such that clients can perform the same generic operations with different servers. Hence, one can quickly reconfigure the system to use different I/O boards by simply starting different servers. For writing control programs, QMotor provides a library, which defines the class `ControlProgram`. To implement a real-time control loop, the user derives a specific class from the class `ControlProgram` and defines several functions that perform the control calculation and the housekeeping. Once a control program is implemented and compiled, the user can start up the QMotor GUI, load the control program, start it, and tune the control strategy from the control parameter window (see Figure 39). Furthermore, the user can set logging modes and display log variables in multiple plot windows (see Figure 38).

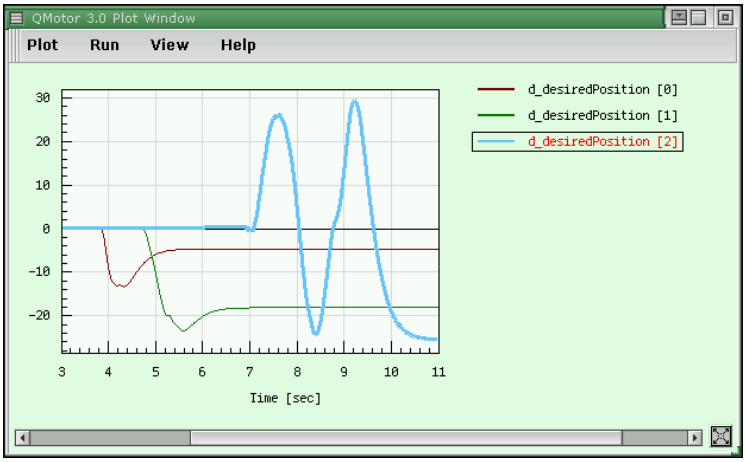


Figure 38. The QMotor Plot Window

To utilize QMotor for the Robotic Platform, classes that require a real-time control loop (*e.g.*, the class `DefaultManipulator`) are derived from the class

`ControlProgram`. Hence, these classes inherit the functionality of a control program (*i.e.*, real-time execution, data logging, and communication with the QMotor GUI). If a class is derived from the class `ControlProgram`, the base class `RoboticObject` automatically creates a new thread of execution that runs the control loop in the background.

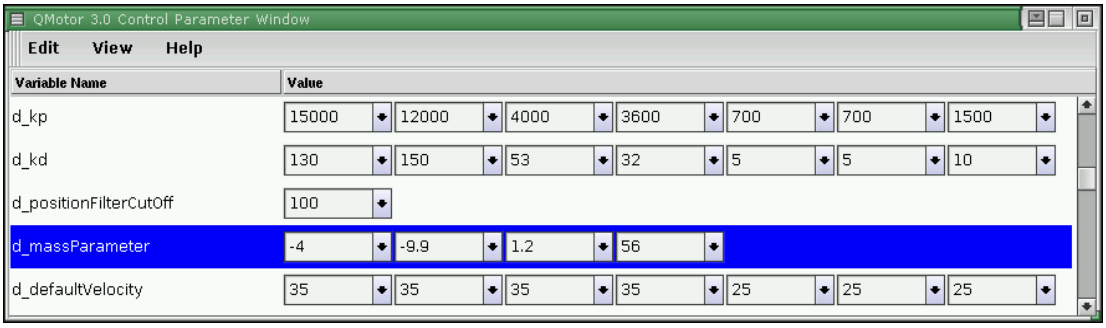


Figure 39. The QMotor Control Parameter Window

The Math Library

Past robot control libraries often introduced their own specific robotic data types. Most of these data types are based on vectors or matrices (*e.g.*, a homogeneous transformation is a 4x4 matrix). Hence, it is more feasible and flexible to use a general C++ matrix library and define robotic types on top of it. Most of the matrix libraries available for C++ use dynamic memory allocation, which risks the loss of deterministic real-time response [33]. Consequently, it would not be possible to utilize these libraries in many real-time components of the Robotic Platform. To overcome this disadvantage, special real-time matrix classes that use templates for the matrix size were developed for the Robotic Platform. This means that the matrix size is known at compile time and dynamic

memory allocation is not required. Besides being feasible for real-time applications, this approach also produces highly optimized code. Due to the use of templates and inline functions, the matrix classes can be as fast as direct programming. That is, with optimized implementation, the multiplication of two 2x2 matrices `C = A * B` is almost as fast as writing:

```
c11 = a11 * b11 + a12 * b21;
c12 = a11 * b12 + a12 * b22;  etc.
```

An additional advantage is that the compiler can check for the correct matrix sizes at compile time (*e.g.,* a matrix multiplication of two matrices with incompatible size is detected during compilation).

Figure 40 and Table 5 show the data types, the class hierarchy and the functionality of the math library. The classes `MatrixBase`, `VectorBase`, `Matrix`, `ColumnVector`, `RowVector`, and `Vector` are parameterized with the data type of the elements. The default element data type is double, which is the standard floating-point data type of the Robotic Platform. The classes `MatrixBase` and `VectorBase` are pure virtual base classes that allow for manipulation of matrices and vectors of an unknown size. Matrices and vectors of an unknown size are required during generic manipulator programming. Figure 41 shows an example program that uses the math library to calculate a position equation.

The math library also provides the classes `LowpassFilter` and `HighpassFilter` for numeric filtering, and the classes `Differentiator` and `Integrator` for numeric differentiation and integration. These classes are parameterized with the data type (*i.e.,*

they work with scalars, vectors, and matrices). The class `MathExeception` is utilized to detect error conditions in the math library.

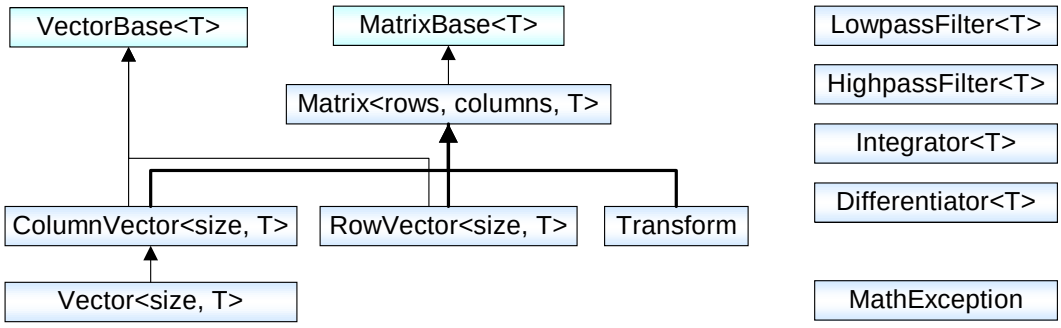
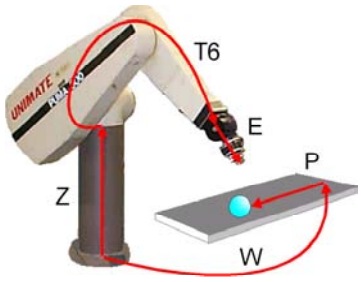


Figure 40. Class Hierarchy of the Math Library

Matrix Functions	Vector Functions	Transformation Functions
Multiplication/division Addition/difference Transpose Getting/setting elements Getting/setting sub-matrices Inverse Unit/Zero matrix Input/output	Length (2-norm) Cross-product Dot-product Element-by-element multiplication	Translation matrix Rotation matrix about x, y, or z axis Rotation matrix about an arbitrary vector Conversion from/to Euler angles Conversion from/to RPY angles

Table 5. Functions of the Matrix, Vector, and Transformation Classes




```
Transform Z = translation(0, 0, 0.7);
Transform E = translation(0, 0, 0.1);
Transform W = translation(1, 0.2, 0.3)
              * xRotation(M_PI);
Transform P = translation(-0.5, 0, 0);

// Solve Z*T6*E == W*P
Transform T6;
T6 = inverse(Z) * W * P * inverse(E);
```

Figure 41. Example Program for the Math Library

Error Management and Safety Features

Each object is responsible to maintain an error status. If a fatal error occurs, any object can request that the object manager shuts down the system. For example, this could be the case when a control torque exceeds its limit. For such a system shutdown, the object manager loops through all objects in the system and calls their `startShutdown()` function. Then, the object manager waits until all objects have completed their shutdown. The completion of the object shutdown is indicated by the `isShutdownComplete()` function.

Documentation

Critical for a high acceptance and a frequent reuse of a library is extensive documentation. The Robotic Platform has been developed by first creating manuals of all components and then using these manuals as requirement documents to guide the implementation. Documentation includes tutorials, external documentation and inline documentation. Example programs are frequently added, as they are essential for quick understanding of functionality. Doxygen [35] is an automatic documentation generator, which creates a reference manual from the inline documentation by processing the source files. Doxygen eliminates the redundancy of inline and external documentation. Doxygen is very versatile, as it creates the documentation in html format (for web publishing), latex format, and Microsoft Word rich text format (RTF).

The Graphical User Interface

GUI components are developed with the C++ GUI class library QWidgets++ [29]. QWidgets++ allows for object-oriented GUI programming. The GUI consists of three parts:

- The Scene Viewer is the default supervising GUI, which is opened automatically at startup of every Robotic Platform program.
- Additionally, each robotic class can have its own control panel. The control panels are opened from the Scene Viewer.
- Finally, the Robotic Platform includes several utility programs (*e.g.,* the Joint Move program and the Teachpendant), which have a GUI.

The GUI is further explained in the next section.

Using the Robotic Platform

The Scene Viewer and the Control Panels

Whenever a program of the Robotic Platform is executed, the Scene Viewer window opens up. It displays the entire 3D scene and also allows the user to open a window that displays a list of currently running objects in the system (see Figure 42). To create a 3D scene, the Scene Viewer loops through all objects that are derived from the class `PhysicalObject` and calls the `get3DModel()` function to obtain the Open Inventor 3D data of that object. Then, the Scene Viewer uses the object connection relationships (specified by the function `setConnection()` of the class `PhysicalObject`) to reorganize the Open Inventor object tree to display the 3D objects at the right position (*e.g.,* to display a gripper being mounted at the end-effector of a manipulator).

Furthermore, the Scene Viewer continuously updates the 3D scene with the current state of all objects (*e.g.,* it uses the current joint position of a manipulator to display the joints in the correct position). Hence, the 3D scene rendered in the Scene Viewer window always represents the current state of the hardware (in simulation mode, the simulated state of the hardware is represented). To select the best viewing position, the user can navigate in the 3D scene using the mouse. As many Scene Viewer windows as desired can be opened to view the 3D scene from different viewing positions at the same time. The user can also open the Object List window. This window displays a list of all objects that are currently instantiated by the robot control program, including class name and object name.

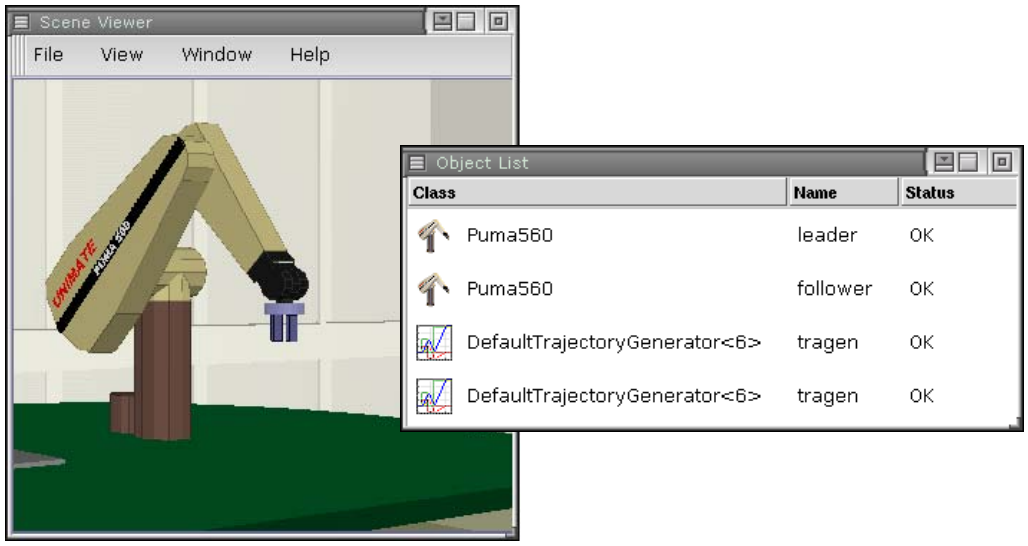


Figure 42. The Scene Viewer and the Object List Window

Each object has an individual pop-up menu (see Figure 43). This pop-up menu appears if the user either: i) right clicks on the object in the Scene Viewer rendering area, or ii) right clicks on an entry in the Object List window. The pop-up menu has options to hide

the object in the rendering area or to select between wire frame and solid display. Additionally, the pop-up menu displays interactive commands that are defined in the specific class of the object. For example, a gripper object has additional menu items to open, close, and relax the gripper. Finally, the user can open the control panel from the object pop-up menu. Each class can have an individual control panel. Figure 44 shows the control panel of the WAM as an example.

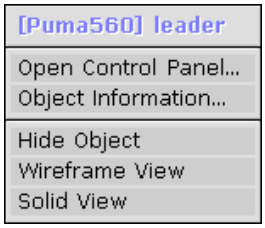


Figure 43. The Object Pop-Up Menu

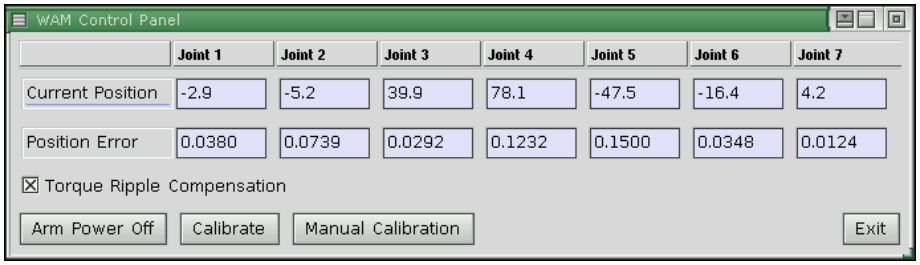


Figure 44. The Control Panel of the WAM Class

The Utility Programs

The Robotic Platform provides a couple of utility programs that help testing the system by performing simple operations. The ***Joint Move*** utility (see Figure 45) is a program to test the servo control of a manipulator. It contains a slider for each joint. The user can move the sliders with the mouse and the manipulator follows immediately.

The ***Teachpendant*** (see Figure 46) uses the zero gravity mode of the manipulator to allow the user to push the manipulator around in the workspace. Once the user has moved the manipulator to a desired target position, this position can be added to a list of points. The Teachpendant also utilizes the trajectory generator to move the manipulator back to stored positions. It is also possible to cycle the manipulator through all or some of the stored positions.

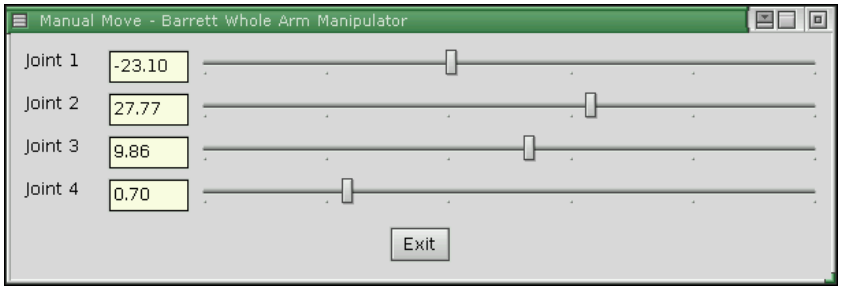


Figure 45. The Joint Move Utility

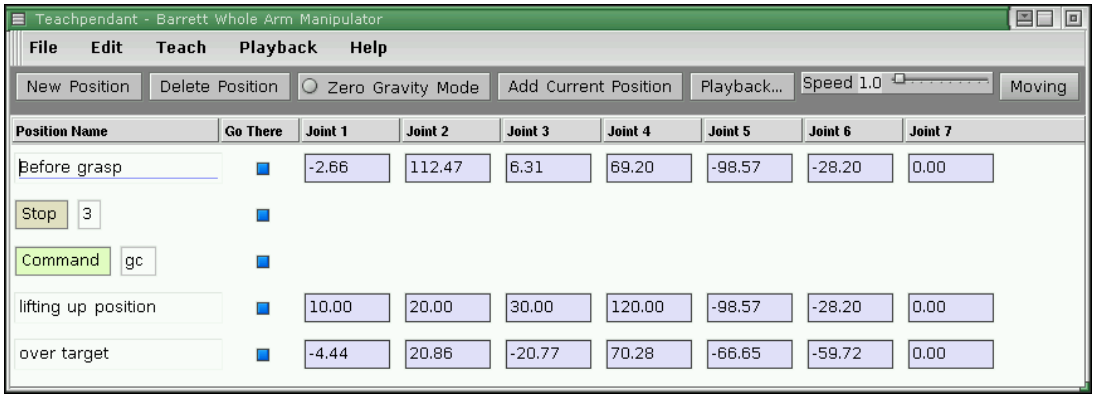


Figure 46. The Teachpendant

Writing, Compiling, Linking, and Starting Robot Control Programs

A robot control program is first compiled and then linked to the Robotic Platform library. The entire Robotic Platform (*i.e.,* all classes and the Scene Viewer) is contained

in a single library. As explained earlier, the system can easily be extended by adding new classes. If the extension is specific to a certain robot control program, the classes can be added to the code of that robot control program. If an extension is used in multiple robot control programs, it is probably more convenient to add the new functionality to the Robotic Platform library. To reflect extensions in preexisting compiled and linked programs, the Robotic Platform library is a dynamic library (*i.e.,* a program loads the library whenever it is started). Therefore, after the library is extended with new functionality, even programs such as the Teachpendant will take advantage of the new functionality without recompilation (*e.g.,* the teachpendant will be able to operate new manipulator types). Once the program is compiled and linked, the user can start it from the command line.

Figure 47 shows the listing of an example robot control program for a simple pick and place operation. Every robot control program first calls `RoboticPlatform::init()`. This function initializes the platform and starts up the Scene Viewer. The command line arguments are passed to `RoboticPlatform::init()` such that any Robotic Platform program can be started with certain default command line options (see Table 6). After `RoboticPlatform::init()` is called, the user's program creates all objects that are required for the robotic task (*i.e.,* a gripper object, a Puma 560 object, and a trajectory generator object are created). The final part of the example program utilizes the trajectory generator object and the gripper object to move the robot to the work piece, close the gripper, pick up the work piece, and drop it at the target position.

```
// Simple pick and place operation

#include “RoboticPlatform.hpp”

void main(int argc, char *argv[])
{
  // Initialize the Robotic Platform framework
  RoboticPlatform::init(argc, argv);

  // Create the objects required for the task
  Puma560 puma;
  DefaultGripper gripper;

  // Create the trajectory generator and connect it to the Puma
  DefaultTrajectoryGenerator<6> tragen;
  puma.setTrajectoryGenerator(tragen);

  // Create a transform that represents the end-effector
  // orientation (End-effector pointing down)
  Transform down = xRotation(M_PI);

  // Move to the object and pick it up
  gripper.open();
  tragen.moveTo(translation(0, 0.5, 0) * down);
  tragen.stop(1);
  gripper.close();
  tragen.stop(1);

  // Move to the target position and drop the object
  tragen.moveTo(translation(0.5, 0.5, 1) * down);
  tragen.moveTo(translation(1, 1, 0) * down);
  tragen.stop(1);
  gripper.open();
  tragen.stop(1);
}
```

Figure 47. A Simple Pick and Place Program for the Robotic Platform

Command Line Switch	Description
`-sim` `-nosim`	Enables/disables simulation mode for all objects
`-gui` `-nogui`	Enables/disables automatic start of the Scene Viewer
`-config <filename>`	Specifies the name of the global configuration file
`-qmotor`	Starts up the QMotor GUI

Table 6. Default Command Line Options of Robotic Platform Programs

Programming Examples

Virtual Walls

The virtual walls program is a good example on how to create a custom servo control. It also demonstrates the use of the manipulator model functions and the math library. Virtual walls are virtual planes in the manipulator's workspace that generate a reaction force once the manipulator is moved into it. Given a plane with the plane equation (using homogeneous coordinates):

$$\underline{w}\,\underline{x} = 0; \quad \underline{w} = \begin{pmatrix} \underline{n} \\ d_0 \end{pmatrix}$$

where $\underline{n}$ is the normal vector of the plane, and d_0 is the distance from the origin. If $\underline{x}_{EndEffector}$ is the current end-effector position, then

$$d = \underline{w}\,\underline{x}_{EndEffector} = \begin{cases} < 0 \text{ end-effector is outside the wall} \\ > 0 \text{ end-effector is inside the wall} \end{cases}$$

Using the control law

$$\tau = \begin{cases} 0 & d < 0 \text{ end-effector is outside the wall} \\ \underline{J}^T(k_f d\,\underline{n}) & d > 0 \text{ end-effector is inside the wall,} \end{cases}$$

creates a joint torque that resists moving the robot into the wall.

To implement a new servo control algorithm, a class is derived from the class `ServoControl` (1) (see Figure 48). The above virtual wall functions are implemented in the function `calculate()`, which calculates the control output. In the function `main()`, the robot object is created as usual. Additionally, an object of the virtual wall servo control class is created (2). Finally, the robot object is instructed to utilize the new servo

control instead of the default position control, and the gravity compensation is enabled to allow the robot to be pushed around (3).

```
#include “RoboticPlatform.hpp”

// ----- Create a new servo control class -----

template <int numJoints>
class VirtualWallServoControl : public ServoControl        (1)
{
 public:
  virtual void calculate()
  {
    // What is the distance of the end-effector to the wall?
    Transform t = d_manipulator->getEndEffectorPosition();
    double distance =
      dotProduct(d_wallCoefficients, t.getColumn(4));
    if (distance > 0)   // No control output
      return;

    // We are inside the wall. Generate reacting force
    Vector<3> force = distance * d_wallCoefficients * d_kf;

    // Convert to joint torque
    Vector<numJoints> pos = d_manipulator->getJointPositon();
    Vector<numJoints> torque;
    d_manipulator->endEffectorForceToJointTorque(pos, force,
                                                 torque);
    // Do the control output
    d_manipulator->setControlOutput(torque);
  }

  Vector<4> d_wallCoefficients;
  double d_kf;
};

// ----- Use the virtual walls servo control -----

void main(int argc, char *argv[])
{
  RoboticPlatform::init(argc, argv);

  // Virtual wall control for 6 joints
  VirtualWallServoControl<6> wallControl;                  (2)
  wallControl.d_kf = 0.01;
  wallControl.d_wallCoefficients = 0, 0, -1, 3;

  Puma560 puma;    // Create the robot object
  puma.setGravityCompensationOn();                         (3)
  puma.setServoControl(wallControl);
}
```

Figure 48. Virtual Walls Example

Comparison of Simulation and Implementation

A very interesting option is to forward the trajectory created by a trajectory generator to two manipulators. In this way, the motion of two manipulators with the same kinematics can be compared, or the behavior of a real manipulator can be compared with a dynamic simulation. The latter application is implemented in the robot program in Figure 49. First, two objects of the class `Puma560` are created, and one of them is set into the simulation mode. Then, both objects are connected to the same trajectory generator to receive the same trajectory.

```
#include “RoboticPlatform.hpp”

void main(int argc, char *argv[])
{
  // Initialize the Robotic Platform framework
  RoboticPlatform::init(argc, argv);

  // Create the robot objects, the second robot is simulated
  Puma560 puma;
  Puma560 pumaSimulated;
  pumaSimulated.setSimulationModeOn();

  // Connect both to the same trajectory generator
  DefaultTrajectoryGenerator tragen;
  puma.setTrajectoryGenerator(tragen);
  pumaSimulated.setTrajectoryGenerator(tragen);

  // Create the trajectory
  Vector<6> target;
  target = 0, 45, -90, 0, 0, 0;
  tragen.move(target);

  target = -50, 0, -70, 50, -80, 0;
  tragen.move(target);
}
```

Figure 49. Example Program to Send the Same Trajectory to Two Robots

Conclusions

The Robotic Platform is a software framework to support the implementation of a wide range of robotic applications. As opposed to past distributed architecture-based robot control platforms, the Robotic Platform presents a homogeneous, non-distributed object-oriented architecture. That is, based on PC technology and the QNX RTP, all non real-time and real-time components are integrated in a single C++ library. The architecture of the Robotic Platform provides efficient integration and extensibility of devices, control strategies, trajectory generation, and GUI components. Additionally, systems implemented with the Robotic Platform are inexpensive and offer high performance. The Robotic Platform is built on the QMotor control environment for data logging, control parameter tuning, and real-time plotting. A new, real-time math library simplifies operations and allow for an easy-to-use programming interface. Built-in GUI components like the Scene Viewer and the control panels provide for a comfortable operation of the Robotic Platform and a quick ramp-up-time for users that are inexperienced in C++ programming.

CONCLUSIONS

This doctoral dissertation has presented three different robot control platforms: the QRobot system, the QMotor Robotic Toolkit, and the Robotic Platform. These platforms demonstrate that proprietary hardware and inhomogeneous distributed architectures are not required anymore to provide a full-featured robot control platform. Instead, the PC platform is capable of integrating all components required for a robot control system. This leads to robot control platforms that are inexpensive and easier to use. Furthermore, the PC platform in combination with a real-time operating system makes a homogeneous architecture feasible, which utilizes a single programming language. This type of architecture in combination with advanced PC software tools and technologies allows to reach a new level of flexibility, extensibility, and ease-of-use.

REFERENCES

[1] N. Costescu, M. Loffler, M. Feemster, and D. Dawson, “QMotor 3.0 – An Object Oriented System for PC Control Program Implementation and Tuning”, Proc. of the American Control Conference, Arlington, VA, June 2001, pp. 4526-4531.

[2] Vladimir Lumelsky, “On human performance in telerobotics”, IEEE Transactions on Systems, Man and Cybernetics, 21(5).

[3] N. Costescu, M. Loffler, E. Zergeroglu, and D. Dawson, “QRobot - A Multitasking PC Based Robot Control System”, Microcomputer Applications Journal Special Issue on Robotics, Vol 18 No. 1, pp. 13-22.

[4] DOE Grant DE-FG07-96ER14728, “Advanced Sensing and Control Techniques to Facilitate Semi-Autonomous Decommissioning of Hazardous Sites”, http://ece.clemson.edu/iaal/doeweb/doeweb.htm.

[5] J. Lloyd, “Implementation of a Robot Control Development Environment”, Masters Thesis, McGill University, Dec.1985

[6] Unimation Inc., Danbury, Connecticut, “500 Series Equipment and Programming Manual”, 1983.

[7] N. Costescu, D. M. Dawson, and M. Loffler, "QMotor 2.0 - A PC Based Real-Time Multitasking Graphical Control Environment", June 1999 IEEE Control Systems Magazine, Vol 19 Number 3, pp. 68-76.

[8] D. Hildebrand, “An Architectural Overview of QNX”, Proceedings of the Usenix Workship on Micro-Kernels & Other Kernel Architectures, Seattle, April, 1992.

[9] QSSL, Corporate Headquarters, 175 Terence Matthews Crescent, Kanata, Ontario K2M 1W8 Canada, Tel: +1 800-676-0566 or +1 613-591-0931, Fax: +1 613-591-3579, E-mail: info@qnx.com, http://www.qnx.com.

[10] P. Corke, "The Unimation Puma Servo System", CSIRO Division of Manufacturing Technology, Preston, Australia.

[11] Trident Robotics and Research, Inc., 2516 Matterhorn Drive, Wexford, PA 15090-7962, (412) 934-8348, http://www.cs.cmu.edu/~deadslug/puma.html, E-mail: robodude@cmu.edu.

[12] Quanser Consulting, 102 George Street, Hamilton, Ontario, CANADA L8P 1E2, Tel: 1 905 527 5208, Fax: 1 905 570 1906, http://www.quanser.com.

[13] B. Armstrong, O. Khatib, J. Burdick , "The Explicit Dynamic Model and Inertial Parameters of the PUMA 560 Arm", Proc. IEEE int. conf. Robotics and Automation 1 (1986), pp. 510-518.

[14] P. Corke, "The ARCL Robot Programming System", CSIRO Division of Manufacturing Technology.

[15] J. Neider, T. Davis, and M. Woo, "OpenGL Programming Guide", Addison-Wesley Publishing Company, New York, 1993.

[16] M. Loffler, N. Costescu, E. Zergeroglu, and D. Dawson, "Telerobotic Decontamination and Decommissioning with QRobot, a PC-Based Robot Control System", Proc. of the IEEE Conference on Control Applications, Anchorage, AK, September 2000, pp. 24-29.

[17] Barrett Technologies, 139 Main St, Kendall Square, Cambridge, MA 02142, http://www.barretttechnology.com/robot.

[18] Z. Yao, N. P. Costescu, S. P. Nagarkatti, and D. M. Dawson, "Real- Time Linux Target: A MATLAB-Based Graphical Control Environment", Proc. of the IEEE Conference on Control Applications, Anchorage, AK, Sept. 2000, pp. 173-178.

[19] The MathWorks, 3 Apple Hill Drive, Natick, MA 01760-2098, http://www.mathworks.com.

[20] D. J. Miller and R. C. Lennox, "An Object-Oriented Environment for Robot System Architectures", IEEE Control Systems February 1991, pp. 14-23.

[21] C. Zieliński, "Object-oriented robot programming", 1997, Robotica volume 15, Cambridge University Press, pp. 41-48.

[22] B. Stroustrup, "What is 'Object-Oriented Programming'?", Proc. 1st European Software Festival. February, 1991.

[23] T.E. Bihari and P. Gopinath, "Object-Oriented Real-Time Systems: Concepts and Examples", IEEE Computer, December 1992, pp. 25-32.

[24] Chetan Kapoor, "A Reusable Operational Software Architecture for Advanced Robotics", Ph.D. thesis, University of Texas at Austin, December 1996.

[25] J. Lloyd, M. Parker and R. McClain, "Extending the RCCL Programming Environment to Multiple Robots and Processors", Proc. IEEE Int. Conf. Robotics & Automation (1988) pp. 465 – 469.

[26] "Direct Drive Manipulator Research and Development Package, Operations Manual", Integrated Motion Inc., Berkeley, CA, 1992.

[27] B. Stroustrup, “An Overview of the C++ Programming Language”, Handbook of Object Technology, CRC Press. 1998. ISBN 0-8493-3135-8.

[28] Richard P. Paul, “Robot Manipulators: Mathematics, Programming, and Control”, MIT Press, Cambridge, Mass., 1981.

[29] Quality Real-Time Systems, LLC., 6312 Seven Corners Center, Falls Church, VA 22044, Website: http://qrts.com.

[30] J. Lloyd, M. Parker and G. Holder, “Real Time Control Under UNIX for RCCL”, Proceedings of the 3rd International Symposium on Robotics and Manufacturing (ISRAM ’90).

[31] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, “CHIMERA II: A Real-Time UNIX-Compatible Multiprocessor Operating System for Sensor-based Control Applications”, tech. report CMU-RI-TR-89-24, Robotics Institute, Carnegie Mellon University, September, 1989.

[32] C. Pelich & F. M. Wahl, “A Programming Environment for a Multiprocessor-Net Based Robot Control Unit”, Proc. 10th Int. Conf. on High Performance Computing, Ottawa, Canada, 1996.

[33] M. Loffler, D. Dawson, E. Zergeroglu, N. Costescu, “Object-Oriented Techniques in Robot Manipulator Control Software Development”, Proc. of the American Control Conference, Arlington, VA, June 2001, pp. 4520-4525.

[34] Josie Wernecke, “The Inventor Mentor”, Addison-Wesley, ISBN 0-201-62495-8.

[35] Doxygen homepage, http://www.stack.nl/~dimitri/doxygen.