QMotor 3.0 - An Object Oriented System for PC Control Program Implementation and Tuning

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

Department of Electrical and Computer Engineering
Clemson University, Clemson, SC 29634-0915
nick@qrts.com,ddawson@ces.clemson.edu

Abstract

QMotor 3.0 provides a versatile framework for the implementation of advanced control algorithms as C++ programs. The QMotor 3.0 graphical user interface (GUI) integrates functionality for the testing and tuning of these control programs. In addition, it also provides advanced data logging, plotting, and data exporting capabilities. By implementing control programs on a real-time PC operating system (OS), QMotor 3.0 eliminates the need for DSP boards. QMotor 3.0's high performance and flexibility allow for the implementation of many different control applications ranging from simple PD control routines to complex, nonlinear, multidimensional control algorithms. The use of C++ for control programs allows for high execution speeds and the implementation of very complex control structures. QNX as the operating system gives high reliability with low overhead such that the control programs can run in an embedded environment. A client/sever architecture decouples the control program from the hardware so that QMotor 3.0 can easily be extended to work with new hardware.

1 Introduction

This paper describes QMotor 3.0, a QNX based object-oriented (OO) single-processor software environment that allows the implementation of real-time control programs on standard Intel processor based personal computers (PCs). The control program, as well as the development tools and graphical user interface (GUI) can all execute simultaneously on the PC due to the deterministic response of the OS. This architecture replaces the traditional multiprocessor Host/DSP board architecture used in control applications. Advantages of a single-processor system include reduced cost and complexity, as well as increased flexibility and upgradability. Since replacing QMotor 2.0, QMotor 3.0 has been used successfully in all of the control experiments performed by the Clemson Control and Robotics group, including motor and robot control, active magnetic bearing experiments, web handling, vibration control in flexible structures, *etc*. Some of these experiments are documented in [1], [2], [3] and [4]. The use of object oriented programming (OOP) techniques along with a client/server architecture allow QMotor 3.0 to be used with many types of hardware devices.

1.1 Previous Research

WinMotor, QMotor 1.0, and QMotor 2.0 are described in [5]. WinMotor and QMotor 1.0 are multi-processor heterogeneous PC Host/DSP single board computer (SBC) systems, where the control executes on a DSP SBC, while a Host PC is used for GUI (plotting), data logging, and gain tuning functions.

QMotor 2.0 is a single-processor system developed by the authors of this paper that executes both the control and the GUI on the same processor. Though [5] discusses QMotor 2.0 in detail, a brief description of the system will be given here in order to facilitate discussion of QMotor 3.0. QMotor 2.0 is based on procedural/functional programming techniques, while QMotor 3.0 is based on OOP techniques to overcome some of the disadvantages of QMotor 2.0. Figure 1 below depicts the QMotor 2.0 architecture.

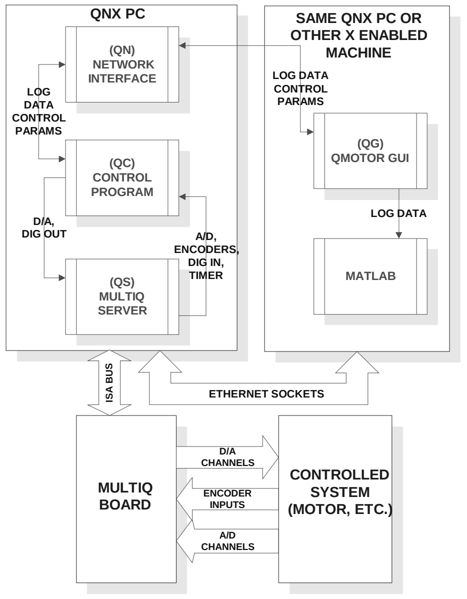


Figure 1 - QMotor 2.0 Architecture

The development of QMotor 2.0 was motivated by the high cost and complexity of multi-processor heterogenous (PC Host/DSP SBC) systems. The use of modern high-speed consumer grade PC CPUs coupled with a PC real-time OS allowed the development of a system that implements the control algorithm as well as the user interface on one CPU. This reduced the cost and complexity of the system, for developers as well as users.

The QMotor 2.0 architecture consists of the control program (QC), the hardware server for the MultiQ motion control board (QS), the network interface (QN), and the GUI (QG). These four programs all executed on the same processor and provided the framework needed to implement and tune control programs. The user fills out certain C language functions in QC (the control program) to implement the control computation. QMotor 2.0 was used to implement many published control experiments at Clemson University, some of which are documented in [6], [7], [8], and [9].

Several disadvantages of QMotor 2.0 became apparent with use. QMotor 2.0 only supported the MultiQ motion control board for hardware interfacing. Support of additional hardware had to be added later, and was hard-coded into the system. For example, a special version of QMotor 2.0 had to be built to support line-scan cameras for a mag-bearing experiment [10]. Another experiment required more than 8 D/A channels thus mandating the use of 2 MultiQ boards, and requiring a new version of QMotor 2.0. A robot control experiment that required special initialization of a motor drive via serial port required yet another modification of the QMotor 2.0 GUI (QG). All of these modifications were hard-coded into QMotor 2.0 programs, and produced multiple versions of the system, which 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). The functional programming techniques used to develop QMotor 2.0 did not allow the user to easily add support for new hardware interfaces (*e.g.* cameras, fast A/D boards, more I/O channels, *etc*.)

2 Object Oriented Programming

Development of QMotor 3.0 was motivated by the success of QMotor 2.0, and was shaped partially by the disadvantages of that system. QMotor 3.0 uses OOP techniques to overcome the disadvantages of QMotor 2.0.

2.1 Code Reuse

OOP techniques allow for code to be reused, which means only one copy of a certain piece of code needs to be maintained. The extensions to QMotor 2.0 mentioned above required three slightly different copies of the source code be maintained. Maintaining only one copy of source code allows bug fixes and improvements to be easily applied to that one copy, and all modules that use that code benefit.

2.2 Easy Extensibility

The use of inheritance and polymorphism allows the system to be easily extended. Functionality can be declared in a common base class, and implemented later in derived classes that are specific to a particular application.

2.3 Disadvantages

There are several popular criticisms of OOP. Two of these, which are applicable to this discussion, are excessive complexity and performance degradation. Excessive complexity can result from the overuse of classes, operator overloading, inheritance, *etc*. When OOP features are abused to an extent where they make the code more complex than it ought to be, or when programmers try to take advantage of obscure and confusing capabilities of the programming language, the resultant code becomes more confusing and complex than a procedural version.

Performance degradation is a serious concern in real-time applications. Some OOP language features depend on run-time processing. Other features simply add overhead to typical operations. Calling a method of an object, as opposed to simply calling a global function, incurs extra overhead of at least one extra jump (depending on the compiler). Assignments in C++ may cause a copy constructor to be called, requiring a function call, where a simple C assignment might translate to one assembly language instruction. The key to taking advantage of OOP functionality in real-time programming is to recognize the performance pitfalls, and avoid them. When implementing QMotor 3.0 in C++, we used the term “sane C++” to refer to features of the language that do not incur a significant runtime performance penalty, but contribute significantly to the readability, maintainability, extensibility, portability, and reusability of the code.

Another performance concern is raised when template functions and classes are used. Templates are instantiated by the compiler for each specific type used. While the programmer only maintains one copy of the template code, increasing the maintainability and extensibility of the code, the compiler replicates the code many times, increasing the size of the executable program. This causes more memory to be used during program execution, and can even impact performance by causing CPU cache misses simply because the executable can not be contained in the cache. While clever means could be used that would result in only one copy of the code, these means would bypass the advantages templates provide in terms of simplicity and type checking. For this reason, templates are used in QMotor 3.0, where appropriate.

2.4 OOP Techniques Used in QMotor 3.0

2.4.1 Inheritance

Inheritance is used to eliminate duplicate functionality in related objects. For example, sorting a list of variables alphabetically is done in the same way regardless of whether the list is a list of control parameters or log variables. Using an abstract data type (ADT) to represent a generic variable, and implementing the sorting functionality at that high level eliminates the need to implement separate methods for sorting control parameters and log variables.

2.4.2 Polymorphism

Polymorphism allows objects that are derived from a common ancestor to behave differently when the same method is called. For example, if the objects Rectangle and Triangle are derived from Shape, and Shape has a `calculateArea()` virtual function, Rectangle and Triangle can provide different results to the same function (`calculateArea()`). This mechanism provides a simple, common interface at the higher level, allowing more specific implementations at a lower level.

2.4.3 Templates

Templates allow code to be reused in a macro-like manner, while maintaining type checking. Templates are used when generic operations are to be applied to many different types of data, and the operations are unaffected by the data types.

3 QMotor 3.0 Architecture

Once again the QNX real-time operating system was chosen as it provides all of the real-time functionality needed for the system, and has proven to be robust and reliable.

3.1 Hardware Servers

In order 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 information about the system (*e.g.* temperature, force, voltage, current, *etc.*) Actuators (*e.g.* motors, electromagnets, *etc.*) are used to change the state of a system.

Sensors may be equipped with a computer interface that processes the sensor data and provides it to the computer in a simple form. An example of this type of sensor is a force/torque sensor that processes the raw strain gauge data, and presents a vector of 3 biased forces and 3 biased torques, in IEEE floating point format, in registers on an ISA bus controller card. Another example is a high-speed video camera coupled with a high-speed framegrabber. This sensor presents the computer with an array of digital values corresponding to the brightness of each pixel in the image. The computer may read the digital values of the pixels directly from the framegrabber's memory. This type of sensor reduces the computational burden on the computer by processing the sensor data, but require sophisticated software interfacing. Other sensors provide a simpler interface, generally converting some physical quantity (temperature, force, distance, *etc.*) into a voltage, or into a digital pulse train, which must then be read by a general purpose interface (*e.g.* an A/D converter, encoder input channel, *etc.*) This type of sensor interfaces to the computer via an A/D board, encoder interface board, *etc.* In both cases, sensors provide inputs through some interface hardware into the computer, for use as inputs to a control program. Actuators fall into similar categories. Simple actuators are much more prevalent (*e.g.* motors, solenoids, *etc.*), and are usually interfaced through a D/A or digital output board.

The software that allows a control program to communicate with the hardware that interfaces sensors and actuators to the computer has been traditionally called a device driver. Drivers generally reside in an operating system's kernel, as in UNIX and MS Windows NT, and as such are difficult to write and maintain. Writing a device driver for a sensor/actuator interface would generally be considered overkill. In addition, accessing a kernel-mode device driver from a user-mode program requires a system call, which can incur significant overhead. This is shown in Figure 2 below.

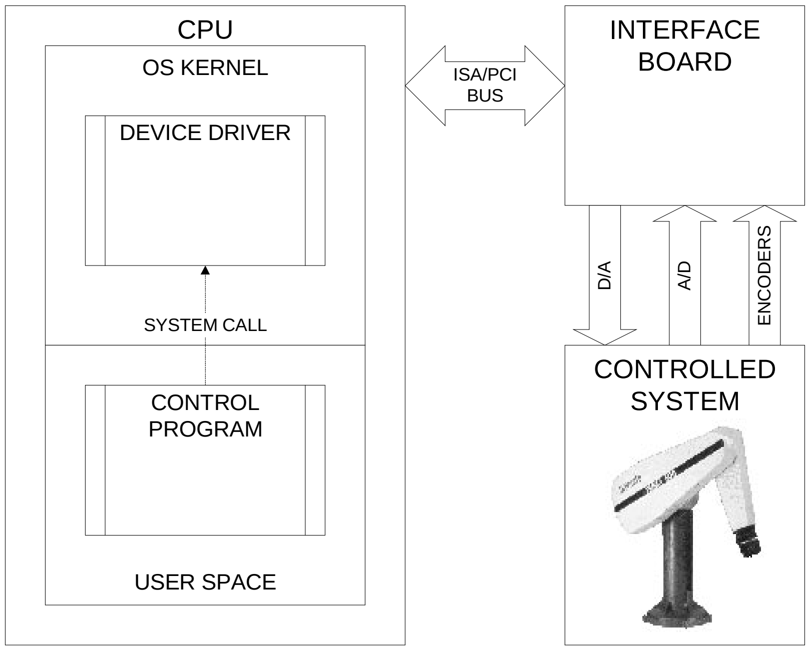


Figure 2 - Kernel Mode Device Driver

Consequently many device manufacturers provide libraries that are linked to the user's control program. These libraries provide access to the hardware interface. This method is far simpler than writing a kernel-mode device driver. It is also more efficient, as there is no need to trap into the kernel. It is also less secure. The device interface library must access hardware directly, therefore it must have privileged access (*e.g.* run as *root*). In addition, since it is linked to the user's control program, the user's control program must run in a privileged mode, and is capable of crashing or corrupting the entire system. Finally, since the control program is linked to the library, only one control program may execute at once, otherwise several programs may attempt to communicate with the hardware interface simultaneously. This architecture is shown in Figure 3 below.

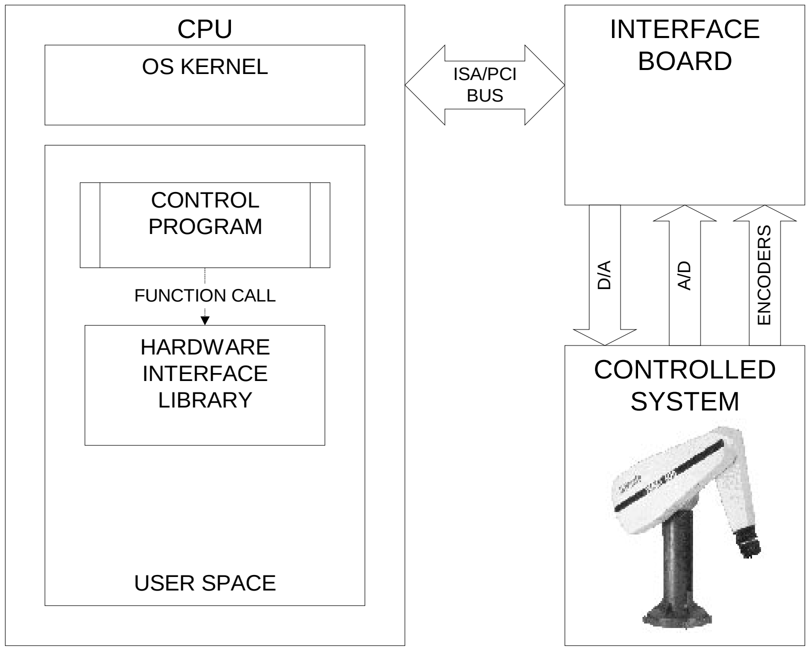


Figure 3 - User Mode Hardware Interface Library

QNX, being a microkernel based OS, does not provide for kernel-mode device drivers. The QNX microkernel provides only minimal functionality (scheduling, IPC, *etc.*) Programs that serve the purpose of device drivers run in user mode. Under QNX, these programs are better called "device servers." An advantage of running device drivers in user mode is that they can be started and stopped at any time, without affecting the kernel or other processes. Another advantage is that they provide services to processes located on other computers on the network, without the need for a network API. QNX uses message passing as its primary IPC mechanism. Message passing is supported inside the microkernel, and is network transparent (*i.e.* there is no difference in the API for sending a message to a process on the same node, or to a process on a different node on the network). Consequently, another advantage is that QNX device drivers can be accessed from any node on the QNX network with the same API as local device drivers (*e.g.* `open ("/dev/ser1","r")` opens the primary serial port on the local node, `open ("//10/dev/ser1","r")` opens the primary serial port on node 10). Traditional API calls such as *open()* and *write()* are translated into messages to the device drivers. While this does incur some overhead, the efficiency of the QNX microkernel makes it a viable implementation. Consequently, QMotor 3.0 uses a client/server architecture with respect to interface hardware, as shown in Figure 4.

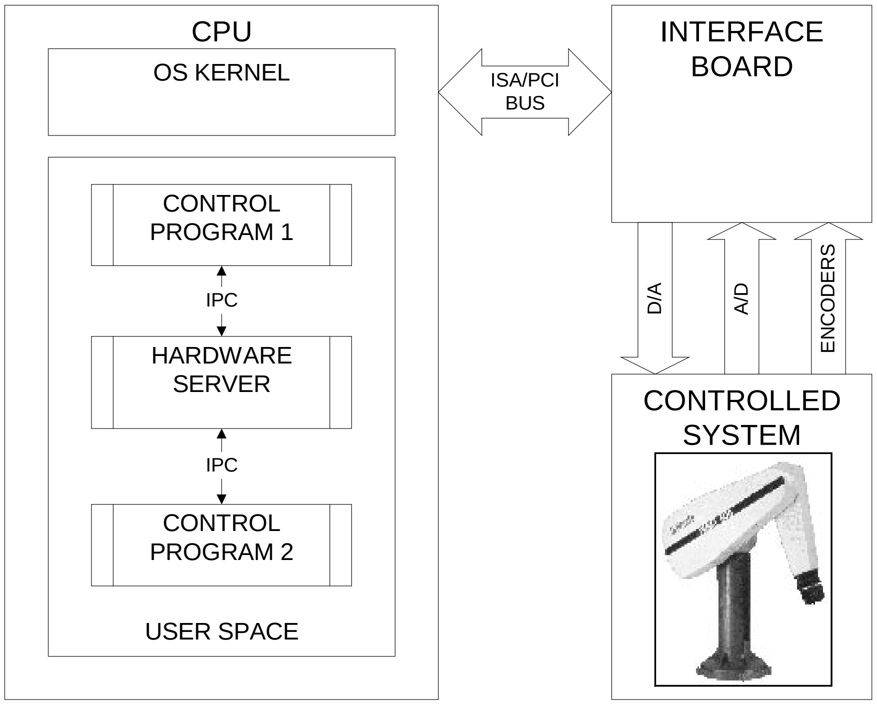


Figure 4 - Hardware Client/Server Architecture

The first step in writing a hardware server is to write the traditional library mentioned above, called a *hardware interface class*. This is a C++ class that provides all of the functionality needed to talk to the hardware (*e.g.* functions such as `inport()`, `outport()`, DMA transfers, interrupt service routines, *etc.* are implemented here). Users may choose to link directly to the hardware interface class, however, this is not recommended. Using the hardware interface class still requires much specific knowledge about the hardware. This class is intended only as a tool for those who will write hardware servers. Once a hardware interface class has been developed and tested for a certain hardware interface, a hardware server is written.

One of the advantages of OOP is code reuse. One way to reuse code is to identify data members and methods that are common to several objects, and design a superclass that contains those methods and members. The more specific objects are then derived from this superclass, and they provide the device specific members and methods.

Our experience has shown that most control experiments require only a few general types of hardware interfaces, listed below.

1. A/D Channels convert an analog voltage signal into a digital representation that can be read by a computer program.
2. Digital Input Channels can provide boolean inputs (on/off).
3. D/A Channels convert a number stored in the computer's memory to an analog voltage.
4. Digital Output Channels provide boolean outputs (*e.g.* toggle a relay, or some other on/off actuator).
5. Encoder Inputs are required for measuring the angular position of motors and other devices.
6. Hardware timer with ISA bus interrupt generation (for timing).

Items 1-5 give rise to the IOBoardServer abstract class, and item 6 yields the TimerServer abstract class. In OOP terminology, and abstract class (AC) is a superclass that contains only pure virtual methods, so that an abstract class can never be instantiated. Only classes derived from the abstract class may be instantiated.

3.1.1 IOBoardServer Abstract Class

The IOBoardServer AC provides the generic data and methods needed for any I/O board server. Some of the data members include the name of the server, the frequency at which it will run, the priority at which it will be scheduled, and other information about the I/O board (# of encoder inputs, # of digital inputs, # of A/D inputs, *etc.*)

The only methods the AC provides are a constructor, a destructor, status information methods, and a `doMessageLoop ()` method. The `doMessageLoop ()` method is pure virtual, which means that it is not defined in the AC, it must be defined by classes that are derived from the AC. This is because each particular I/O board server will have a different implementation since the board hardware is different. Examples of classes derived from the IOBoardServer are discussed below.

3.1.1.1 MultiQServer

The MultiQServer is a hardware server for Quanser Consulting's MultiQ 1, 2, and 3 boards. These boards have 8 digital inputs and outputs (16 for the MultiQ 1), 8 A/D, 8 D/A, 3 timers, and anywhere from 0 to 6 (or 8 for the MultiQ 3) encoder channels. The MultiQServer class uses a MultiQ hardware interface class to communicate with the MultiQ board hardware. The constructor for the MultiQServer class first calls the IOBoardServer constructor, passing the generic information (*e.g.* server name, frequency, *etc.*) to the superclass. However, the MultiQServer constructor accepts additional board-specific parameters (*e.g.* base address of the board, IRQ used for the timer, # of A/D channels, # of D/A channels, *etc.*) The pure virtual doMessageLoop() method declared in the AC is defined here. Pseudocode for this function is shown below.

```
do forever
      wait for a tick from the timer source
      if server is falling behind report the error and quit (this
            would be caused if the server is performing too many
            operations to complete within one sample period)
      read the encoders, a/d, digital inputs
      write the digital outputs and d/a
```

The order of operations is not exactly as listed above. Operations are interleaved so that time waiting for A/D conversions to complete is minimized. An A/D read requires 19 microseconds. Encoder reads and digital inputs/outputs as well as analog outputs require 5 microseconds. The server initiates an A/D read on one channel, then writes a D/A channel, reads an encoder, and then checks on the status of the A/D. This is repeated so that non A/D operations are interleaved with the A/D reads.

3.1.1.2 STGServer

The STGServer works with the ServoToGo S8 model 1 and 2 motion control board. This board provides 8 A/D, 8 D/A, 8 encoder inputs, 32 bits of digital input/output, one timer that can trigger an ISA bus interrupt, and a watchdog timer. The constructor for this server is similar to the MultiQServer constructor, but requires additional parameters because of the additional capabilities of the S8 board (*e.g.* the digital bits can be configured as inputs or outputs). Programming the S8 is more complicated than programming the MultiQ, however this is hidden from the user due to the use of the hardware client/server architecture. The `doMessageLoop()` function of the STGServer is similar to that of the MultiQServer.

3.1.1.3 CBDIOServer

The CBDIOServer works with the ComputerBoards CBDIO24/CTR3 boards. This board provides 24 bits of digital I/O (which can be configured as input or output), 3 counters, and the ability to generate an ISA bus interrupt when the count reaches zero. The CBDIOServer class derives from the IOBoardServer AC. Its constructor is similar to the MultiQServer constructor, in that it accepts board specific parameters (base address, IRQ of the timer, *etc.*) However it does not have any A/D, D/A, or encoder capabilities. The `doMessageLoop ()` method declared in the AC is defined here.

```
do forever
      wait for a tick from the timer source
      if server is falling behind report the error and quit (this
            would be caused if the server is performing too many
            operations to complete within one sample period)
      read the digital inputs
      write the digital outputs
```

3.1.1.4 FastADServer

The MultiQ and STG S8 boards can read 8 channels of A/D at about 5KHz maximum if none of the other board features are used (*e.g.* A/D, encoders, *etc.*) There are two variables that determine this rate. The sample period T_s is equal to some CPU overhead (instructions executing on the CPU to set up the A/D conversion) called T_O added to the actual time for the A/D converter on the I/O board to complete a conversion (called T_C). Upgrading the CPU to a faster clock speed will decrease T_O but will have no effect on T_C, which is a function only of the A/D converter. On modern CPUs, T_C dominates T_O. The A/D converters on the MultiQ and STG S8 boards require about 19 microseconds to convert. To read all 8 channels requires 152 microseconds, which yields a rate of 6.5KHz. The actual rate of about 5KHz is due to overhead, since the 19 microseconds is the time required for one conversion.

When the D/A channels and encoders are also used, these boards can run at a maximum of about 3KHz. For most control experiments, 3KHz or less is an adequate sample rate. Some experiments do require a higher rate. One example was an active magnetic bearing experiment performed at Clemson University. Ideally the sample rate should have been 20KHz. To approach this rate, faster A/D boards were needed. The ComputerBoards PCI-DAS1602/16 is a 16 channel 200KHz 16 bit resolution A/D board. The maximum acquisition rate of this board can be achieved only by using large DMA buffer transfers. This is inappropriate in a control situation, where each sample must be read and acted on during each control period. The real sample rate achieved with this board on a Pentium II 266 was 12KHz for reading 16 channels. Compared to about 1.5KHz for 16 channels (2 boards) with the other boards, this is a significant increase. Note that due to the use of client/server architecture and OOP techniques, the control programs did not need to be modified to take advantage of this high-speed A/D board.

3.1.2 IOBoardClient

The hardware servers listed above are programs that communicate with hardware directly. The user writes a control program, which must somehow interact with the hardware. Control programs use an IOBoardClient class. This is not an AC, like IOBoardServer, this is a class that can be instantiated. It is a general class that can work with any IOBoardServer derived server. This means that if a control program uses the IOBoardClient class to communicate with a MultiQServer, it does not need to be changed to communicate with an STGServer or a CBDIOServer. If the control program did not use the client/server model, and rather used the hardware interface class (or some other statically linked library), this would not be possible. The control program would have to be modified and recompiled in order for it to work with a different I/O board, as in QMotor 2.0. Table 1 lists the methods provided by the IOBoardClient class.

Table 1 - IOBoardClient Class Methods

Method Prototype	Description
IOBoardClient (char *servername)	Constructor. Connects the client to the named IOBoardServer. This could be a MultiQServer, STGServer, CBDIOServer, FastADServer or any other server derived from the IOBoardServer base class.
~IOBoardClient ()	Destructor. Disconnects the client from the server.
int isStatusOk(), int isStatusError()...	Status inquiry methods.
int getNumEncoders ()	Returns how many encoders this I/O board has (0 – none). A STGServer would return 8, a FastADServer would return 0.
int getEncoderValue (int channel)	Returns the current encoder count for the given encoder channel.
int getEncoderIndexValue (int channel)	Returns the encoder count that was latched when an index pulse last occurred (not supported by all boards).
void setEncoderValue (int channel, int value)	Preset the encoder value for the given channel.
void setEncoderIndexValue (int channel, int value)	Preset the encoder index value for the given channel.
int getNumAdc ()	Returns the number of A/D channels on used by the server. On an STG S8 could return 0 – 8, depending on how the server was started. On a FastADServer could return 0 – 16. Running a server with only the # of A/D channels needed active allows the server to run faster.
int getNumDac ()	Get the # of D/A channels the server supports. On an STG S8 this could be 0 – 8, *etc.*
double getMinDacVoltage ()	Get the minimum output voltage that his server supports (STG S8 would be –10, on a MultiQ this would be –5).

double getMaxDacVoltage ()	Get the maximum output voltage that his server supports (STG S8 would be +10, on a MultiQ this would be +5).
double getMinAdcVoltage ()	Get the minimum input voltage that this server supports. On an STG S8 this would be either –5 or –10 (jumper settable). On a MultiQ this would be –5.
double getMaxAdcVoltage ()	Get the maximum input voltage that this server supports. On an STG S8 this would be either +5 or +10, on a MultiQ this would be +5.
double getAdcValue (int channel)	Returns the current voltage read by the given A/D channel.
void setDacValue (int channel, double voltage)	Writes the given voltage out to the given D/A channel.
int getNumDiginBits ()	Returns the number of digital input bits. On an STG S8 this could be up to 32, on a MultiQ this is 8.
int getNumDigoutBits ()	Returns the number of digital output bits. On an STG S8 this could be up to 32, on a MultiQ this is 8.
unsigned char getDiginByteValue (int byte)	Return the value of the given digital input byte.
int getDiginBitValue (int bitPosition)	Return the value (0 or 1) of the given digital input bit.
void setDigoutByteValue (int byte, unsigned char value)	Write the given byte to the given digital output byte.
void setDigoutBitValue (int bitPosition, int value)	Write the given value (0 or 1) to the given digital output bit.

The following simple program shows how to write –3 volts to D/A channel 0.

```
#include “IOBoardClient.hpp”

main ()
{
  IOBoardClient iobc (“iobs0”);

  iobc->setDacVoltage (0, -3);
}
```

Note that this program will work unchanged with both MultiQ or STG S8 boards, or any other I/O board that has at least one D/A channel.

3.2 Timer Servers

To implement a control algorithm in the QMotor 3.0 environment, a timer server must be executing in order to provide an accurate clock signal. Timer servers provide QNX proxy messages to their clients at a desired frequency. For example if the control frequency is selected as *2 KHz*, then the control program executes every *0.5 ms*. Each timer server has one clock source but may have multiple clients. Clients ask the timer server to "wake" them periodically at any frequency that is an integer divisor of the timer server's clock source. The clock source may be a software source such as a QNX timer or a hardware source such as the MultiQ board's timer circuitry.

3.2.1 TimerServer Abstract Class

The TimerServer AC has one timing source, and can service many timer clients. Each time the timing source ticks, the server executes, awakens any clients that need to execute, and exits.

In order for the control program to run deterministically, that is at a fixed frequency, regardless of all other system activity, a hardware timer interrupt is used. A hardware ISR must be used because hardware interrupts preempt all processes, including the highest priority processes. QNX provides the facility to make one interrupt the highest priority interrupt in the system. This interrupt is assigned to a hardware timer IRQ. Note that currently timer clients must execute at frequencies that are integer divisors of the timer server’s base frequency. A timer server that runs at 3KHz could have clients running at 3KHz, 1.5KHz, 1KHz, 600Hz, *etc*. but not 2.9KHz.

Timer servers that have a hardware timer component attach an ISR to the timer’s IRQ. This ISR increments a sequence counter in shared memory, and triggers the QNX proxy that scans through the list of connected clients and wakes those that need to run.

3.2.1.1 MQTimerServer

The MultiQ board provides one oscillator that is tied to the input of three counters. Each counter can generate an ISA bus interrupt on terminal count. QMotor 3.0 applications typically only use one timer server, so only one of these interrupts is used.

3.2.1.2 STGTimerServer

The STG S8 board has one timer, which serves as the timing source for the timer server, as well as for the internal functions of the board. This board can generate one ISA bus interrupt on terminal count.

3.2.1.3 CBTimerServer

The CIO-DIO24/CTR3 board provides one oscillator that can be connected to up to three counters (in a daisy chain configuration) to produce very low frequencies. This board can generate one ISA bus interrupt on terminal count.

3.2.1.4 QNXTimerServer

While hardware timers are required to guarantee that the timer server has highest priority, in situations where timing is not that critical and the frequency will not exceed 2KHz, a QNX software timer can be used. The QNXTimerServer uses a QNX timer to wake its clients. Note that a QNXTimerServer may fall behind with no way to detect this condition, and should not be used for hard real-time applications where missed control cycles would impact performance. All hardware interrupts will preempt a QNXTimerServer and so a very active PCI network card or large amounts of disk activity could starve a QNXTimerServer.

3.3 TimerClient

All timer servers are accessed through the generic TimerClient class. A program can use the TimerClient class to insure it executes at a fixed frequency, and to detect if the computation it is performing is too slow to execute in one control period. One advantage of using a TimerClient is that the program does not need to be recompiled or modified in order to use a different timing source (*e.g.* MultiQ board, STG S8, QNX software timer, *etc.*) Table 2 lists the methods provided by the TimerClient class.

Table 2 - TimerClient Class Methods

Method Prototype	Description
TimerClient (char *timerServerName, double frequency, int priority = -1)	Constructor. Connects the client to the named TimerServer. This could be a MQTimerServer, STGTimerServer, CBDIOTimerServer, QNXTimerServer, or any other server derived from the TimerServer base class. The client will verify that the server can support the specified frequency. The optional priority parameter allows the client to request a given process priority from the server (see the setprio() QNX library function). A priority of –1 indicates the priority of the client should not be modified.
~TimerClient ()	Destructor, disconnects from the timer server.
resetBaseTicks (int ticks)	Resets the server's tick count for this client. Should be called right before starting the control loop execution.
start ()	Tells the server to start waking this client up.
stop ()	Tells the server to stop waking this client up.
pid_t waitForTick (void *buffer=0, int sizeOfBuffer=0)	Waits for a tick from the timer server, or for asynchronous messages from other processes. Returns –1 and sets status to OK if the client is running on time. If an asynchronous messages was received, returns the PID of the sending process. Falling behind the timer, or problems with the server's base timer are reported by an error status.

3.4 ControlProgram

The ControlProgram class provides a framework for developing control programs. All of the details of running in real-time, logging variables, changing control parameters, *etc.* are handled by the class. Certain methods are provided by the class, and others must be written by the user. Table 3 below lists the methods provided by the class.

Table 3 - ControlProgram Class Methods

Method Prototype	Description
ControlProgram (int argc, char *argv[], char *timerServerName, int priority)	Constructor. The command line arguments are passed, as is the name of the timer server to connect to, and the desired priority of the control program.
~ControlProgram ()	Destructor.
registerLogVariable (double *address, char *name, char *desription=””, int rows = 1, int cols = 1)	This function is called to register a variable as a “log variable”. Log variables may be logged in real time during the control run. Scalers and 1 and 2 dimensional arrays are supported.
registerControlVariable (double *address, char *name, char *description=””, int rows = 1, int rows = 1)	Called to register a variable as a “control parameter.” Control parameters may be changed during the control run, and are typically used for tuning gains, though they may have other uses. Scalars and 1 and 2 dimensional arrays are acceptable.
setEventTriggeredLogMode (char *variableName)	Sets the logging mode to event triggered. Use enable/disableLogging() in the control loop to start/stop logging an event triggered variable. This is useful if you’re interested in logging data after some event occurs and you don’t know when the event will occur.
enableLogging (char *variableName)	Enable logging for the given variable. Used to enable/disable logging of a variable from within the control loop, useful for even triggered logging.
disableLogging (char *variableName)	Disable logging for the given variable.
setContinuousLogMode (char *variableName)	Log the last N seconds of data for this variable. This is useful if you want to keep the last N seconds of data, prior to the end of the control.
setTimedLogMode (char *variableName)	This is the most common logging mode. This variable will be logged for a certain time, starting at a certain time.
setNoLogMode (char *variableName)	This is variable will not be logged.
setLogStartTime (char *variableName, double logStartTime)	When logging in timed mode, start logging this variable at the given time (in seconds).
setLogDuration (char *variableName, double logDuration)	All log modes must specify a duration – this is used to allocate the memory for the log. The total log memory for this variable will be duration times frequency.
setLogFrequency (char *variableName, double logFrequency)	Set the frequency at which the variable will be logged. The log frequency must be an integer divisor of the control frequency (*e.g.* if the control frequency is 1000, you may log at 1000, 500, 250, 200, 100, *etc*. Hz, but not at 900).

setWatched (char *variableName)	Set this variable to be watched during program execution. The supervisor may read its value at any time.
setNotWatched (char *variableName)	Do not watch this variable during the control run.
mainLoop ()	Call this function to begin execution of the main control loop.

Table 4 lists the data members defined in the ControlProgram class that can be used by the user.

Table 4 - ControlProgram Class Data Members

Member Declaration	Description
double d_controlFrequency	This is the frequency in Hz at which the control will run.
double d_controlPeriod	Inverse of the control frequency, in seconds.
double d_controlDuration	How long the control will run (in seconds).
int d_runForever	Set to 1 to indicate an infinite control run, set to 0 if this is a timed run (and also set d_controlDuration).
double d_elapsedTime	Elapsed time in seconds since the control started.
int d_elapsedTicks	Elapsed ticks since the control started. One tick is the represents the passing of one control period.

Table 5 lists the methods that he user must write.

Table 5 - ControlProgram Class Virtual Functions User Must Write

Method Prototype	Description
int startControl ()	This function will be called when the user presses the START button in the GUI. It is called each time the control is started.
int control ()	This function implements the control loop. Do your inputs, control computations, and outputs here. control() will execute at the given control frequency. Return 0 to continue running the control, or non-zero to abort the control.
int stopControl ()	This function is called when the control run ends (either because a timed run has finished, the user has pressed the STOP button in the GUI, or some other condition has caused the control to stop). This is a good place to reset the system to a know state (*e.g.* zero out the D/A channels, *etc.*)
int handleMessage (pid_t pid, char *message)	Return 0 to continue running, or non-zero to

	stop the control. This function will be called if the control program receives a QNX message it does not recognize (*e.g.* it is not from the timer or a server). This function can be left out, or can be used to implement IPC with other processes in your control loop.

Appendix A lists a simple control program that connects to a MultiQ timer server and a MultiQ IO board server.

3.5 Supervisor

The Supervisor class provides a framework for writing user interfaces that supervise the execution of control programs. The QMotor GUI is a Supervisor. If a user wants to develop a custom GUI for use with a QMotor control program, he may use the Supervisor class to do so. The Supervisor class provides the following methods that can be used by a derived class.

Table 6 - Supervisor Class Methods

Method Prototype	Description
Supervisor ()	Constructor.
~Supervisor ()	Destructor.
loadControlProgram (char *controlProgramFullPath)	Loads the given control program.
unloadControlProgram ()	Unloads the current control program.
saveConfiguration (char *configFileFullPath)`	Saves the current configuration to the given configuration file.
startControl ()	Starts the execution of the main control loop. In the QMotor GUI this is called when the user hits the START button.
stopControl ()	Stops execution of the control program.
char *exportRunToMatlab (char *matlabFileFullPath, char *exportingClass = 0)	Exports the data from the current control run to a MATLAB file.
char *exportLogVariablesToMatlab (ofstream &matlabFile)	Exports all logged variable data to a MATLAB file.
char *exportControlParametersToMatlab (ofstream &matlabFile)	Exports the current values of the control variables to a MATLAB file.
receiveEnvironmentParameters ()sendEnvironmentParameters ()	Sends/Receives the list of environment parameters to/from the control program.
sendLogVariableList () receiveLogVariableList ()	Sends/Receives the list of log variables to/from the control program.
sendControlParameterList () receiveControlParameterList ()	Sends/Receives the list of control parameters to/from the control program.

cleanUpAfterControlProgramDied ()	Called if the control program crashes. Should clean up all memory, *etc*.

3.6 Utility Classes

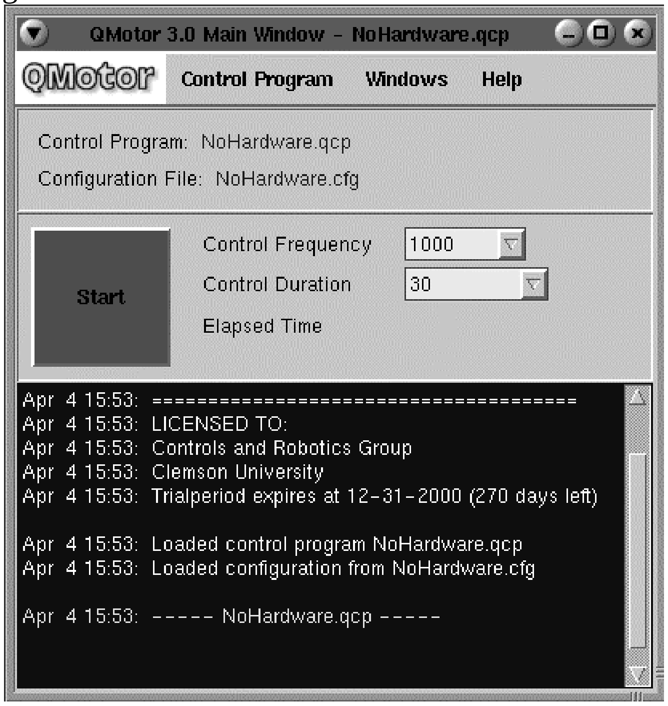
QMotor relies on several utility classes. The IOBoardClient class, discussed previously, is a utility class. The other utility classes used provide list management, shared memory functionality, filtering, *etc*.

4 The QMotor 3.0 GUI

The QMotor GUI allows the user to interact with the control program. It is used to start and stop a control program, tune gains online, and log and plot data in real-time.

4.1 Main Window

From the main window, the user can load a program, set the control duration and control frequency, and start executing the desired control algorithm. In addition, the main window also allows the user to open the following sub-windows: 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 5.



Control Program: NoHardware.qcp
Configuration File: NoHardware.cfg

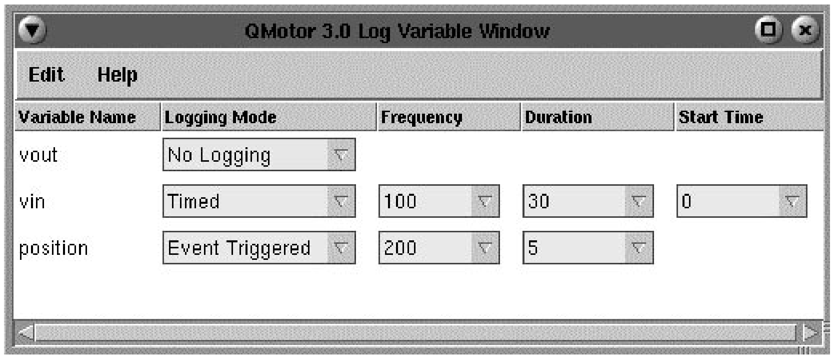
Control Frequency 1000
Control Duration 30
Elapsed Time

```
Apr  4 15:53:  ======================================
Apr  4 15:53:  LICENSED TO:
Apr  4 15:53:  Controls and Robotics Group
Apr  4 15:53:  Clemson University
Apr  4 15:53:  Trialperiod expires at 12-31-2000 (270 days left)

Apr  4 15:53:  Loaded control program NoHardware.qcp
Apr  4 15:53:  Loaded configuration from NoHardware.cfg

Apr  4 15:53:  ----- NoHardware.qcp -----
```

Figure 5 - Main Window

4.2 Log Variable Window

The log variable window (Figure 6) 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, the logging frequency, logging start time, and the logging duration.

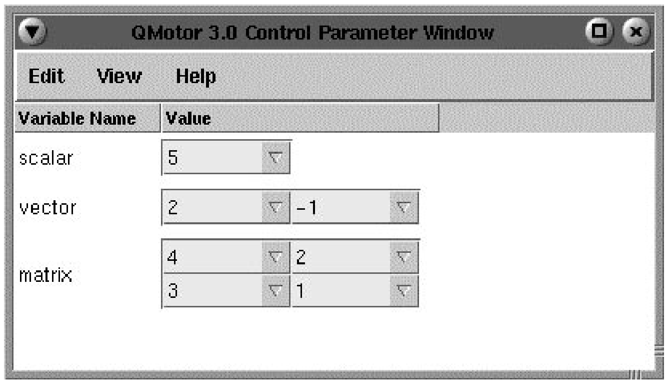


Variable Name	Logging Mode	Frequency	Duration	Start Time
vout	No Logging			
vin	Timed	100	30	0
position	Event Triggered	200	5	

Figure 6 - Log Variable Window

4.3 Control Parameter Window

The control parameter window (Figure 7) 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 various values without recompiling the C++ control program (*note*: the control parameters can only be set to constant values; no complex functions are permitted). Control parameter values can be modified while the control program is running (online parameter tuning).

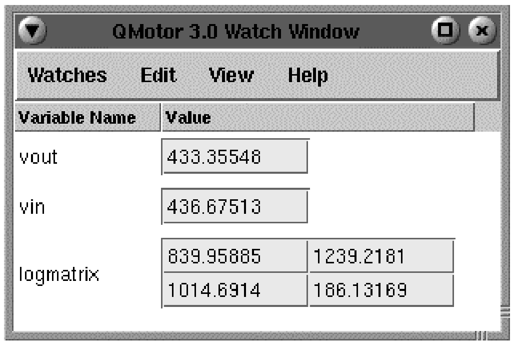


Variable Name	Value	
scalar	5	
vector	2	-1
matrix	4	2
	3	1

Figure 7 - Control Parameter Window

4.4 Watch Window

The watch window (Figure 8) allows the user to see the real-time values of selected log variables during control execution. Note that the last logged value of a variable is displayed in the watch window at the termination of the control cycle.

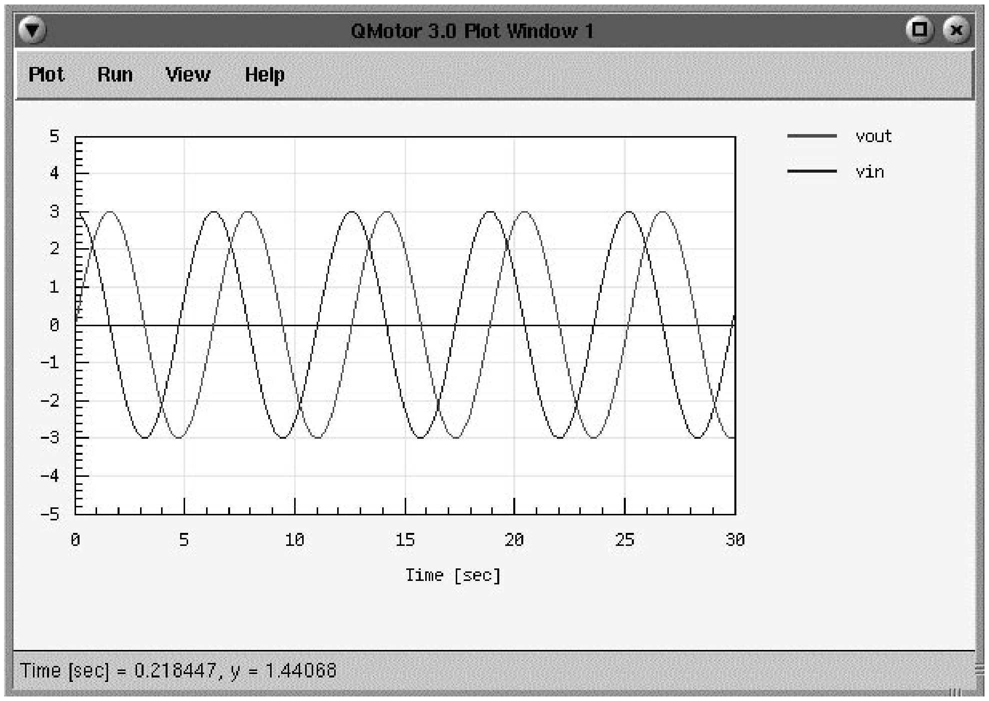


Variable Name	Value	
vout	433.35548	
vin	436.67513	
logmatrix	839.95885	1239.2181
	1014.6914	186.13169

Figure 8 - Watch Window

4.5 Plot Windows

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 9). All log variables appearing in the *log variables window* are available for plotting purposes (*note*: a variable is only available for plotting if the variable is set to be logged from the *log variable window*). Any number of plot windows may be open at once, and any number of variables may be plotted in each window. Numerous auto-scaling options are available. The plot windows may be zoomed and panned, and provide very powerful plotting options, as well as export to MATLAB.



```
Time [sec] = 0.218447, y = 1.44068
```

Figure 9 - Plot Windows

5 Conclusion

This paper documents the successful use of OOP techniques in the development of QMotor 3.0. These techniques have allowed QMotor 3.0 to be flexible and easily extensible. Support for several new hardware interface boards has been added after QMotor 3.0 was finished, simply by providing new hardware servers that are compatible with the IOBoardClient class. This did not require any modification of the QMotor 3.0 source code and user control programs did not need to be recompiled to take advantage of new interface hardware. Multiple hardware interface boards can be used by one QMotor 3.0 control program by simply starting multiple hardware servers. This is in contrast to QMotor 2.0, where the source code itself had to be modified to accommodate new hardware or multiple boards.

QMotor 3.0 has been used as the basis for a robot control system, called the QMotor Robotic Toolkit, which also uses OOP techniques to provide a system that can control Puma, Barrett and IMI robots and is extensible to any robotic system. The QMotor RTK was initially developed using Puma manipulators, and was later extended to the Barrett WAM and IMI Direct Drive robot. QMotor 3.0 has been used by Clemson University and others to implement a wide variety of control algorithms, some of which are documented in [1], [2],.[3], and [4].

A new version of the QNX real-time operating system will be released shortly. This version, called the QNX Real-Time Platform (RTP), has many advantages over the old QNX 4.2x OS, including support for symmetric multiprocessing (SMP), and free availability for non-commercial use. QMotor 3.0 will be ported to this new OS, with support for SMP added to allow control programs to take advantage of multiple processors. Note that this is a homogenous multiprocessor system, which is much simpler and less expensive that the old Host PC/DSP SBC heterogeneous systems used in the past. QMotor 3.0 will also be enhanced for use in industrial and embedded systems, with the ability to connect the GUI to an already executing control program without stopping the control. This would aid in debugging or tuning mission critical control programs that can not he started and stopped.

6 Appendix A - Sample QMotor 3.0 Control Program

```
///==============================================================================
/// file   : MyControlProgram.hpp
///------------------------------------------------------------------------------
/// Class definition for the user's control program.
///==============================================================================

#include "ControlProgram.hpp"
#include "IOBoardClient.hpp"

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
class MyControlProgram : public ControlProgram
{
 protected: // Protected data

       // Log Variables
       double d_vin;
       double d_vout;
       double d_vinOverTwo;

       // Control Parameters
       double d_scaleFactor;
       double d_offset;

       // Hardware Servers
       IOBoardClient *d_iobc;

 public: // Public methods

       MyControlProgram
       (
              int argc,                     // # of cmd line arguments
              char *argv[],                 // Array of cmd line arguments
              char *timerServerName="ts0",  // Name of the timer server providing timing
              char *ioboardServerName="mqs0",// Name of the IOBoard server providing I/O
              int priority = 27
       );
              // Constructor, initialize hardware servers, etc. here. Called only once,
              // when the control program is loaded.

       ~MyControlProgram ();
              // Destructor, do cleanup here. Called only once, when the control program
              // is unloaded.

       int startControl();
              // Called each time the control is started by the Supervisor (eg. the
              // START button is pushed. Optional.

       int stopControl();
              // Called at the end of each control run. Optional.

       int control ();
              // Main control function. Called each time through the control loop.

       int handleMessage (pid_t pid, char *message);
              // Called when a message that is not from the Timer server arrives.
              // This allows you to receive QNX message from other processes. This
              // function is optional.
};
```

```
///==============================================================================
/// name   : MyControlProgram
///------------------------------------------------------------------------------
/// input  : argc - # of command line args passed to this program
///          argv - Ptr to array of command line args
///          timerServerName - Name of the timer server which will provide the
///                            timing for this control program.
/// Constructor for the user's control program. This is called when the
/// program first starts. Connections to needed servers should be made here,
/// log and control variables are initialized and registered here. Any setup
/// that must occur when the program is first started should be placed here.
///==============================================================================

MyControlProgram::MyControlProgram (int argc, char *argv[], char *timerServerName,
  char *ioboardServerName, int priority)
  : ControlProgram (argc, argv, timerServerName, priority)
{

  // Initialize your servers here. I will use one IOBoard server.

  d_iobc = new IOBoardClient (ioboardServerName);

  if (d_iobc->isStatusError ()) // If couldn't locate the IOBoard server,
  {                             // set the status and return.
        d_status.setStatusError ();
        return;
  }

  // Set control program parameters. If running under a Supervisor (like the
  // GUI), these values will be overriden by the Supervisor.

  d_runForever = 0;
  d_controlFrequency = 100; // Control freq. in Hz
  d_controlDuration = 10.0;    // Control duration in seconds (-1 = run forever)

  // Initialize your control variables here. These must be declared in
  // the class definition for this control program (eg. MyControlProgram.hpp)

  d_scaleFactor=1.0;
  d_offset=0.0;

  // Register your control variables. Only registered control variables will
  // appear in the Supervisor (ie. the GUI). You do NOT have to register all
  // your variables, only the ones you want to change from the GUI. The
  // 2nd parameter is the description that will be seen in the GUI.

  registerControlParameter (&d_scaleFactor, "d_scaleFactor",
               "Amplitude Scale Factor");

  registerControlParameter (&d_offset, "d_offset", "DC Offset");

  // Initialize your log variables here.

  d_vin=0;
  d_vout=0;

  // Register your log variables. Only registered log variables can
  // be logged/plotted by the Supervisor. The 2nd parameter is the
  // description that will be seen in the GUI.


  registerLogVariable (&d_vin, "d_vin", "Input Voltage");
  registerLogVariable (&d_vout, "d_vout", "Output Voltage");
  registerLogVariable (&d_vinOverTwo, "d_vinOverTwo", "Half Input Voltage");

  d_status.setStatusOk (); // Constructor succeeded
}
```

```
///==========================================================================
/// name   : ~MyControlProgram
///--------------------------------------------------------------------------
/// Destructor for the user's control program. Called only once, as the
/// control program exits. Put any cleanup stuff that must happen when your
/// conrol program quits here.
///==========================================================================

MyControlProgram::~MyControlProgram ()
{

  delete d_iobc; // Disconnect from the IOBoard server
}


///==========================================================================
/// name   : startControl
///--------------------------------------------------------------------------
/// Called each time a control run is started. If running from the GUI, this
/// will be called each time the START button is pushed. Do setup that must
/// occur before each control run here (eg. initializing some counters, etc.)
///==========================================================================

int MyControlProgram::startControl() {
  return (0);
}


///==========================================================================
/// name   : stopControl
///--------------------------------------------------------------------------
/// Called each time a control run ends. If running from the GUI, this
/// will be called each time the STOP button is pushed, or when a timed run
/// ends, or when the control aborts itself.
///==========================================================================

int MyControlProgram::stopControl() {


  d_iobc->setDacValue (0, 0.0); // Zero out the DAC we were using.

  return (0);
}


///==========================================================================
/// name   : stopControl
///--------------------------------------------------------------------------
/// return : 0 - Control run may continue, nonzero - Control should stop
/// Called each control cycle. Do your input, control computations, and output
/// here. If you return 0, the control will continue to execute. If you return
/// nonzero, the control will abort. You may want to abort if some error
/// condition occurs (excessive velocity,  etc.)
///==========================================================================
int MyControlProgram::control ()
{
  double factor;

  // This simple control just multiples the input of ADC channel 0 by a
  // scale factor and adds a DC offset to it, writing the result out to
  // DAC channel 0.

  // Input
  d_vin = d_iobc->getAdcValue (0);

  // Control calculations

  // d_elapsedTime is provided by the superclass ControlProgram and is the
  // time since the control run started (sec)

  factor = sin(d_elapsedTime) * d_scaleFactor;
  d_vout = factor * d_vin + d_offset;
  d_vinOverTwo = d_vin / 2.0;

  // If the output voltage is positive, enable logging of d_vinOverTwo. In
  // the log, I will only have values of d_vinOverTwo logged at times
  // when the output voltage is positive (event-triggered logging mode).

  if (d_vout > 0)
        enableLogging ("d_vinOverTwo");
  else
        disableLogging ("d_vinOverTwo");

  // Output
  d_iobc->setDacValue (0, d_vout);

  return 0;
}


///===============================================================================
/// name   : handleMessage
///-------------------------------------------------------------------------------
/// return : 0 - Control run may continue, nonzero - Control should stop
/// This optional function allows your control program to receive QNX messages
/// from processes other than the Timer server. If a msg arrives and it is not
/// from the Timer server, this function will be called to handle it. Based on
/// the message you may choose to stop the control (return nonzero) or continue
/// the control (return 0).
///===============================================================================

int MyControlProgram::handleMessage (pid_t pid, char *message)
{
  int msg;

  msg = *((int *)message); // Convert the msg to an integer
  Reply (pid, 0, 0);       // Reply to the msg so the sender is unblocked


  if (msg == 1) // If the msg was "1" abort the control.
        return (1);

  return (0);           // The control may continue.
}



main (int argc, char *argv[])
{

  // Instantiate the control program.
  // The first 3 arguments are required (argc, argv, and the name of the timer
  // server. The fourth argument and beyond are determined by the user.
  // Here it's just the name of the IOBoard server used for I/O. If you use
  // additional servers you may pass their names as arguments here.

  MyControlProgram cp (argc, argv, "mqts0", "mqs0", 19);

  // If couldn't start the control (maybe couldn't connect to one of the
  // hardware servers, etc.), abort.
  if (!cp.d_status.isStatusOk ())
  {
        cerr << "Can't start control" << endl;
        abort ();
  }

  // Log input voltage in timed mode, starting at time=0 for 0.02 seconds
  // at a logging frequency of 1000 Hz.

  cp.setTimedLogMode ("d_vin");
  cp.setLogFrequency ("d_vin", 1.0);
  cp.setLogDuration ("d_vin", 10.0);
  cp.setLogStartTime ("d_vin", 0.0);

  cp.setWatched ("d_vin");

  // Log output voltage in ring-buffer mode, storing 0.01 seconds worth of
  // log data, at a logging frequency of 1000 Hz. Whenever the control is
  // stopped, the log for this variable will have the last 0.01 seconds
  // worth of data in it (i.e. data for tfinal through tfinal-0.01 seconds

  cp.setContinuousLogMode ("d_vout");
  cp.setLogFrequency ("d_vout", 1.0);
  cp.setLogDuration ("d_vout", 0.1);

  cp.setWatched ("d_vout");

  // Log this variable (1/2 the input voltage) at a log frequency of 1KHz,
  // storing at most 0.02 seconds worth of log data. The control program
  // must explicitly do enableLogging() and disableLogging() to enable and
  // disable logging for this variable when an interesting event occurs.

  cp.setEventTriggeredLogMode ("d_vinOverTwo");
  cp.setLogFrequency ("d_vinOverTwo", 1.0);
  cp.setLogDuration ("d_vinOverTwo", 0.1);

  cp.setWatched ("d_vinOverTwo");

  // This mainLoop() is defined by the superclass ControlProgram. It takes
  // care of control program timing, variable logging, etc. It will call the
  // user's functions (startControl(), control(), handleMessage(),
  // stopControl()) as necessary.


  cp.mainLoop();
}
```

7 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, accepted, to appear.

[5] 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

[6] S. P. Nagarkatti, F. Zhang, B. T. Costic, D. M. Dawson and C. D. Rahn, "Velocity Tracking Control of an Axially Accelerated Web", *Mechanical Systems and Signal Processing*, accepted, to appear.

[7] H. Canbolat, D. Dawson, S. P. Nagarkatti, and B. Costic, "Boundary Control for a General Class of String Models", *Proc. of the American Control Conference*, Philadelphia, PA, June 1998, pp. 3472-2476.

[8] E. Zergeroglu, W.E. Dixon, D. Haste, and D. M. Dawson, "A Composite Adaptive Output Feedback Tracking Controller for Robotic Manipulators", *Proc. of the American Control Conference*, San Diego, CA, June, 1999, pp. 3013-3017.

[9] M. Feemster, P. Aquino, D. M. Dawson, and D. Haste, "Sensorless Rotor Velocity Tracking Control for Induction Motors", *Proc. of the American Control Conference,* San Diego, CA, June, 1999, pp. 2158-2162.

[10] S.P. Nagarkatti, D.M. Dawson, M.S. de Queiroz, and B. Costic, "Boundary Control of a Two-Dimensional Flexible Rotor", *Proc. of the Conference on Decision and Control*, Tampa, FL, Dec 1998, pp. 2581-2586.