QMotor 3.0 and the QMotor Robotic Toolkit: A PC-Based Control Platform

Markus S. Loffler, Nicolae P. Costescu, and Darren M. Dawson

Department of Electrical and Computer Engineering
Clemson University, Clemson, SC 29634-0915
[loffler, ncostes, ddawson]@ces.clemson.edu

Introduction

QMotor 3.0 is a QNX based object-oriented single-processor software environment that allows the implementation of real-time control algorithms as C++ programs on standard Intel-processor-based personal computers (PCs). The QMotor 3.0 graphical user interface integrates functionality for testing and tuning of these control programs. In addition, it provides advanced data logging, plotting, and data exporting capabilities. The control program, as well as the development tools and the graphical user interface (GUI), can all execute simultaneously on the PC due to the deterministic response of the operating system (OS). This architecture replaces the traditional multiprocessor Host/DSP board architecture often used in control applications. Advantages of a single-processor system include reduced cost and lower complexity, as well as increased flexibility and upgradability. A client/server architecture decouples the control program from the hardware so that QMotor 3.0 can easily be extended to work with new hardware. QMotor 3.0 has been used successfully in many of the control experiments performed by the Clemson Control and Robotics group, including motor and robot control, active magnetic bearing experiments and vibration control for flexible structures. Some of these experiments are documented in [1], [2], [3] and [4].

The high performance and flexibility of QMotor 3.0 allow for the implementation of many different control applications ranging from simple linear control routines to complex, nonlinear, multidimensional control algorithms. As an example of a complex control application, the control of robot manipulators with QMotor 3.0 is addressed in the second part of the article. Specifically, we present the design of the QMotor Robotic Toolkit (QMotor RTK). The Robotic Toolkit is a set of QMotor 3.0 control programs, C++ libraries, and utility programs for robot manipulator control and trajectory generation. It includes servo control programs for the Puma 560 manipulator, the Barrett Whole Arm Manipulator (WAM) [5], and the Integrated Motion Inc. (IMI) two-link manipulator, as well as a joint-level trajectory generator and a GUI. The RTK is a good example for demonstrating the open architecture of QMotor 3.0; that is, the RTK illustrates the use of multiple cooperating control programs and the integration of GUI programs. Additionally, the RTK demonstrates how object-oriented techniques can be applied to control implementations to ensure extensibility and code reuse.

Previous Research

WinMotor, QMotor 1.0, and QMotor 2.0 [6] are the predecessors to QMotor 3.0. WinMotor and QMotor 1.0 are heterogeneous PC Host/DSP single-board computer (SBC) systems, where the control executes on a DSP SBC while a Host PC is used for the GUI to provide plotting, data logging, and control parameter tuning functions. The primary disadvantages of the Host/DSP architecture are the high cost of the hardware, the limited flexibility of the system, and the complexity of the software.

To overcome these disadvantages, we developed QMotor 2.0, a single-processor control environment that executes both the control program and the GUI on the same processor. The use of modern high-speed consumer-grade PCs coupled with a PC real-time OS allowed the development of a system that implements the control algorithm and the user interface on one CPU. This configuration reduced the cost and complexity of the system, for developers as well as for users. Several disadvantages of QMotor 2.0 became apparent with use. The first version of QMotor 2.0 only supported the MultiQ motion control board for hardware interfacing. To support additional hardware (*e.g.,* cameras, fast A/D

boards, more I/O channels), multiple versions of QMotor for different hardware boards had to be developed since QMotor 2.0 did not support a flexible hardware interfacing architecture. This disadvantage resulted in confusion and higher maintenance costs (bug fixes and updates had to be applied to all of the versions, instead of to just one version). Other disadvantages of QMotor 2.0 were limited logging capabilities (*e.g.,* it was not possible to set multiple logging modes, frequencies, and durations), limited plotting capabilities (*e.g.,* dynamic autoscaling and multiple plot windows were not available), and the lack of online parameter tuning.

To overcome the disadvantages of QMotor 2.0 and extend the capabilities of the QMotor framework for more complicated control applications, we have developed QMotor 3.0 and the QMotor RTK. We decided to extend the QMotor 3.0 framework with a robotic toolkit because of the complexity normally associated with a robotic application; that is, manipulator control systems contain not only the servo control implementation, but also trajectory generation, programming interfaces, and a user interface. Although many robot control languages have been created for the purpose of controlling manipulators, they are usually provided by the manipulator vendor and custom-tailored to the specific manipulator type. Since many of these robot control languages are not very flexible (*e.g.*, with regard to interfacing to new system components such as sensors and visual feedback), previous research focused on building robot control libraries on top of a commonly used programming language (*e.g.,* "C"). RCCL [7] and ARCL [8] are examples of such libraries. However, there is no straightforward way to modify the servo control algorithm in RCCL and ARCL (*e.g.*, for Puma 560 robots, the servo control runs on a proprietary Mark II controller); therefore, one cannot implement new control strategies [9]. 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.

The robot control platforms described above have another common problem: If new functionality is needed or if new hardware is required, one must modify the source code. Modification of the internals of a complex robot control system is very error-prone. To overcome this problem, object-oriented approaches have been used with regard to robot control libraries. As an example of object oriented design, RIPE [10], developed at Sandia National Laboratories, defines an intuitive hierarchy of classes for robotic hardware. However, RIPE does not use object-oriented concepts at the servo level. MMROC+ [11] uses an object-oriented design only for error handling and process communication. OSCAR [12] is an extensive library that addresses many issues of object-oriented design for robotic systems. Specifically, OSCAR focuses mainly on the operational software layer (the layer between the user interface and the servo control). OSCAR is complex and requires multiple computing platforms. Zero++ [13] is a multiprocessor-net system that uses object-oriented concepts mostly for the programming interface. To summarize, none of the above-described robot control libraries use an entirely object-oriented design (*e.g.*, the servo control is not included in the object-oriented design) or provide functionality for data logging, data plotting, and control parameter tuning. Additionally, many of the past libraries require multiple computing platforms (*i.e.*, special controllers, DSPs or PCs running different Operating Systems) and/or proprietary hardware. As opposed to past systems, the RTK implements a homogeneous object-oriented design on a single PC and includes data logging, data plotting, and control parameter tuning. Specifically with regard to controls and QMotor, the features of Object Oriented design most heavily exploited here are *data hiding* (so that the user is able to implement certain functions without harming the control execution framework), and *inheritance* (so that parts of existing code can be re-used). These are demonstrated in the QMotor RTK section. The primary focus of this research has been to develop a real-time control platform using a single hardware and software environment (PC running QNX OS). QMotor was designed for simplicity and ease of use, in comparison to current research efforts such as the Open Control Platform (OCP) [14]. A user can write working QMotor control programs in less than two hours, and the GUI is intuitive enough to get familiarized with in less than ten minutes.

QMotor 3.0

Overview

QMotor 3.0 runs on standard Intel PCs, a cost-effective and widely supported hardware platform. A single I/O board or multiple I/O boards provide the interface to the hardware. As with QMotor 2.0, the QNX real-time operating system [15] was selected as the software platform, as it provides all of the real-time functionality needed for the system and has proven to be robust and reliable. Although the system was developed under QNX 4, it was later ported to the QNX Real-Time Platform (RTP). The QNX RTP has the advantage that it is free for non commercial use. Additionally, it provides higher compatibility with UNIX systems than QNX 4 (*e.g.*, it provides a POSIX-compliant interface and familiar development tools from the Linux OS, such as the GNU compiler). QMotor 3.0 consists of three main parts (see Figure 1):

- **The hardware client/server architecture,**
- **The control program library,**
- **The QMotor GUI.**

In the following sections, these components are explained in detail.

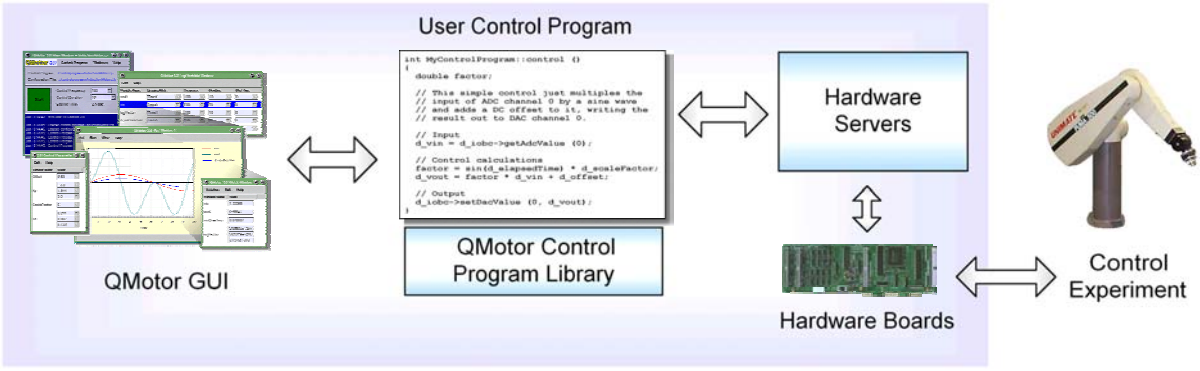

Figure 1. Overview of the QMotor 3.0 architecture.

Hardware Client/Server Architecture

To control a physical system, a computer control program must be able to interact with it. Information about a system is determined through the use of sensors, which measure and report signals of the system (*e.g.,* temperature, force, voltage). Actuators (*e.g.,* motors, electromagnets) are used to change the state of the system. Both sensors and actuators utilize some sort of interface hardware such as ISA or PCI I/O boards. The software to operate these types of I/O boards has been traditionally called a device driver. Traditional device drivers (as in UNIX and Microsoft Windows NT) generally reside in an operating system's kernel and, as such, are difficult to write and maintain. In addition, accessing a kernel-mode device driver from a user-mode program requires a system call, which incurs overhead (see Figure 2).

Consequently, many device manufacturers provide hardware interfacing libraries that are linked to the user's control program. This method is simpler than writing a kernel-mode device driver. It is also more efficient, since there is no need for a system call into the kernel. However, this method is also less secure. The device interface library must access hardware directly; therefore, the user's control program must have privileged access (*i.e.,* it must be run as *root*) and, hence, is capable of crashing or corrupting the entire system. Additionally, multiple programs may interfere while simultaneously attempting to communicate with the hardware board. To avoid this problem, only one control program may access the hardware at a time.

The architecture of the QNX OS allows for an approach that overcomes the above-mentioned problems. Specifically, since QNX is a microkernel-based OS, it does not provide for kernel-mode device drivers. The QNX microkernel provides only minimal functionality (scheduling, message passing, *etc.*). Programs that serve the purpose of device drivers run in user mode. For a user program (*e.g.*, a control program) to communicate with the device driver, interprocess communication (IPC) is used. The IPC is implemented by using shared memory and/or QNX message passing. Because a device driver serves the requests from one or multiple user programs, they are typically called

hardware servers. The user programs that use the server to communicate with the hardware are called *clients*. The *client/server architecture* is illustrated in Figure 3.

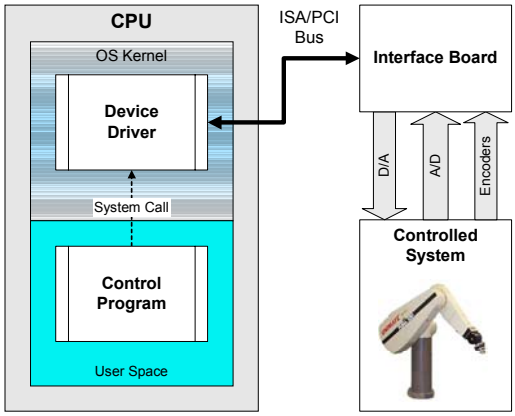

Figure 2. Traditional kernel mode device driver architecture.

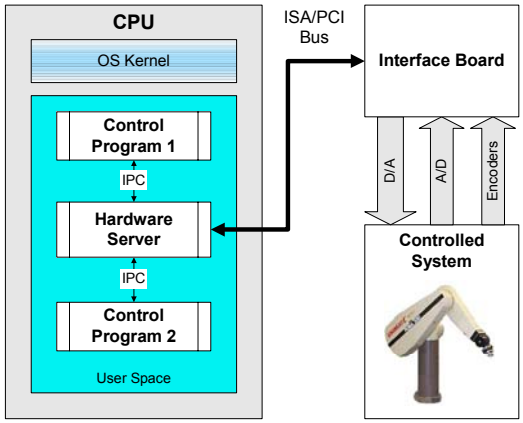

Figure 3. Hardware client/server architecture.

The advantages of using the client/server architecture as opposed to the traditional device driver architecture are:

- ***Easier configuration:*** Servers can be started and stopped at any time.
- ***Easier development:*** The hardware server is less complex than a device driver and can be developed and debugged in user mode.

- ***Performance:*** When using shared memory as the IPC mechanism, the only communication overhead is context switches between the client process and the server process.
- ***Networking:*** QNX allows message passing to work over a network without any changes in the software. Hence, the client program can be located on a different machine from the server program, provided that the network is deterministic and fast enough for the transfer of I/O data.
- ***Hardware sharing:*** Multiple clients can connect to the same hardware server and thereby share the same hardware without interfering with each other.
- ***Using generic clients:*** A control program can use generic clients that are independent of the specific hardware board/hardware server. This advantage is explained further in the next section.

I/O board servers are the most frequently used servers. All I/O board servers cycle continuously in a loop, reading analog-to-digital (A/D) channels, encoder channels, and digital inputs from the I/O board and writing digital-to-analog (D/A) values and digital outputs to the I/O board. all I/O board servers have common functionality which is reused by means of object-oriented techniques [16]. Specifically, a C++ base class `IOBoardServer` is designed to perform IPC with the client via shared memory. Then, the specific server classes (*e.g.*, `MultiQServer`, `STGServer`) are derived from the class `IOBoardServer` by adding the code required for operation of the specific hardware board. Using a common base class not only reduces redundancy, but it also allows for generic clients. Since the communication with the client is performed in the base class, it is independent from the specific I/O board. Hence, the same generic client can be used with a variety of different I/O boards. Because hardware servers run as separate programs, a change in I/O boards only requires starting up a different server program. Client programs (*e.g.*, QMotor 3.0 control programs) use the class `IOBoardClient`, which provides a simple interface for the IPC with the I/O board server. Figure 4 illustrates the client/server architecture for the ServoToGo board. Table 1 lists all I/O board servers available for QMotor 3.0.

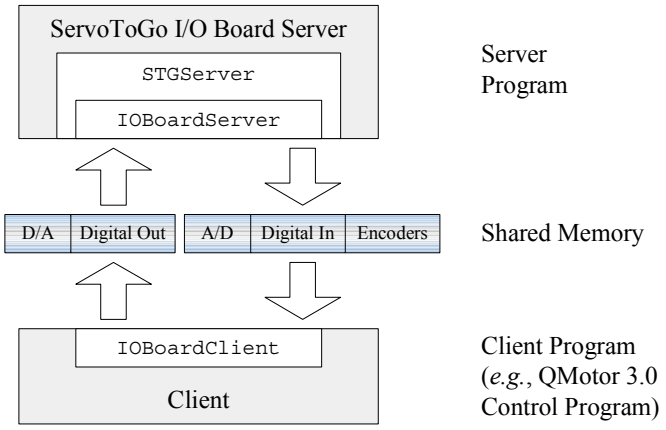

Figure 4. The client/server architecture for I/O boards.

Table 1. I/O Boards

Hardware Board	A/D	D/A	Encoders	Digital I/O	Timers	Special Features
Quanser MultiQ ½	8	8	6	16/8	3	-
Quanser MultiQ 3	8	8	8	8	3	-
ServoToGo S8 ½	8	8	8	32	1	Watchdog Timer
ComputerBoards CBDIO24/CTR3	-	-	-	24	1	-
ComputerBoards PCI-DAS1602/16	16	2	-	24	3	200-kHz Sampling Frequency

Timer servers provide an accurate periodic clock signal to one or more timer clients (*e.g.,* QMotor 3.0 control programs) by sending QNX proxy messages to the timer clients. The client frequency must be an integer divisor of the timer server's clock source. Similar to the I/O board servers, a base class `TimerServer` contains the common functionality for all timer servers; that is, the base class manages a list of clients and periodically sends proxy messages to them. The derived classes implement the clock source, which may be a hardware source such as the ServoToGo S8 board's timer circuitry or a software source such as a QNX timer. (Note: Software timers should only be used for testing purposes, since they do not provide 100% reliable timing.) A timer server is available for the MultiQ board, the

ServoToGo board, and the CBDIO24/CTR3 board. Timer servers run at the highest priority in the system to ensure that they cannot be delayed by other processes.

The class `TimerClient` provides a simple interface to communicate with the timer server. It allows the user's program to execute at a certain frequency and also detects if the computation is too slow to complete in one timer period. Similar to the I/O board client, the timer client is a generic client; hence, timer client programs do not need to be changed or recompiled when switching the timer server. A timer client is part of the QMotor 3.0 framework to provide the timing for the control program. Figure 5 illustrates the timer client/server architecture for the MultiQ I/O board.

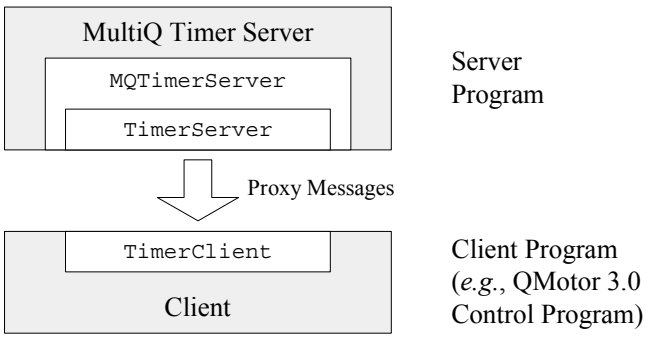

Figure 5. The client/server architecture for timer boards

The Control Program

The class `ControlProgram` provides a framework for developing control programs. All details of the control program execution (*e.g.,* creating a real-time control loop, logging variables, changing control parameters) are handled by this class. To implement a specific control application, the user derives a class from the `ControlProgram` class (*e.g.*, the class `ManipulatorControl`) and fills in the necessary functionality to implement the control algorithm. This functionality is contained in six virtual functions that are left blank in the base class `ControlProgram` (see See Table 2). The use of virtual functions allows a derived class to reimplement their functionality; thus, even if such a function is called from the base class, the reimplemented function will be used.

Table 2. The Main Functions of a QMotor 3.0 Control Program

`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 every time 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.

An example QMotor 3.0 control program for a proportional-derivative (PD) controller is presented below. A QMotor control program usually starts with the declaration of a new class that implements the specific control application (here, the class `PDControl`). This new class is derived from the class `ControlProgram`. In the class declaration, all control variables are listed. Additionally, an object to operate the I/O board client and a low-pass filter object are declared.

```
class PDControl : public ControlProgram
{
 protected:
    double q;         // Current Position
    double qd;        // Desired Position
    double error;     // Current Position Error
    double errorDot;  // Derivative of Current Position Error
    double torque;    // Output Torque

    double kp;        // Proportional Control Gain
    double kd;        // Derivative Control Gain
    double amplitude; // Amplitude of Desired Sine Trajectory
    double frequency; // Frequency of Desired Sine Trajectory

    double errorPrevious;      // Error of last control cycle
    ButterworthFilter<double> filter;  // Filter for backwards difference

    // ----- Clients -----
    IOBoardClient *iobc;      // To operate the I/O board

    (...)
};
```

The function `enterControl()` is used to register *log variables* and *control parameters*. The values of log variables are automatically logged to a buffer, which can be plotted and exported from the GUI.

Control parameters are used for tuning the control from the GUI (*i.e.,* the user can change the values of the control parameters from the GUI). By registering these variables, the QMotor 3.0 framework learns which C++ variables are to be transferred from/to the GUI.

```
int PDControl::enterControl()
{
  registerLogVariable(&q, "q", "Current Position");
  registerLogVariable(&qd, "qd", "Desired Position");
  registerLogVariable(&error, "error", "Position Error");
  registerLogVariable(&errorDot, "errorDot", "Position Error Derivative");
  registerLogVariable(&torque, "torque", "Control Torque");

  registerControlParameter(&kp, "kp", "Proportional Gain");
  registerControlParameter(&kp, "kd", "Derivative Gain");
  registerControlParameter(&amplitude, "amplitude",
                           "Amplitude of Desired Sine Trajectory");
  registerControlParameter(&frequency, "frequency",
                           "Frequency of Desired Sine Trajectory");

  return 0;
}
```

The function `startControl()` creates the I/O board client, which is called `iobc`. The parameter `qrts/iobs0` of the `IOBoardClient` constructor selects the desired I/O board server by specifying its name. Additionally, a filter for the backwards difference calculation is initialized in this function.

```
int PDControl::startControl()
{
   // Create the I/O board client
  iobc = new IOBoardClient("qrts/iobs0");

  // Initialize the filter
  filter.setCutOffFrequency(100);
  filter.setSamplingTime(d_controlPeriod);
  filter.setAutoInit();

  return 0;
}
```

Finally, the function `control()` implements the PD control algorithm. It is continuously called by the framework at the control frequency. The following code snippet also demonstrates the communication with the I/O board server via the use of the `IOBoardClient` object `iobc`:

```
int PDControl::control()
{
  // Get the current analog input from A/D Channel 0
  q = iobc->getAdcValue(0);

  // Calculate the desired trajectory
  qd = sin(frequency * d_elapsedTime) * amplitude;

  // Calculate the error and the derivative of error (backwards difference)
  error = q - qd;
  errorDot = filter.filter((error - errorPrevious) / d_controlPeriod);
  errorPrevious = error;

  // Calculate PD control
  torque = kp * error + kd * errorDot;

  // Set the current analog output of D/A channel 0
  iobc->setDacValue(0, torque);

  return 0;
}
```

The function `stopControl()` ensures that no voltage is sent to the D/A channel by setting it to zero. It then deletes the `IOBoardClient` object `iobc` to disconnect from the I/O board server.

```
int PDControl::stopControl()
{
  // Zero out the DAC
  iobc->setDacValue(0, 0);

  // Disconnect from I/O board server
  delete iobc;

  return 0;
}
```

Following development of a control program, the code is compiled and linked to the control program library. To start a control program, the user has two options: i) run it from the command line in stand-alone mode, or ii) run it from the QMotor GUI. The latter option is explained in the following section.

QMotor GUI

The QMotor GUI is built for the QNX Photon MicroGUI graphical environment, and allows the user to interact with the control program. It is used to start and stop a control program, tune control gains online, and view/plot data in real time.

From the main window, the user can load a control program, set the control duration and control frequency, and execute the control program. In addition, the main window allows the user to open the following subwindows: i) the *log variable window*, ii) the *control parameter window*, iii) the *watch window*, and iv) numerous *real-time plot windows*. The main window is shown in Figure 6.

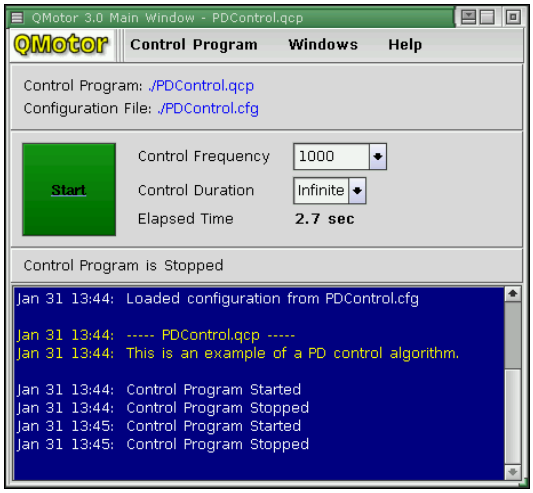

Figure 6. Main window.

The log variable window (Figure 7) displays a list of all available variables that have been registered for data logging in the control program. For each log variable, the user can specify the logging mode, frequency, start time, and duration.

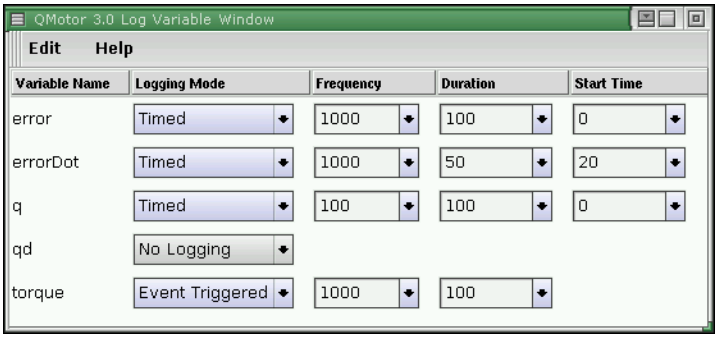

Figure 7. Log variable window.

The control parameter window (Figure 8) displays a list of all variables that have been registered as control parameters in the C++ control program. From this window, the control parameters can be

adjusted to the desired values without recompiling the C++ control program code. Control parameter values can be modified while the control program is running (*i.e.,* online parameter tuning is provided).

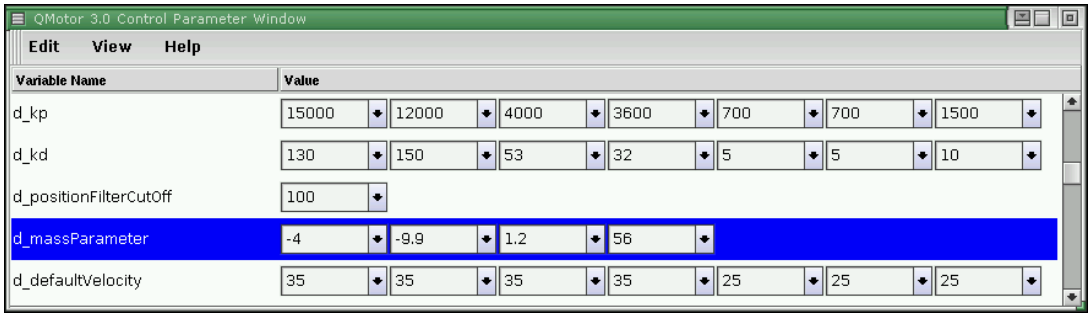

Figure 8. Control parameter window

The watch window (Figure 9) allows the user to see the real-time values of selected log variables during control execution.

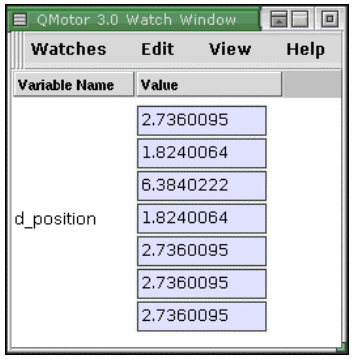

Figure 9. Watch window

The QMotor 3.0 GUI allows the user to monitor logged variables during control execution in the form of numerous real-time plot windows (Figure 10). All variables selected for logging are available for plotting purposes. Any number of plot windows may be open at once, and any number of variables may be plotted in each window. Numerous autoscaling options are available. The plot windows provide flexible plotting options, including an export function to MATLAB.

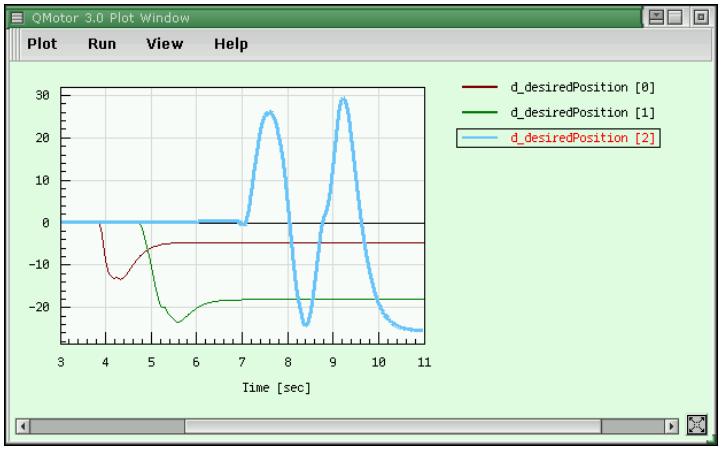

Figure 10. Plot windows.

QMotor RTK

The QMotor RTK toolkit is specifically designed for the control of robot manipulators. The RTK is structured as a combination of ready-to-execute programs and C++ libraries. Since the RTK is built on top of QMotor 3.0, its main components are QMotor 3.0 control programs. Consequently, the user has the capability to log and plot control signals and tune the robot controller. The RTK is a modular and extensible robot control platform; hence, it is a good demonstration of the versatility of QMotor 3.0. The QMotor RTK works only at the joint level (*i.e.*, forward/inverse kinematics and Cartesian trajectory generation are not currently included). It contains servo control programs for the WAM, the Puma 560, and the IMI manipulator. Also included is a generic joint-level trajectory generator and a GUI-based teachpendant. Additionally, various utility programs are part of the RTK. Figure 11 depicts 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 an RTK system, one only has to start different programs. For example, to replace the WAM with a Puma 560 robot, one would start the program `pumacontrol` instead of `wamcontrol`.

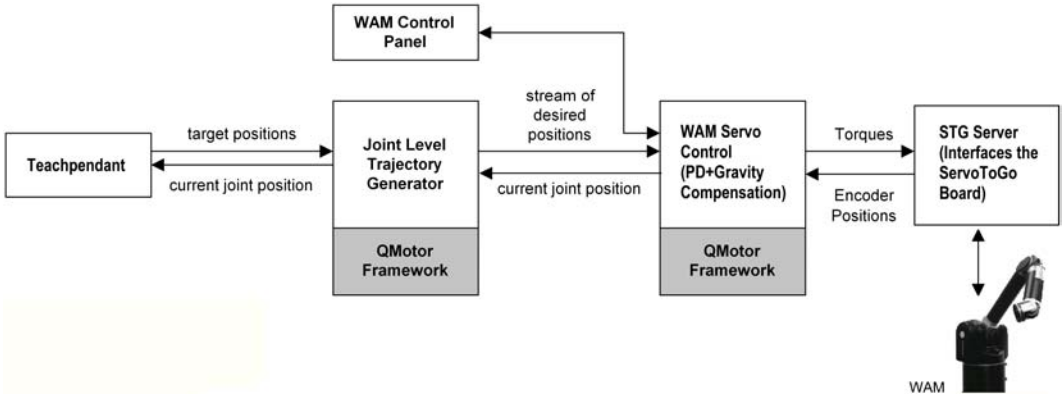

Figure 11. A typical QMotor RTK configuration.

Design Philosophies

In contrast to the procedural programming approach used for ARCL and RCCL, QMotor 3.0 and the RTK were both developed using an object-oriented approach. To highlight this difference in programming philosophy, we first note that 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, although procedural programming treats them separately, the object-oriented design ties them together; that is, they are grouped together in a construct called a *class*. The system can have any number of classes, 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, output torques) and all functions that are related to the control (*e.g.*, calculate the control algorithm, enable the arm power). To design an object-oriented system, the software engineer must carefully group data and functions into classes. With regard to a software platform for robotic applications, this choice is often intuitive; that is, classes represent physical objects (*e.g.*, manipulators), functional components (*e.g.*, the trajectory generator), and GUI

components. Consequently, the use of classes leads to a very intuitive modeling of the system. Several useful programming techniques are used in object-oriented programming: i) data abstraction, ii) encapsulation, iii) polymorphism, and iv) inheritance [16]. Among other benefits, these programming techniques have the following advantages:

- To use a class, an *object* of the class has to be instantiated. To operate multiple physical objects (*e.g.*, to control two manipulators of the same kind), the programmer simply instantiates multiple objects of the same class.
- *Polymorphism* is the ability to provide the same interface to objects related by inheritance, but differing in implementation. The technique, implemented using *virtual* functions, is useful for developing generic programs (*e.g.*, a trajectory generator can use the same generic interface for different manipulators). The correct implementation of the overridden function in the appropriate derived class associated with the object is chosen during execution of the program.
- 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.

Since inheritance is frequently used in the RTK, we will examine the concept of inheritance with regard to manipulator control software in detail. Once the software engineer starts to design classes for a manipulator control system, similarities between these classes become apparent. A class for the Puma 560 robot and a class for the WAM contain common functionality (*i.e.*, they both use a servo control algorithm, determine the current position by encoders, *etc.*). A simple approach for developing both classes would be to first construct 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 12a). 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 used. To use inheritance, a base class `ManipulatorControl` is defined. This base class contains the common functionality described above. Then, the specific classes for the Puma 560 and the WAM manipulator are *derived* from this base class (see Figure 12b). Deriving means that the classes take over the functionality and data from the base class. Additionally, they are able to reimplement parts of this functionality and/or add new functionality and data. Once the base classes have been developed, the source code of the base class does not need to be changed and recompiled 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 (after recompilation). Hence, inheritance greatly supports code reuse.

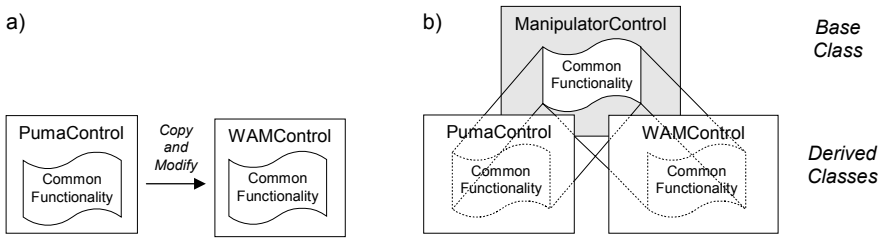

Figure 12. Code reuse through a) code duplication, and b) object-oriented programming.

For general-purpose applications, object-oriented programming has become more and more popular over the last two decades. In real-time systems, however, the use of object-oriented programming has caught on more slowly. To some degree, this is due to the belief that object-oriented languages are inefficient and that they have unpredictable temporal characteristics [17]. Neither of these concerns can be attributed specifically to object-oriented programming. 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 [18] for detailed information about C++ overhead).

The use of an object-oriented design is only the first step in supporting code reuse. Whether the code will be reused for many applications is highly dependent on its simplicity and its design. That is, 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 significant part of the software was often dedicated to establishing real-time and distributed computation using multiple processors, architectures, and operating systems. Such an architecture leads to large platforms that are more complex and heterogeneous. Furthermore, the technological progress in PC hardware and operating systems has made heterogeneous architectures superfluous for many applications. Hence, this article 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 real-time programming and communication overhead because these features are provided by QMotor 3.0 and the QNX OS.

Previous platforms also attempted to include a wide range of robotic functionality. This approach contributes to additional complexity as well, and it often fails to achieve the desired outcome. The spectrum of robotic research areas and applications is so broad that a robotic platform is never able to include all of them (*i.e.*, a specific application often requires modification of the platform when new functionality is required). We believe it is more beneficial for developers to be able to build on a lightweight and solid base of low-level functionality than to extend or modify a full-scale system. Hence, this research presents a bottom-up approach that starts by providing a flexible servo control level and then adds higher-level components (*i.e.*, 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. Components run independently from each other, separated by a clearly defined interface. Since researchers are often interested in just one special component of a robotic platform (*e.g.*, they are interested in improving the servo control algorithm), the RTK's modularity allows them to focus on their interest without learning the internals of the rest of the platform. Figure 13 shows some examples

of different configurations of the QMotor RTK (all RTK components are indicated by white boxes and components replaced by the user are indicated by gray boxes).

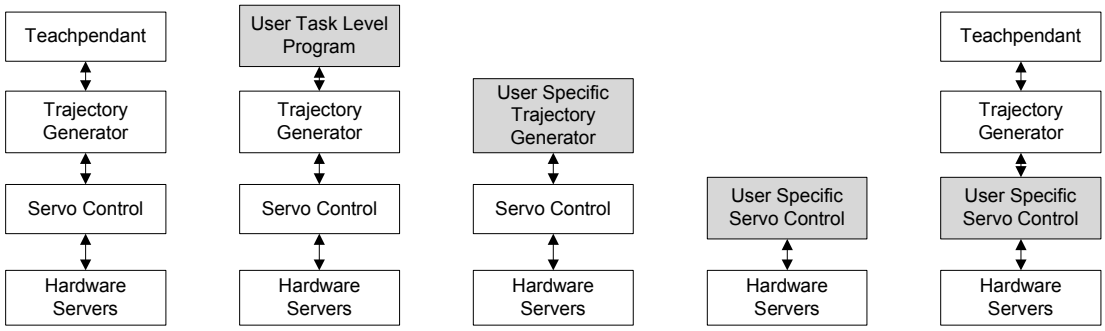

Figure 13. Example configurations of the QMotor RTK.

Manipulator Control Classes

The lowest level of the QMotor RTK is the servo control level, which consists of QMotor control programs for the Puma 560 robot, the WAM, and the IMI robot. These control programs implement an independent PD joint-tracking controller. 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 3 and 4.

Table 3. Common and Specific Data 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 4. Common and Specific Functionality for the Manipulator Control

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)

All common functionality (Table 4, left column) and data (Table 3, left column) are contained in the base class `ManipulatorControl`. This class, which is derived from the `ControlProgram` class, implements the functions `enterControl()`, `exitControl()`, `startControl()`, `stopControl()`, `control()`, and `handleMessage()` of the `ControlProgram` class. Figure 14 depicts the flowcharts of these functions. Note that the `handleMessage()` function is not shown in the flowcharts. All of the functions listed in the flowcharts (`control()`, `checkJointLimits()`, *etc.*) are virtual functions.

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, new functions are added and certain functions are reimplemented with modified functionality, as listed in Table 3 and 4(right column). Since the major part of the work is

done in the base class `ManipulatorControl`, the derived classes are significantly smaller and simpler. The following extensions are made in the `PumaControl` class:

- The automatic encoder calibration procedure is 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 reimplemented to take into account the coupling of joints 4, 5, and 6.
- Gravity compensation is added. Gravity compensation calculates the torques resulting from the manipulator's weight and adds these to the output torque for compensation [19].

Similarly, the class `WAMControl` contains some extensions to implement WAM-specific functionality:

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

The only reimplemented functions of the class `IMIControl` are the arm power functions. As the IMI does not have software control for arm power, the arm power functionality is removed in the derived class `IMIControl`.

Trajectory Generator

The trajectory generator is a separate QMotor control program that creates a stream of setpoints and forwards them to the manipulator control using QNX message passing. As the message protocol is generic, the trajectory generator can be used with any manipulator supported by the RTK. The trajectory generator operates at the joint level. It 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 in 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 path-blending algorithm is similar to that in [20].

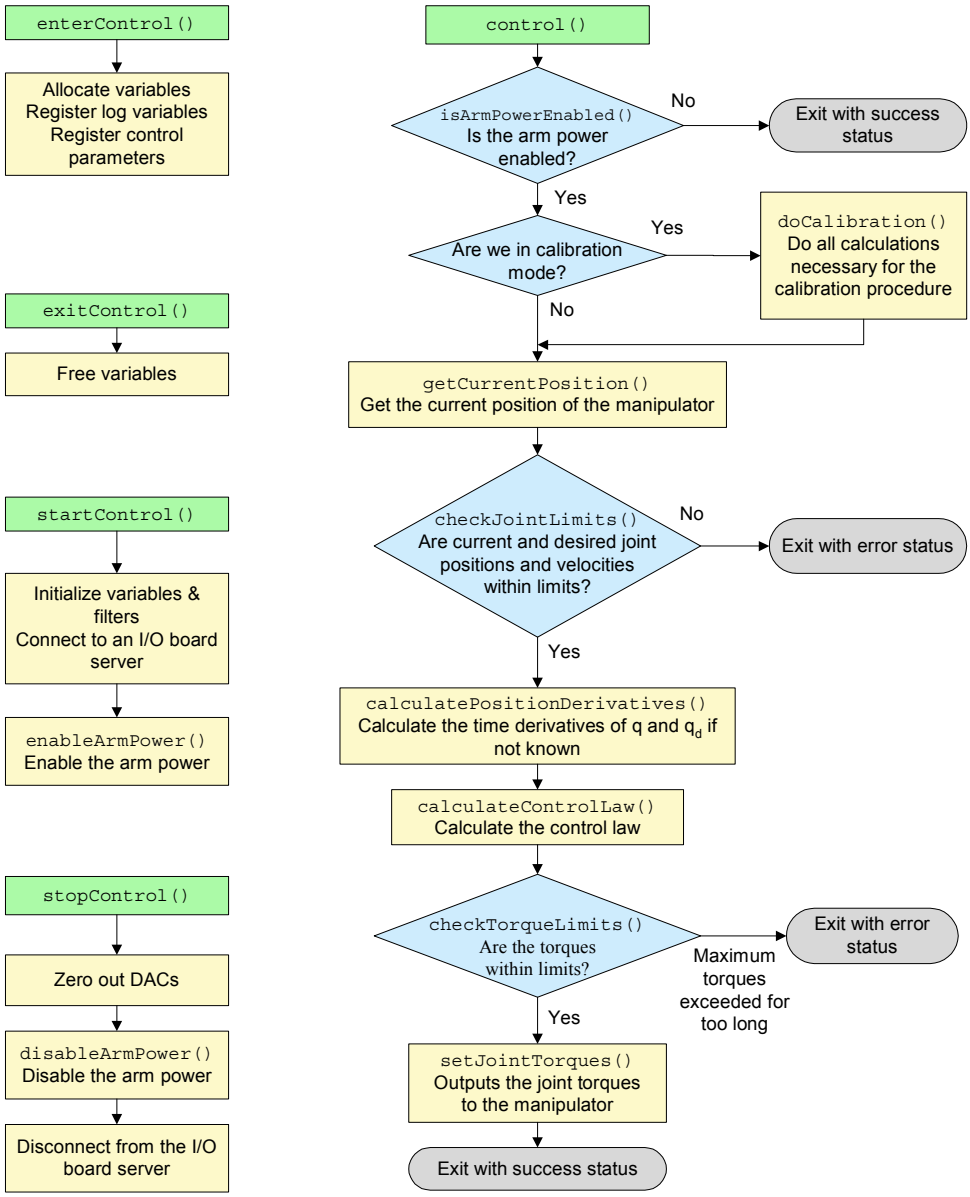

Figure 14. Flowchart of the functions enterControl(), exitControl(), startControl(), stopControl(), and control() in the class ManipulatorControl.

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. To use object-oriented techniques for the GUI, all GUI components are implemented with the C++ library QWidgets++ [21]. The RTK contains four GUI programs: the manipulator control panel, the WAM control panel, the manual-move utility, and the teachpendant. The manipulator control panel (see Figure 15) is a generic control panel that works with all manipulators. The WAM control panel (not shown) extends the manipulator control panel by adding buttons for enabling and disabling the torque ripple compensation.

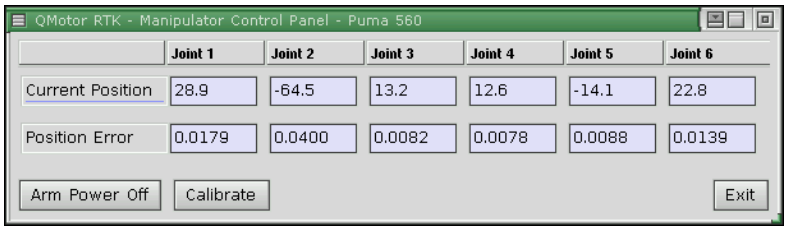

Figure 15. The generic manipulator control panel.

The manual-move utility (see Figure 16) 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.

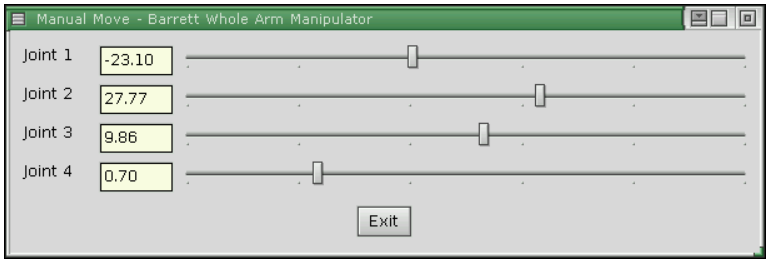

Figure 16. The manual-move utility.

The teachpendant (see Figure 17) 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 uses 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.

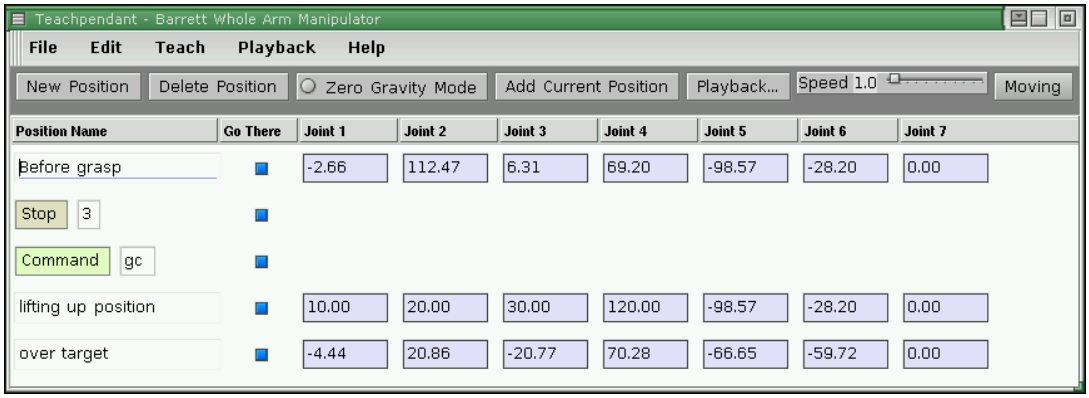

Figure 17. The teachpendant.

Extending the System Using Inheritance

The previous sections explained how object-oriented techniques accelerate the addition of new manipulator control programs 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 a PD controller to a proportional-integral-derivative (PID) controller.

Figure 18 shows the function `calculatePositionControl()`, which calculates the PD control in the base class `ManipulatorControl`. To implement the new controller, a new class `WAMPIDControl` is derived from the class `WAMControl` (see Figure 19, [a]). This class

reimplements the function `calculatePositionControl()`. The reimplemented function first calls the `calculatePositionControl()` function of the base class and, hence, uses the algorithm for the PD control of the base class (see Figure 19, [b]). Then the integral term is added (see Figure 19, [c]). Note that the function `calculatePositionControl()` of the base class and the derived class are distinguished by the scope prefixes `ManipulatorControl::` and `WAMPIDControl::`.

```
void ManipulatorControl::calculatePositionControl()
{
  // PD control
  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]);
  }
}
```

Figure 18. 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 19. The derived class WAMPIDControl.

Conclusions

This article documents the architecture of the control environment QMotor 3.0. Object-oriented techniques and client/server architectures were used to foster flexibility and extensibility. Support for several new hardware interface boards was added after QMotor 3.0 was completed, simply by providing new hardware servers that are based on the `IOBoardServer` class. QMotor 3.0 has been used by Clemson University and other research institutions to implement a wide variety of control algorithms, some of which are documented in [1], [2], [3], and [4].

QMotor 3.0 has also been used as the basis for a robot control system called the QMotor RTK, which also uses object-oriented techniques. The QMotor RTK was initially developed using Puma manipulators and was later extended to the Barrett WAM and the IMI Direct Drive robot. The QMotor RTK reuses code for implementing different manipulator control programs and GUI programs. Specifically, generic base classes and specific classes for the Puma 560 robot, the WAM, and the IMI robot have been developed. Figure 20 relates the code size of the generic and specific RTK components to the total code size, illustrating that the implementation of new manipulators requires a significantly smaller coding effort once the generic base class is implemented.

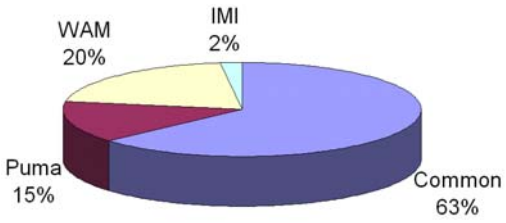

Figure 20. Code size ratios for the supported manipulators.

References

[1] B.T. Costic, S.P. Nagarkatti, D.M. Dawson, and M.S. de Queiroz, "Autobalancing DCAL controller for rotating unbalanced disk," *Proc. of the American Control Conference*, Chicago, IL, June 2000, pp. 2092-2096.

[2] M. Feemster, A. Behal, P. Aquino, and D.M. Dawson, "Tracking control of the induction motor in the presence of magnetic saturation effects," *Proc. of the IEEE Conference on Decision and Control*, Phoenix, AZ, Dec. 1999, pp. 341-346.

[3] W.E. Dixon, D.M. Dawson, E. Zergeroglu, and A. Behal, "Adaptive tracking control of a wheeled mobile robot via an uncalibrated camera system," *Proc. of the American Control Conference,* Chicago, IL, June 2000, pp. 1493-1497.

[4] E. Zergeroglu, D.M. Dawson, M.S. de Queiroz, and M. Krstic, "On global output feedback tracking control of robot manipulators," *Proc. of the IEEE Conference on Decision and Control*, Sydney, Australia, Dec. 2000, pp. 5073-5078.

[5] *BA4-310 Software User Manual*, Barrett Technologies, 139 Main St., Kendall Square, Cambridge, MA 02142, http://www.barretttechnology.com/robot.

[6] N. Costescu, D. Dawson, and M. Loffler, "QMotor 2.0 - A real-time PC-based control environment," *IEEE Control Systems Magazine*, June 1999, pp. 68-76

[7] 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.

[8] P. Corke and R. Kirkham, "The ARCL robot programming system", *Proc. Int. Conf. Robots for Competitive Industries*, Brisbane, Australia, pp. 484-493.

[9] N. Costescu, M. Loffler, E. Zergeroglu and D. M. Dawson, "QRobot - A multitasking PC-based robot control system," *Microcomputer Applications Journal Special Issue on Robotics*, vol. 18 no. 1, pp. 13-22.

[10] D.J. Miller and R.C. Lennox, "An object-oriented environment for robot system architectures," *IEEE Control Systems Magazine, February 1991*, pp. 14-23.

[11] C. Zieliński, "Object-oriented robot programming,", *Robotica*, vol. 15, pp. 41-48, 1997.

[12] Chetan Kapoor, "A reusable operational software architecture for advanced robotics," *Ph.D. thesis*, University of Texas at Austin, December 1996.

[13] C. Pelich and 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.

[14] L. Wills, S. Kannan, S. Sander, M. Guler, B. Heck, J.V.R. Prasad, D. Schrage and G. Vachtsevanos, "An open platform for reconfigurable control", *IEEE Control Systems Magazine*, June 2001, pp. 49-63.

[15] 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, [online] Available: http://qnx.com.

[16] B. Stroustrup, "What is 'object-oriented programming'," *Proc. 1st European Software Festival,* February 1991.

[17] T.E. Bihari and P. Gopinath, "Object-oriented real-time systems: Concepts and examples," *IEEE Computer*, December 1992, pp. 25-32.

[18] B. Stroustrup, "An overview of the C++ programming language," *Handbook of Object Technology*, CRC Press, Boca Raton, 1999.

[19] B. Armstrong, O. Khatib, and J. Burdick, "The explicit dynamic model and inertia parameters of the PUMA 560 Arm," *Proc. IEEE Int. Conf. Robotics and Automation 1*, 1986, pp. 510-518.

[20] Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*., Cambridge, MA: MIT Press, 1981.

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