Design and Implementation of the Robotic Platform

Markus S. Loffler, Vilas Chitrakaran, Darren M. Dawson

Department of Electrical and Computer Engineering, Clemson University

Corresponding Author:

Markus Loffler

Department of Electrical and Computer Engineering

Clemson University

Clemson, SC 29634

Phone: (864) 656-7218

Fax: (864) 656-7220

E-Mail: loffler@ces.clemson.edu

Abstract

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. To address previous disadvantages, this paper describes the design and implementation of the Robotic Platform, an object-oriented development platform for robotic applications. The Robotic Platform includes hardware interfacing, servo control, trajectory generation, 3D simulation, a graphical user interface, and a math library. As opposed to distributed solutions, the Robotic Platform implements all these components in a homogenous architecture that utilizes a single hardware platform (a standard PC), a single programming language (C++), and a single operating system (the QNX Real-Time Platform) while guaranteeing deterministic real-time performance. This design leads to an open architecture that is less complex, easier to use, and easier to extend. Particularly, the area of multiple cooperating robots benefits from this kind of architecture, since the Robotic Platform achieves a high integration of its components and provides a simple and flexible means of communication. The architecture of the Robotic Platform builds on the following state-of-the-art technologies and general purpose components to further increase simplicity and reliability: i) PC technology, ii) the QNX Real-Time Platform, iii) the Open Inventor library, iv) object-oriented design, and v) the QMotor control environment.

Keywords:

Robot Control, PC, Real-Time, Object-Oriented, QNX

Design and Implementation of the Robotic Platform

Markus S. Loffler, Vilas Chitrakaran, Darren M. Dawson

Department of Electrical and Computer Engineering, Clemson University

Abstract

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. To address previous disadvantages, this paper describes the design and implementation of the Robotic Platform, an object-oriented development platform for robotic applications. The Robotic Platform includes hardware interfacing, servo control, trajectory generation, 3D simulation, a graphical user interface, and a math library. As opposed to distributed solutions, the Robotic Platform implements all these components in a homogenous architecture that utilizes a single hardware platform (a standard PC), a single programming language (C++), and a single operating system (the QNX Real-Time Platform) while guaranteeing deterministic real-time performance. This design leads to an open architecture that is less complex, easier to use, and easier to extend. Particularly, the area of multiple cooperating robots benefits from this kind of architecture, since the Robotic Platform achieves a high integration of its components and provides a simple and flexible means of communication. The architecture of the Robotic Platform builds on the following state-of-the-art technologies and general purpose components to further increase simplicity and reliability: i) PC technology, ii) the QNX Real-Time Platform, iii) the Open Inventor library, iv) object-oriented design, and v) the QMotor control environment.

1 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 1). 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. 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 [1] and ARCL [2] 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 [3] and Chimera [4]). 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 [5] facilitated the management of more complex projects while also fostering code reuse and flexibility. For example, the robot control libraries RIPE [6], MMROC+ [7], OSCAR [8], and ZERO++ [9] utilized object-oriented techniques in robot programming. We have also witnessed the proliferation of real-time Unix-like operating systems for the PC [10], which facilitate the replacement of proprietary hardware components for real-time control [11]. 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) [12], for example, integrates real-time manipulator control and the graphical user interface (GUI) all on a single PC platform.

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 1. Building Blocks of a Modern Robot Control System

Despite the extensive functionality of the PC platform, much of the research in robot control software utilizes distributed and inhomogeneous architectures [6][8][9]. 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 [10]), and only one programming language (C++ [13]) 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.

2 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 [11]. 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 [10] consists of the QNX6/Neutrino operating system and additional components for development and multimedia. QNX6 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 [5, 12]. The language of choice is C++, as it provides the whole spectrum of object-oriented concepts while maintaining high performance [13].

Open Inventor. Open Inventor [14], 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 [15] is used for the Robotic Platform.

3 The Design and Implementation of the Robotic Platform

3.1 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 [1]). 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 2. 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.

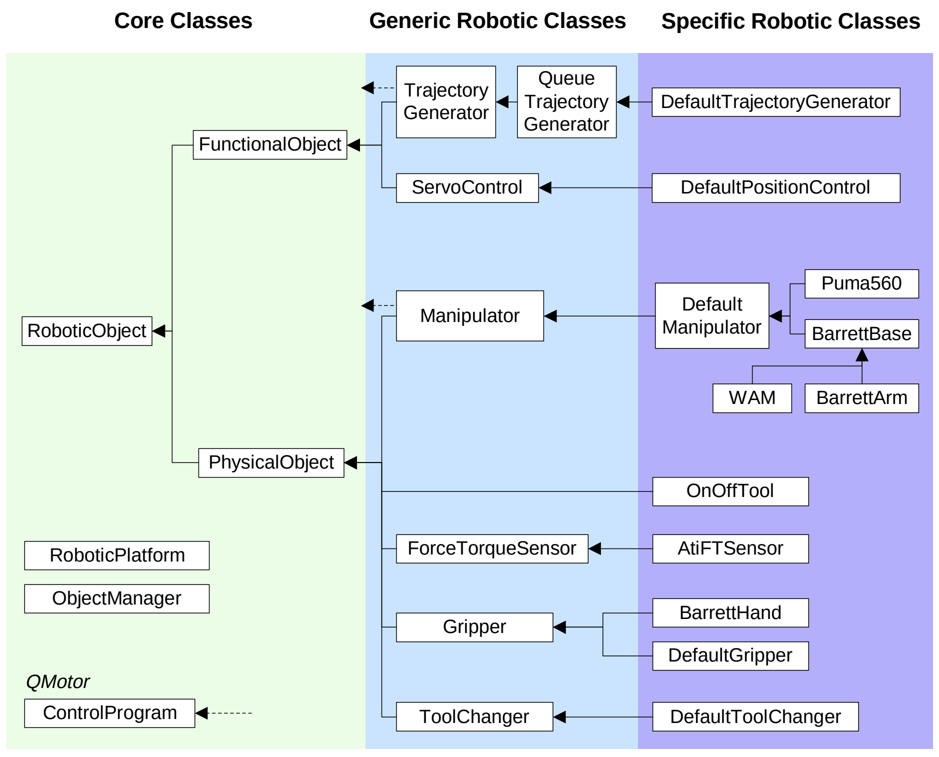


Figure 2. Class Hierarchy of the Robotic Platform

In addition to the classes shown in Figure 2, 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 paper.

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 3.

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 4.

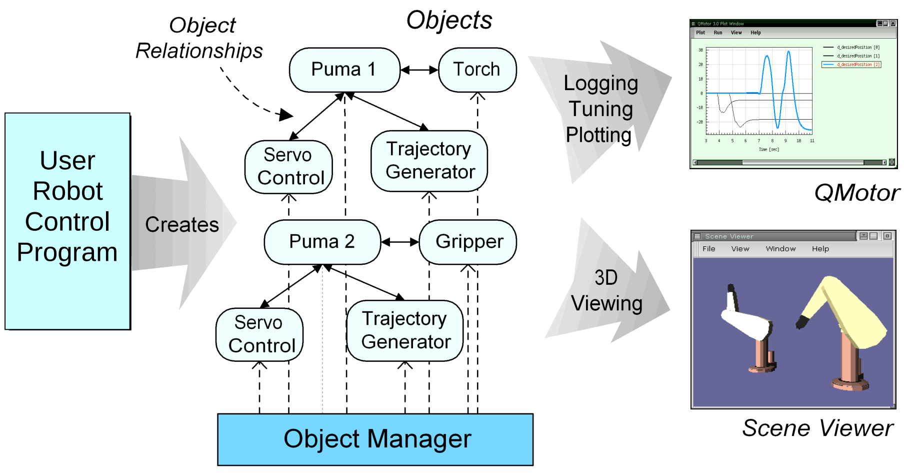


Figure 3. Run-Time Architecture of the Robotic Platform

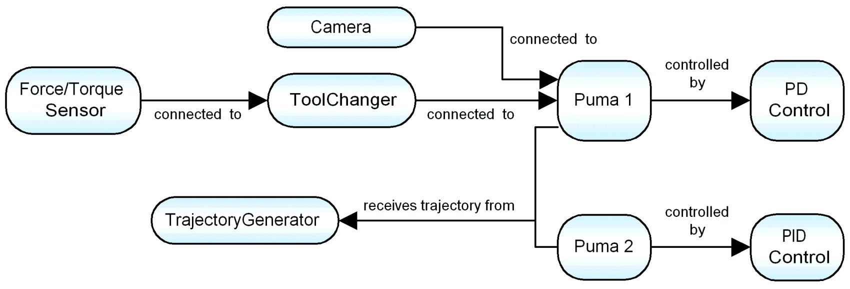


Figure 4. Object Relationships in an Example Scenario

3.2 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 5):

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.

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.

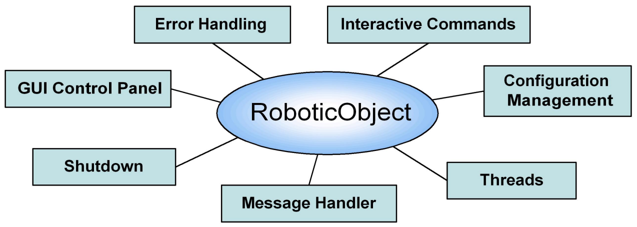


Figure 5. The Class RoboticObject

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 6. 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.

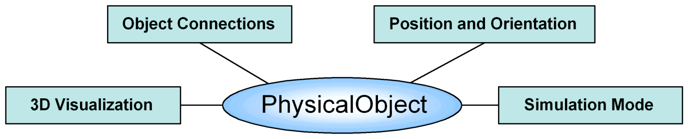


Figure 6. The Class PhysicalObject

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`.

3.3 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 [12].

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 7.

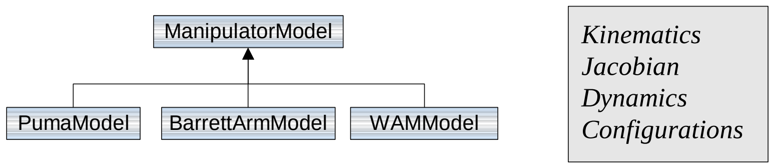


Figure 7. The ManipulatorModel Classes

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 8 illustrates this relationship in an example scenario.

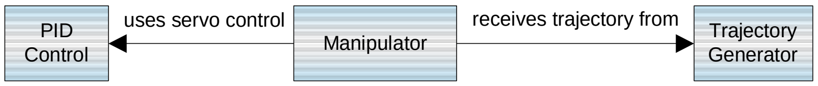


Figure 8. Object Setup for the Servo Control and the Trajectory Generation of a Manipulator

3.4 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 [16], 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.

3.5 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 9). Table 1 lists possible object settings and their related member functions defined by the classes `RoboticObject` and `PhysicalObject`. Derived classes are able to define additional settings.

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

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

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

Figure 9. An Example Global Configuration File

Name of Setting	Description	Equivalent Member Functions
`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	`PhysicalObject::setTransform()`
`simulationMode on` `simulationMode off`	If "`on`" is specified, use simulation mode for this object	`PhysicalObject::setSimulationModeOn()` `PhysicalObject::setSimulationModeOff()`
`display off` `display solid` `display wireframe`	Specifies if and how the object is displayed in the Scene Viewer	`PhysicalObject::setDisplayOff()` `PhysicalObject::setDisplaySolid()` `PhysicalObject::setDisplayWireframe()`

Table 1. Object Settings of the Configuration File

3.6 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.

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 10).

```
    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 9). 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.

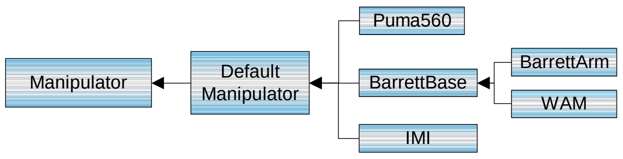


Figure 10. The Generic Class Manipulator and its Derived Classes

3.7 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 [12], 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 11 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).

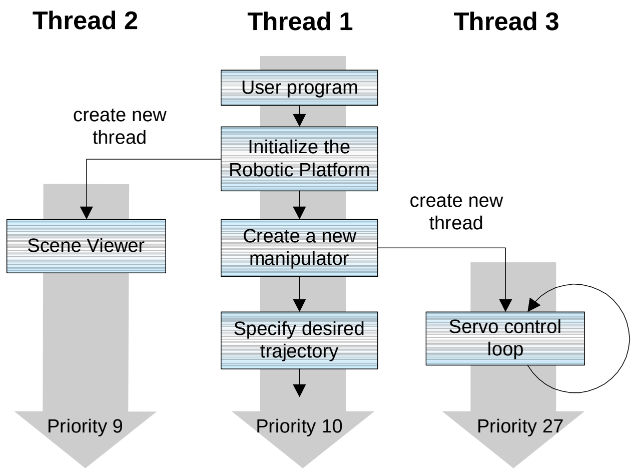


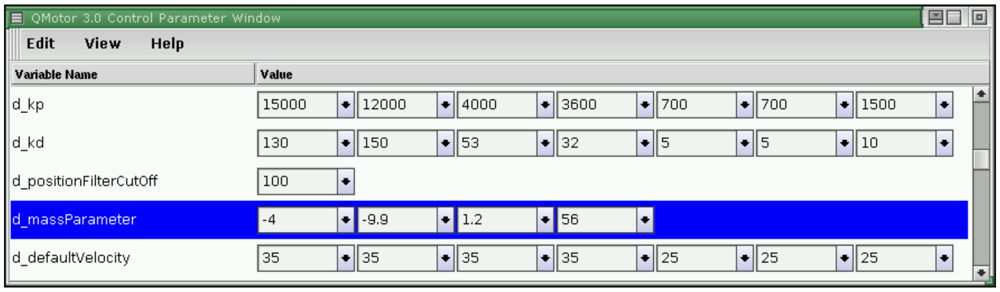
Figure 11. Creating New Threads for Concurrency

[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.

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`).

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

QMotor [15] 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 12). Furthermore, the user can set logging modes and display log variables in multiple plot windows (see Figure 13).



Variable Name	Value						
d_kp	15000	12000	4000	3600	700	700	1500
d_kd	130	150	53	32	5	5	10
d_positionFilterCutOff	100						
d_massParameter	-4	-9.9	1.2	56			
d_defaultVelocity	35	35	35	35	25	25	25

Figure 12. The QMotor Control Parameter Window

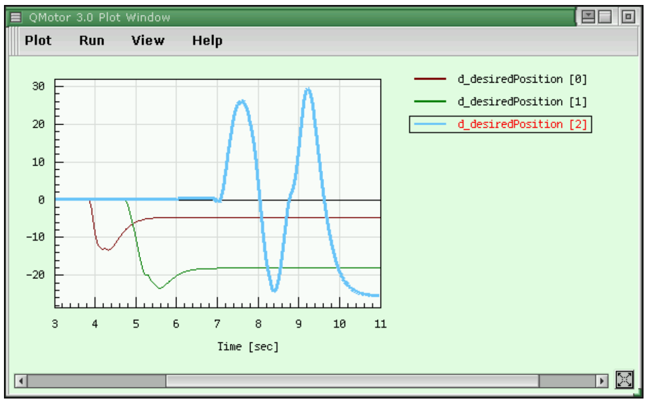


Figure 13. The QMotor Plot Window

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.

3.9 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 [12]. 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 14 and Table 2 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 15 shows an example program that uses the math library to calculate a position equation.

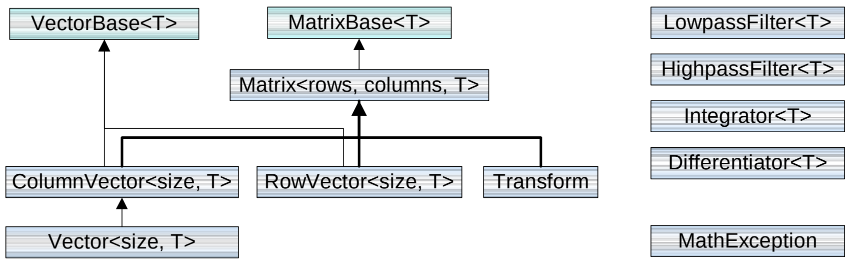
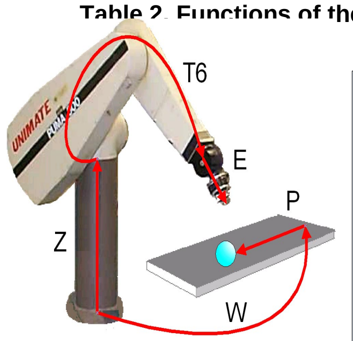


Figure 14. Class Hierarchy of the Math Library

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.

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

Table 2. 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 15. Example Program for the Math Library

3.10 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.

3.11 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 [17] 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).

3.12 The Graphical User Interface

GUI components are developed with the C++ GUI class library QWidgets++ [18]. QWidgets++ allows for object-oriented GUI programming. The GUI consists of three parts:

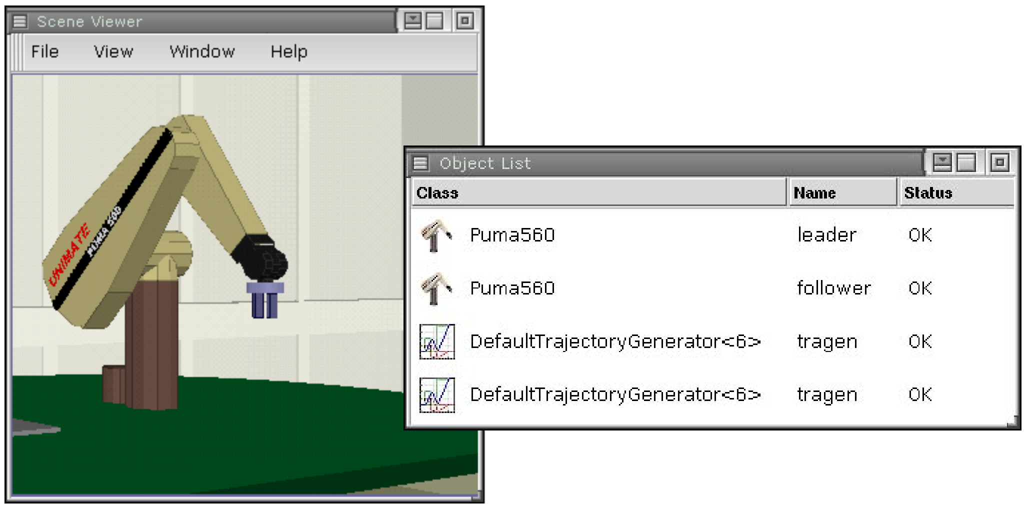
- The Scene Viewer is the default supervising GUI, which is opened automatically at startup of every Robotic Platform program.
- Additionally, each robotic class can have its own control panel. The control panels are opened from the Scene Viewer.
- Finally, the Robotic Platform includes several utility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

The GUI is further explained in the next section.

4 Using the Robotic Platform

4.1 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 16). 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.



Class	Name	Status
Puma560	leader	OK
Puma560	follower	OK
DefaultTrajectoryGenerator<6>	tragen	OK
DefaultTrajectoryGenerator<6>	tragen	OK

Figure 16. The Scene Viewer and the Object List Window

Each object has an individual pop-up menu (see Figure 17). 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 18 shows the control panel of the WAM as an example.

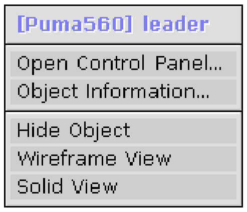
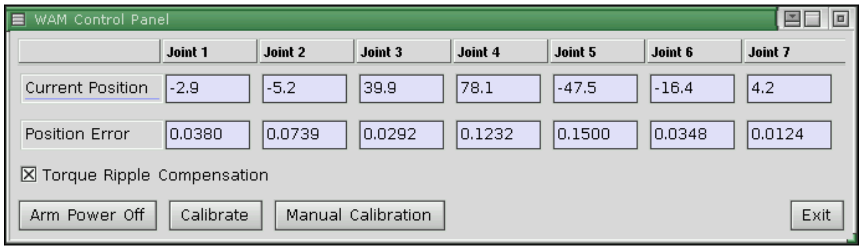


Figure 17. The Object Pop-Up Menu



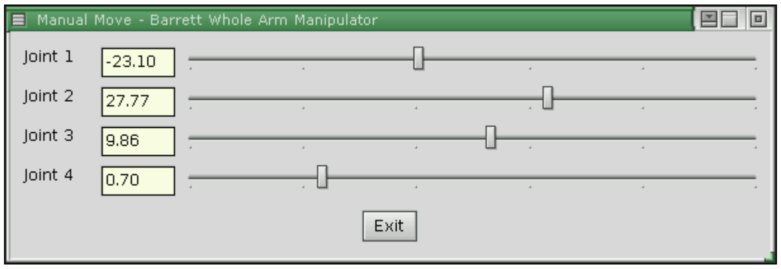
	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6	Joint 7
Current Position	-2.9	-5.2	39.9	78.1	-47.5	-16.4	4.2
Position Error	0.0380	0.0739	0.0292	0.1232	0.1500	0.0348	0.0124

☒ Torque Ripple Compensation

Figure 18. The Control Panel of the WAM Class

4.2 The Utility Programs

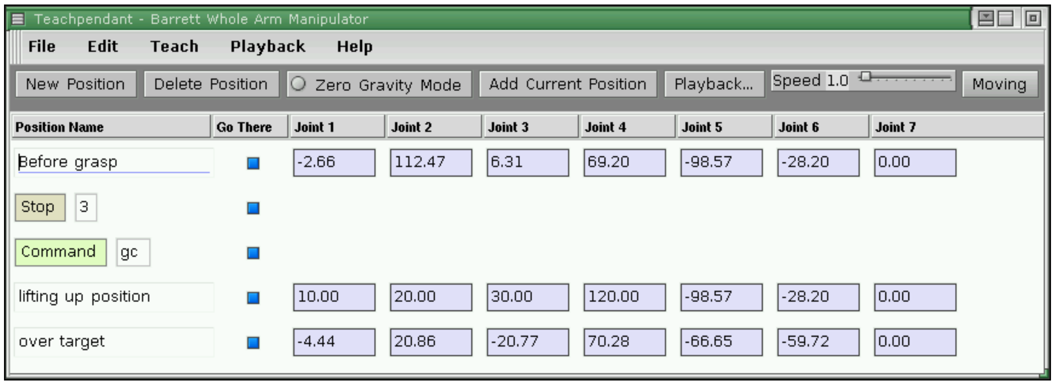
The Robotic Platform provides a couple of utility programs that help testing the system by performing simple operations. The ***Joint Move*** utility (see Figure 19) is a program to test the servo control of a manipulator. It contains a slider for each joint. The user can move the sliders with the mouse and the manipulator follows immediately.



Joint 1	-23.10
Joint 2	27.77
Joint 3	9.86
Joint 4	0.70

Figure 19. The Joint Move Utility

The ***Teachpendant*** (see Figure 20) 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.



Position Name	Go There	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6	Joint 7
Before grasp	🟦	-2.66	112.47	6.31	69.20	-98.57	-28.20	0.00
Stop 3	🟦							
Command gc	🟦							
lifting up position	🟦	10.00	20.00	30.00	120.00	-98.57	-28.20	0.00
over target	🟦	-4.44	20.86	-20.77	70.28	-66.65	-59.72	0.00

Figure 20. The Teachpendant

4.3 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 21 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 3). Additionally, these options can be controlled from C++ code as well, by using the functions listed in the third column of Table 3. 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.

Command Line Switch	Description	Equivalent in C++

`-sim`
`-nosim`
	Enables/disables simulation mode for all objects	`RoboticPlatform::setSimulationModeOn()`
		`RoboticPlatform::setSimulationModeOff()`

`-gui`
`-nogui`
	Enables/disables automatic start of the Scene Viewer	`RoboticPlatform::setGuiOn()`
		`RoboticPlatform::setGuiOff()`

`-config <filename>`	Specifies the name of the global configuration file	`RoboticPlatform::setConfigurationFileName()`
`-qmotor`	Starts up the QMotor GUI	-

Table 3. Default Command Line Options of Robotic Platform Programs

```
// 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 object
  DefaultTrajectoryGenerator<6> tragen;
  puma.setTrajectoryGenerator(tragen);

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

  // 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 21. A Simple Pick and Place Program for the Robotic Platform

5 Programming Examples

5.1 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 22). 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);

  Puma560 puma;    // Create the robot object

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

  puma.setGravityCompensationOn();                                 (3)
  puma.setServoControl(wallControl);
}
```

Figure 22. Virtual Walls Example

5.2 Comparison of Simulation and Implementation

A very interesting option is to forward the set points 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 23. 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 set points.

```
#include “RoboticPlatform.hpp”

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

  // Create the robot objects
  Puma560 puma;
  Puma560 pumaSimulated;

  // The second robot is simulated
  pumaSimulated.setSimulationModeOn();

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

  Vector<6> target;

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

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

Figure 23. Example Program that Forwards 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.

References

[1] 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.

[2] P. Corke and R. Kirkham, “The ARCL Robot Programming System”, Proc. Int. Conf. Robots for Competitive Industries, Brisbane, Australia, pp. 484-493.

[3] 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).

[4] 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.

[5] B. Stroustrup, “What is ‘Object-Oriented Programming’?“, Proc. 1st European Software Festival. February, 1991.

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

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

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

[9] 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.

[10] 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, Email: info@qnx.com, http://qnx.com.

[11] 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.

[12] 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, to appear.

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

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

[15] 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, to appear.

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

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

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