QMotor 2.0 – A Real-Time PC Based Control Environment

N. Costescu, D. Dawson, and M. Loffler

This paper describes QMotor 2.0, a QNX based 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 operating system (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 its development, QMotor 2.0 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. Four of these experiments are documented in [9-12].

Multiprocessor Systems

The traditional approach to implementing control algorithms as digital computer programs is to use two computers. A typical system consists of a general-purpose computer (GPC) and an embedded DSP board. See [1-4] for experiments using such a system.

Typical Architecture

Fig. 1 shows the typical Host/DSP architecture. The GPC may be a Mac, PC, Unix workstation, etc., and is called the "host". The DSP board is usually interfaced to the host via the host's bus. The control program is developed on the host using a cross development system. It is then downloaded to the DSP board, where it executes. Data to be logged during the control run must be transferred quickly to the host computer, since the DSP board has a very limited amount of RAM (usually 32K or 64K words). The user interface, if present, executes on the host. The user can start and stop the control program, modify control gains, and perform other monitoring

---

This paper is a subset of material presented at IEEE ACC 1998, in Philadelphia, PA. N. Costescu (ncostes@ces.clemson.edu), D. Dawson (ddawson@ces.clemson.edu) and M. Loffler (loffler@ces.clemson.edu) are with the Control and Robotics group in the Department of Electrical and Computer Engineering at Clemson University. Mailing address: 105 Riggs Hall, ECE Department, Clemson University, Clemson, SC 29634-0915.

functions from this interface. All connections to the controlled system (input and output) are made through the DSP board.

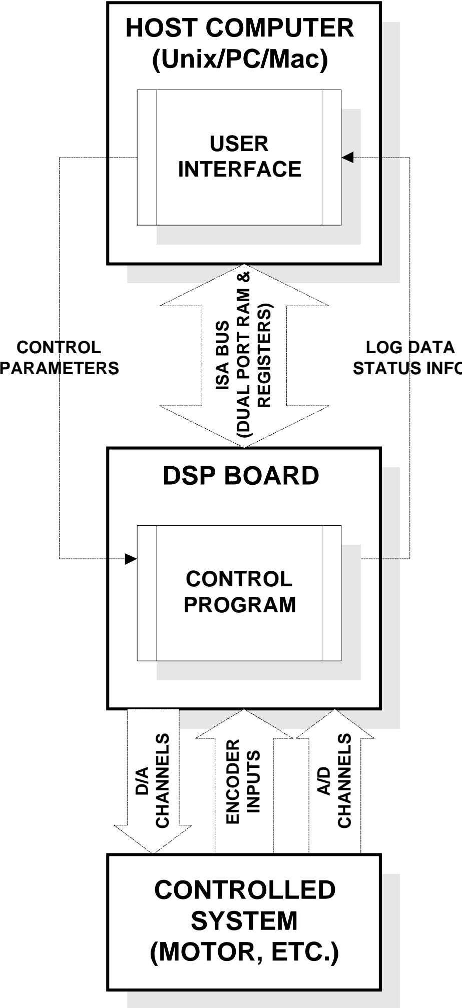

Fig. 1. Typical Host/DSP Architecture

Advantages

Hanselmann gives a basic introduction to the advantages of DSP based computers for use in control applications in [5]. One advantage of the Host/DSP architecture is that it guarantees deterministic response. Not all multiprocessor systems can guarantee deterministic response (e.g. MS-Windows NT, SMP-Linux, etc.) The Host/DSP system is an example of an asymmetric multiprocessor system. Each processor is dedicated to a certain task. In this case, the DSP board runs only the control program. The host computer executes all other functions that do not require

deterministic response (e.g. user interface, data plotting). Since the DSP board is dedicated to running the control program, there are no stringent requirements for the host computer, either with regard to processing power or deterministic response. It is for this reason that the host computer can run a non real-time OS such as MS-DOS, MacOS, MS-Windows, etc.

The other advantage of the Host/DSP solution is that the DSP board is designed to execute small programs that contain many floating point operations very quickly. In the past, the type of GPCs available in a typical university control lab were not powerful enough to perform the necessary floating point computations at reasonable control frequencies (e.g. 2KHz, depending on the controlled system). For these reasons, the multiprocessor Host/DSP system has traditionally been the choice of control engineers for the implementation of control programs.

Examples

WinMotor and QMotor 1.0, two examples of Host/DSP based systems, are examined below.

WinMotor

WinMotor was implemented by the Clemson University Controls and Robotics group in the early 1990s. At that time, there were turnkey Host/DSP systems available for control applications, but they were expensive (well over $10,000 per station for a system from dSpace), and not modifiable (i.e. source code was not available). There were lower cost systems available, but they did not provide user-friendly interfaces. Therefore, WinMotor was conceived with the goals of developing a low-cost, flexible, user-friendly, user-modifiable (i.e. source code is available) system.

The MS-DOS version of the TMS320C30 interface library was ported to MS-Windows 3.1, and a graphical user interface (GUI) was developed as a native MS-Windows 3.1 application. This GUI provides facilities for the user to edit, compile, download, start, and stop DSP control programs. It also provides the ability to plot control variables during the control run. Log data can be saved in Matlab format. A screenshot of the WinMotor GUI appears in Fig. 2.

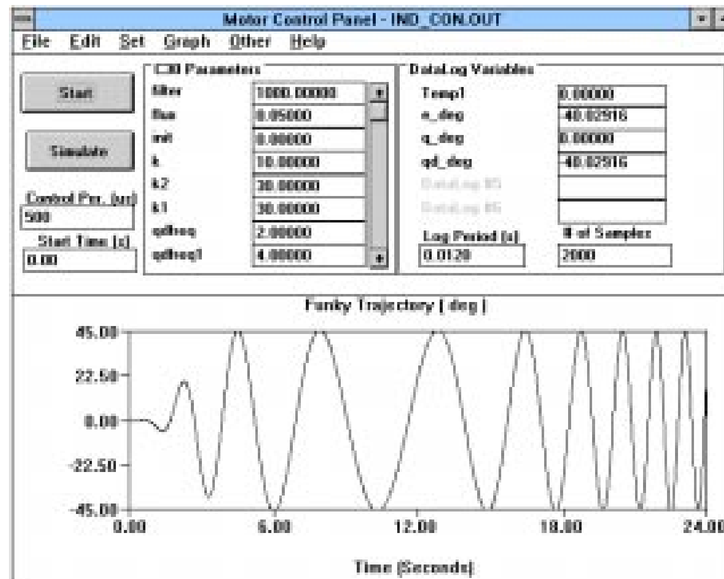

Fig. 2. WinMotor Graphical User Interface

Control gains are entered in the “C30 Parameters” section of the GUI. Logged variables appear in the “Datalog Variables” section of the GUI. The control period, logging period, and number of log samples to be stored are all set from the GUI. Logged variables are plotted during the control run in the lower part of the GUI window. Control programs are edited and compiled through commands available on pull-down menus.

QMotor 1.0

The motivation for development of QMotor 1.0 was the desire to use a more robust multitasking OS that would permit remote execution and display. MS-Windows 3.1 and MS-DOS were replaced by QNX, a real-time microkernel based OS. The GUI was also reimplemented using X/Motif to allow remote display on any computer capable of executing an X server. Aside from minor software differences that enhanced performance or usability (e.g. interrupt-based data logging, concurrent processing, remote execution, multiple data plots), QMotor 1.0 is fundamentally the same system as WinMotor. Control programs developed with WinMotor can be recompiled and executed with QMotor 1.0. A screen shot of the QMotor 1.0 GUI appears in Fig. 3.

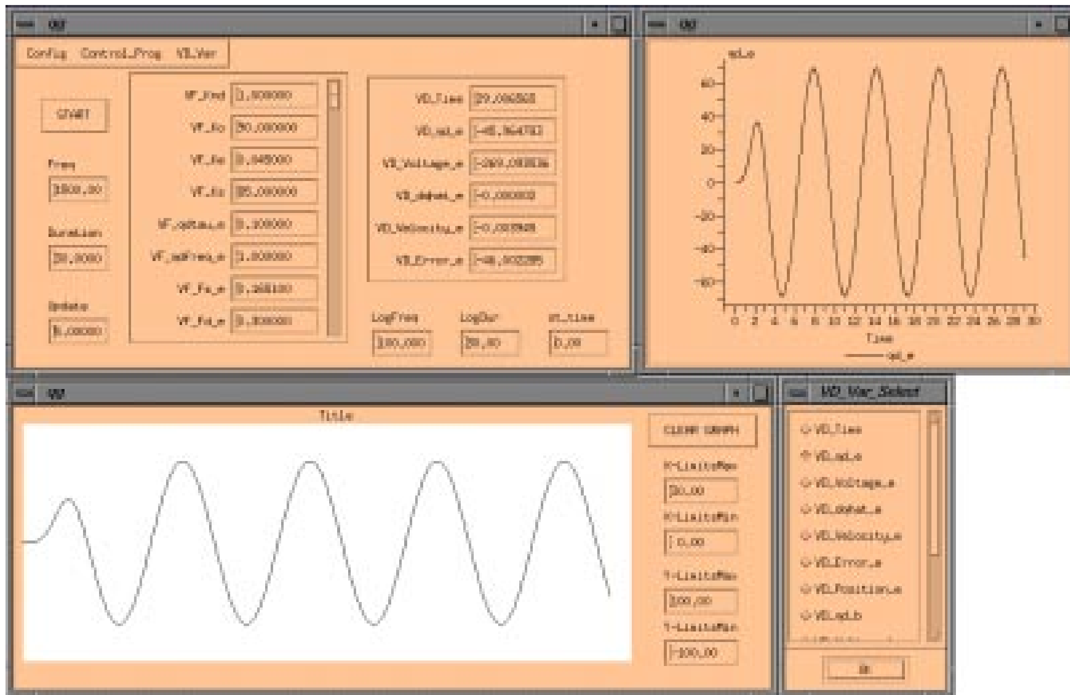

Fig. 3. QMotor 1.0 Graphical User Interface

The layout of the QMotor 1.0 GUI is very similar to that of the WinMotor GUI. Control gains and log variables appear in the main window, as do the pull-down menus containing the compilation and editing commands. The bottom window displays plots of the variables during the control run. The upper right window is a new feature. After the control run is finished, any number of “post-control” plot windows may be opened. These windows can display any number of variables plotted against time, or even one variable plotted against another. The post-control plot windows may be resized, panned, or zoomed.

Disadvantages

Multiprocessor systems have two major disadvantages: cost and complexity. Cost of the hardware and software increases with the number of processors. The cost increase is intensified since the processors in the Host/DSP architecture are heterogeneous and do not share the same OS and development tools. A general-purpose host computer must be purchased, along with its OS and development tools (e.g. a PC with MS-Windows and Visual C++). The DSP board must also be purchased, along with its cross development tools (e.g. Spectrum C30 board with TI C compiler).

A multiprocessor system is also more complex. System administration skills are required for the host, as well as for the DSP board. For interactive control applications, the user must become familiar with the host computer's native development tools in addition to the DSP development tools used to develop the control programs. This is necessary for development of the host resident front-end of the application (e.g. the WinMotor or QMotor 1.0 GUIs). The transfer of data between the DSP board and the host can also be an intricate process, requiring some programming sophistication if real-time plotting and other dynamic interaction between host and DSP must occur.

Single-Processor Systems

Motivation

The disadvantages listed above motivated the investigation of a possible single-processor solution. Several recent advances in PC hardware and software technology have made the PC a viable platform for a single-processor implementation.

The two advantages of multiprocessor systems listed in the previous section are deterministic response and processing power. Until recently, the processing power necessary for the computation of complex control algorithms in real-time was not available on GPCs. This was the reason for pairing a DSP board with a GPC (instead of simply using multiple general-purpose processors). This is no longer true.

One of the experiments being run with WinMotor and subsequently QMotor 1.0 was a direct-drive two-link revolute planar robot with exaggerated joint flexibilities [6]. The control program consists of twenty-six pages of DSP 'C' code (430 lines). By replacing writes to the D/A channels and reads from the encoder and A/D channels with stubs, a platform independent "control benchmark" was created. This control benchmark was representative of the complexity of the control programs developed by our group at the time. Since the hardware specific I/O code segments were removed from the benchmark, it is a measure of the time required for the computation portion of the control program (including all software filtering). The control benchmark was ported to 16-bit DOS, 32-bit DOS, MS-Windows 3.1, MS-Windows 95, and MS-Windows NT. Execution times on various hardware platforms and OSs were compared. Speedup relative to our C30 Host/DSP system was also computed. A speedup of 1.0 indicates the

system is as fast as the C30, a speedup of 2.0 indicates the system is twice as fast as the C30, etc. Table 1 shows these results. Execution time is for one million iterations.

Table 1. Benchmark Execution Time (Seconds) and Speedup (Relative to Base C30 System)					
OS	CPU	32 bit DOS Execution Time (sec)	16 bit DOS Execution Time (sec)	32 bit DOS Speedup	16 bit DOS Speedup
-	C30	548		1.00	1.00
Win 3.1	486DX2/66	398	514	1.38	1.07
Win 3.1	P5/133	36	61	15.22	8.98
Win95	P5/166	30	50	18.27	10.96
NT 4.0	P6/180	19	36	28.84	15.22
NT 3.51	P6/200	17	33	32.24	16.61
NT 4.0	PII/400	7	13	78.29	42.15

The C30 board computes the control at 1.82KHz. A 486DX2/66 computes the control 1.38 times faster than the C30 DSP board (2.51KHz). A 400MHz Pentium II computes the same control 78.29 times faster than the C30 (142KHz). This performance agrees with the benchmarks performed in [2]. These results indicate that modern GPCs have the processing power required to implement complex control algorithms.

It is interesting to note that the C30 is rated at more than 33 MFLOPS (million floating-point operations per second) by its manufacturer, while the 486DX2/66 is rated at only 3 MFLOPS. It is naive to believe that this MFLOPS rating translates into useful speed when executing a control program, as the control benchmark shows. Here, a 486DX2/66 beat a chip that was rated at over 10 times the 486DX2/66's MFLOPS rating. This is partly due to the fact that a control program consists of more than floating-point operations performed on operands that have been pre-fetched into registers. It is also interesting that the Pentium family of processors showed roughly double the performance increase when running the 32-bit version of the benchmark, when compared to the 16-bit version. The 486DX2/66 was only 38% faster running the 32-bit benchmark compared to the 16-bit benchmark.

The availability of hard real-time PC OSs such as QNX, LynxOS, RTLinux, etc. has also eliminated the multiprocessor architecture's other advantage, deterministic response. Using these OSs, control programs and non real-time tasks can execute concurrently on one processor. Combining the greater processing power of modern PC central processing units (CPUs) with real-time OSs provides the basis for a single-processor architecture that can provide all the functionality of multiprocessor architectures.

Examples

Single-processor PC based systems that claim to be able to run a control program at high frequencies with deterministic response are becoming more common. Most couple a real-time executive with a commercial (or free) non-deterministic OS. Examples include Hyperkernel and RTLinux.

Hyperkernel

Hyperkernel provides a small real-time subsystem for MS-Windows NT. The main advantage of Hyperkernel is that it uses standard MS-Windows NT development tools and executes concurrently with the MS-Windows NT kernel. Disadvantages include cost ($5,000 for a non-academic license, $2,500 for an academic license – not including the cost of MS-Windows NT and development tools) and the fact that it is a fairly new product.

RTLinux

RTLinux is a patch for Linux, the popular free Unix-like OS. It provides a small executive that lies between the hardware interrupts and the Linux kernel. This executive runs real-time tasks (e.g. control programs), which are compiled with the standard Linux development tools. It also runs the Linux kernel as one of these real-time tasks. Communication between real-time tasks and standard Linux tasks is performed using first in first out queues (FIFOs).

RTLinux has many advantages. It is free, and can be downloaded on the World Wide Web (http://rtlinux.cs.nmt.edu/~rtlinux). It can be incorporated into the symmetric multiprocessor version of the Linux kernel (SMPLinux), which allows the use of dual-processor motherboards. An SMP-RTLinux machine could conceivably use one processor to run the control program while the other runs the Linux kernel, creating a system similar to traditional Host/DSP systems, without their disadvantages (except for a slight increase in cost). RTLinux is also small and efficient.

There are some disadvantages to RTLinux. As with most free software, it is not officially supported. The mechanisms for communication between real-time tasks and non real-time Linux tasks are quite limited. Some Unix system administration skills are required to install a Linux system and then apply the RTLinux patch. Despite any disadvantages, RTLinux provides adequate facilities for implementing a control program in real-time.

Design Considerations

A successful single-processor system should incorporate the advantages of traditional multiprocessor systems (deterministic response, adequate computational capability), while also eliminating their disadvantages (cost, complexity, lack of upgradability, lack of flexibility, etc.). Deterministic response is achieved by using a real-time OS (QNX, LynxOS) or a real-time patch/extension to a non real-time OS (Hyperkernel, RTLinux). Adequate computational capability is provided by the use of modern PC CPUs like the Pentium and Pentium II. Cost, lack of upgradability, and flexibility are addressed by choosing a consumer grade GPC, such as the IBM PC compatible computer. Complexity is reduced by using the same OS and development tools to implement both the control program and the user interface. Hardware interfacing is simplified by using integrated motion control interface boards, such as Quanser Consulting's MultiQ board, which provides A/D, D/A, encoder inputs, digital I/O, and timers, all on one board.

QNX

OS Selection

In the traditional Host/DSP architecture, there were no constraints placed on the host's OS since the control program ran on the DSP board. New concerns about the host's OS are introduced by the desire to execute the control and GUI on the same CPU. The OS requirements are listed below.

1. *Low Overhead*: Context switch times and interrupt latency should be minimized.
2. *Deterministic Response*: Hard real-time response down to sub-millisecond resolution is required. An interrupt can not be delayed by a variable amount of time, depending on system load, as in soft real-time or non real-time OSs.
3. *Priority Based Multitasking Preemptive Scheduling*: Since the control program will compete with the GUI and other processes for CPU time, the OS must provide a priority based scheduler that can preempt low priority running processes to allow higher priority processes to execute.
4. *Priority Based Interrupt Service Routine (ISR) Nesting*: Interrupt priorities should be well defined, and high priority ISRs should preempt lower priority ISRs. This

allows a timer ISR to preempt a disk drive ISR, guaranteeing that the control program will always run regardless of system hardware activity.

5. *Network Interface*: The system should provide standard network interfaces. Link layer support should include Ethernet while the transport layer should include TCP/IP. This allows remote operation over the Internet.
6. *Remote Login*: The ability to log in from a remote workstation is highly desirable and available on most good OSs. This facilitates remote system administration and allows remote operation.
7. *Graphical User Interface*: This is necessary if a user-friendly graphical interface with which the user can interact is needed. It is preferable if the windowing system has remote display capabilities built in (as do X and Photon).
8. *Ease of Interfacing to Hardware*: It should be fairly easy to write code that interfaces to the hardware needed for control experiments (A/D, D/A boards, etc.). The OS should not require kernel mode device drivers, as do Unix and MS-Windows NT.

MS-DOS is a non real-time single-tasking monolithic kernel. MS-Windows 3.1 uses MS-DOS as the underlying OS. MS-Windows 95 does not have real-time capabilities. MS-Windows NT has no hard real-time capabilities, and makes interfacing to hardware very difficult and inefficient due to its hardware abstraction layer. Unix style OSs do provide preemptive schedulers, though due to the lack of kernel data structure protection (for historical reasons), a process executing in kernel mode can not be preempted. Unix therefore does not provide the deterministic response needed for hard real-time applications. Interfacing to hardware under Unix is also complicated, requiring modification of the kernel. There are hacks and patches to these OSs that try to add real-time capabilities, but they are not robust.

Key QNX Features

Many real-time OSs are currently available for PCs. QNX, a real-time microkernel OS developed by QNX Software Systems Limited, was selected. QNX met all of the above requirements, and was also very affordable due to an excellent educational program. Since QNX is a microkernel, device drivers are not contained in the kernel. In fact, a device driver is not

necessary in order for a program to communicate with a hardware device. If a user program is given sufficient privilege, it can directly access memory and I/O ports, attach hardware interrupt service routines, etc. This makes writing hardware interface software very easy. QNX has very low context switch times (6μsec on a 486DX2/66) and low interrupt latency (7μsec on a 486DX2/66). Photon is the native windowing system, and X/Motif is also available. QNX native networking is built into the microkernel, but TCP/IP is also available. See [7] or [8] for more details about the architecture of QNX.

QMotor 2.0

QMotor 2.0 is an implementation of the single-processor architecture for the development and execution of control programs.

Overall Design

A block diagram of the overall design of QMotor 2.0 appears in Fig. 4.

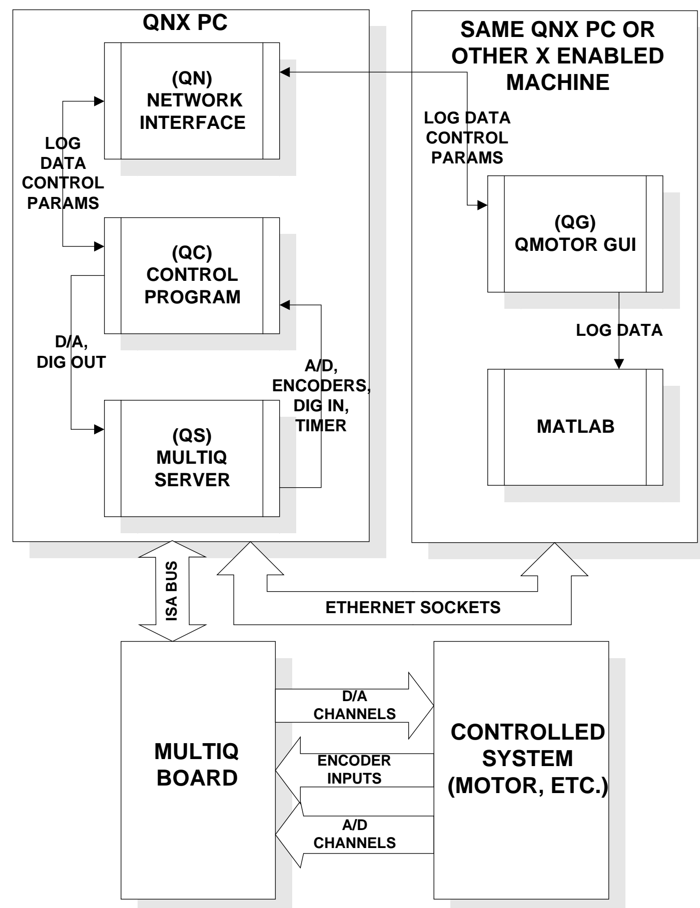

Fig. 4. Overall QMotor 2.0 Design

Hardware Architecture

The hardware components of QMotor 2.0 are listed below.

1. *Intel Pentium Family PC:* An ISA bus based PC with an Intel Pentium or equivalent processor is necessary because QNX only runs on Intel platforms. It is possible to use a 486; however, performance will be limited. Memory requirements depend on how frequently data is to be logged, the duration of the log, and the number of variables logged. X also requires a fair amount of memory. Our PCs usually have 32 or 64 megabytes of RAM, though 16 megabytes should suffice. If the GUI is running on another PC or workstation, the PC running the control can have as little as 1 megabyte of RAM.

2. *MultiQ Board:* The Quanser Consulting MultiQ board is the interface to the controlled system. It provides 3 hardware timers, 8 A/D, 8 D/A, 8 digital I/O lines, and 6 quadrature encoder channels. One of the hardware timers is used to generate the control timer interrupt. Other interface boards may be used, however, the use of other boards requires development of an interface library. Development of the MultiQ interface library took us about one week.

3. *Ethernet Network Interface:* An Ethernet network interface card allows the GUI to be either displayed remotely via the remote display capabilities of X, or remotely executed on another hardware platform which supports native X.

Software Architecture

QMotor 2.0 is divided into four separate cooperating processes: QS (MultiQ server), QC (client – control program), QN (network interface) and QG (GUI). QS accesses the MultiQ board hardware, and handles the hardware timer interrupt. It virtualizes the MultiQ hardware by providing shared memory global variables (see Table 2) that QC can access. QS updates these variables each control cycle, in the timer ISR (see the next section).

Table 2. QS Virtualized MultiQ Hardware

MultiQ Hardware Device	Global Variables in Shared Memory
Encoders (6 channels)	int G_ENC[6];
A/D Converters (8 channels)	int G_ADC[8];

Digital Input Word (8 bits)	unsigned short *G_DIGIN;
D/A Converters (8 channels)	int G_DAC[8];
Digital Output Word (8 bits)	unsigned short *G_DIGOUT;

The user writes 5 functions (described below) that are embedded into QC. QN communicates with QG via TCP/IP internet domain sockets. QG allows the user to edit, compile, and execute the control program. It also provides for plotting in real-time and after the control run.

One Interrupt Cycle

Fig. 5 shows one complete interrupt cycle and how the four modules (QS, QC, QG, QN) interact. Once a control program is loaded and the control parameters such as frequency, duration, gains, etc. are set by the user, execution of the control program begins. One of the timers on the MultiQ board is programmed to generate an ISA bus interrupt every control period. We will now step through the sequence of events that takes place when the timer interrupt is triggered.

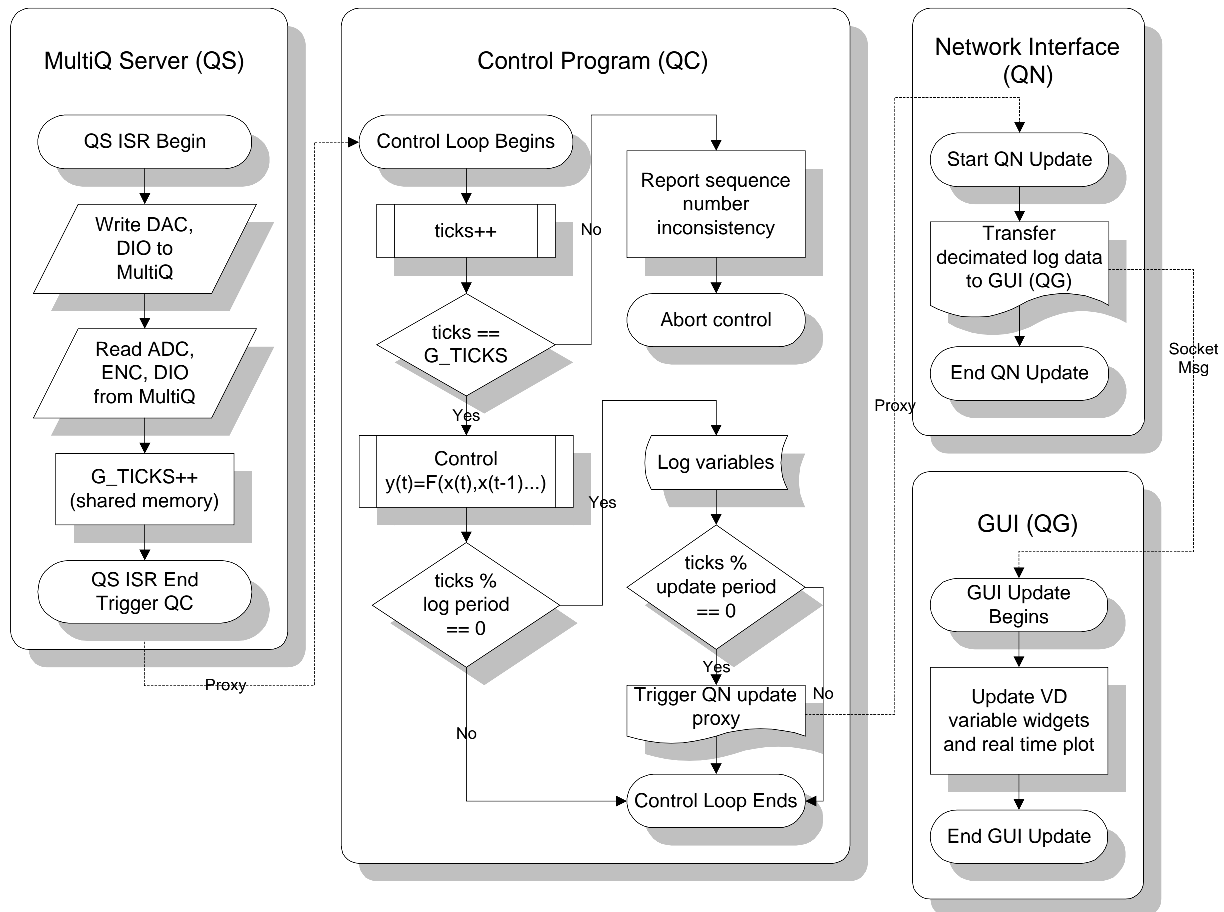

Fig. 5. One Interrupt Cycle

QS (QMotor MultiQ Server)

1. *QS ISR*: The ISR that has been attached to the MultiQ's timer IRQ begins execution. It preempts any executing process or lower priority hardware ISR.

2. *Hardware Output*: The output values written by QC during the last control period (*G_DAC[], G_DIGOUT*) are written from the global shared memory locations to the actual MultiQ hardware outputs.

3. *Hardware Input*: New values are read from the A/D channels, digital inputs, and encoder registers. These values are stored in the global shared memory variables where the control program accesses them (*G_ADC[]*, *G_DIGIN*, *G_ENC*).

4. *Increment Sequence Counter*: If the control frequency is too high, the ISR will begin to starve the control program. This condition is detected by a sequence counter that is stored in the global shared variable *G_TICKS*.

5. *Trigger Control Program*: Sends a non-blocking *proxy* message to wake up QC.

QC (QMotor Control Program)

6. *Receive Proxy*: Normally, the control program will be blocked waiting for a QNX message to arrive. If this message is from the proxy process set up to receive proxy messages from QS, the control program enters the main control loop. If the message is from another process, this indicates the control has been terminated abnormally (emergency stop, or some other exception condition).

7. *Increment Sequence Counter*: A local sequence counter is incremented. If the difference between this sequence counter and the global shared *G_TICKS* sequence counter has increased, a sequence error is reported and the control is terminated. This indicates that the control frequency is too high, and that the timer ISR has been starving the control program.

8. *Input*: The user's *input()* function is executed. Filtering of input signals can be performed here, along with unit conversion for the A/D and encoder channels. The inputs are read by simply referencing the global shared variables *G_ADC[]*, *G_ENC[]*, and *G_DIGIN*, which have been set by the QS ISR (see step 3 above).

9. *Control*: The control law is computed and generates the desired output variables.

10. *Log*: If this is a logging period, variables are logged. If this is an update period, QN is notified via a proxy that it should send the decimated log data to QG.
11. *Go to 6*: One iteration through the control loop is finished, so the control program blocks and awaits the next message.

QN (QMotor Network Interface)

12. *Transfer Decimated Log Data*: Data can be logged at any frequency up to and including the control frequency. Only a small fraction of this data is needed to maintain a real-time plot. To minimize overhead during the control run, only as much data as is needed to update the real-time plot and the displayed log variables is sent. Once the control terminates, the full log is transferred to QG.

QG (QMotor Graphical User Interface)

13. *Update Real-Time Plots/Displays*: QG is blocked and awaits a message from QN. If the message contains real-time plot data, QG updates the displayed log variables as well as the real-time plot. The other possibility is that an *end-of-run* message has been received from QN. This would indicate either that the control duration has expired or that an emergency stop has occurred.

This sequence is repeated each hardware timer interrupt period.

Using QMotor 2.0

Fig. 6 is a screenshot of the QMotor 2.0 GUI. In the upper left corner is the emergency stop button. Control parameters (control frequency, log frequency, etc.) are entered in the top section of the main window. The *start* button is also located in this section. The middle section is divided into two vertical panes. The left pane contains the *VF variables*. These are control gains and other variables that are used in the control program, which the user can modify from the GUI. The right pane contains the *VD variables*. These are variables that can be logged and plotted during the control run. The third section of the window contains the real-time plot. Variables selected for plotting during the control run appear in this section. The plot is auto-scaling along the Y axis (value) and the X axis (time). The bottom section of the main window is a message area.

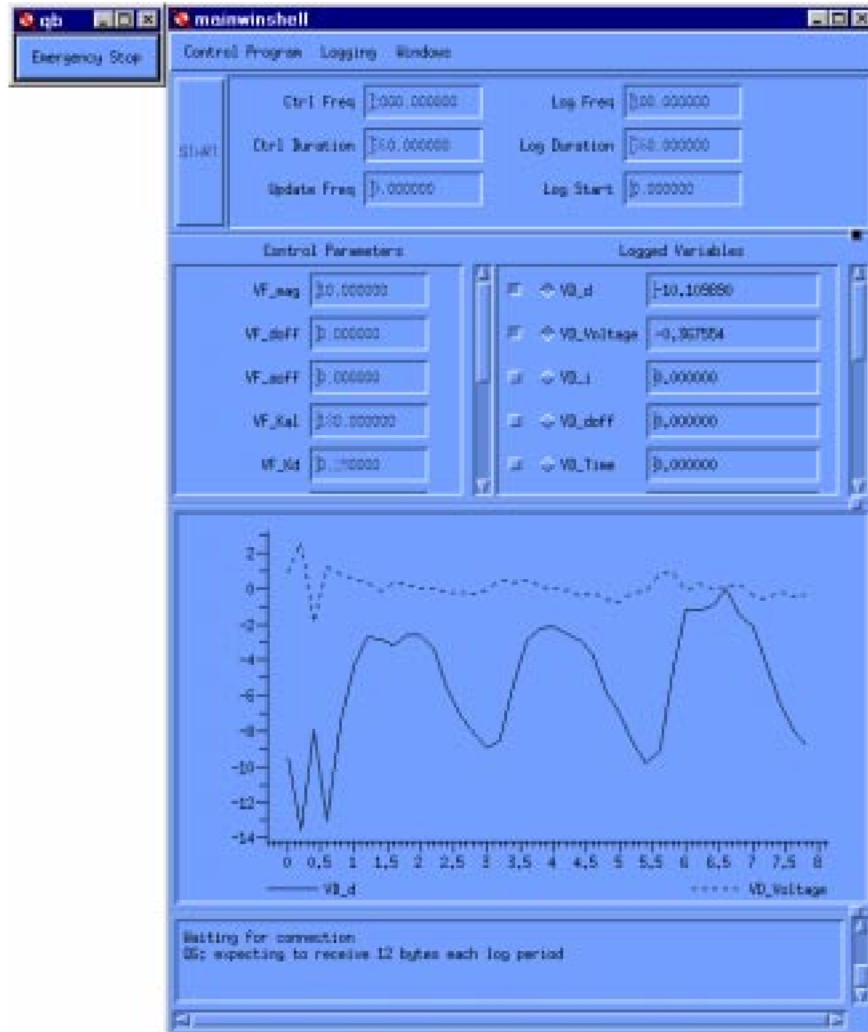

Fig. 6. QMotor 2.0 Graphical User Interface

A QMotor 2.0 control program is a C program written by the user. There are five C functions that the user must provide.

- *init_control()* is an initialization function. It is called every time the user hits the *start* button on the GUI.
- *input(), control(), output()*: The main control loop consists of the *input()*, *control()*, and *output()* functions. They are called in that order, each control period. The division into three functions is purely logical. Users generally perform their input (from encoders, A/D, etc.) and filtering in the *input()* function, control computations in the *control()* function, and output to D/A, etc. in the *output()* function.

- *end_control()* is called after the control run is ended. A control run will end after a pre-determined period of time (e.g. 30 seconds) or when the user hits the *stop* button on the GUI.

The control program is then compiled from the GUI. The standard QNX Watcom C/C++ compiler is used. This is the same compiler that was used to develop the GUI. Once all compilation errors are corrected, the control program is started by the GUI. The user can then modify the control parameters. Control variables (*VF variables*) can also be set by the user at this time. The control program is started by pressing the *start* button. It will now run at the control frequency until the control duration expires, or the user presses the *stop* button in the GUI. All of our hardware interface boxes have a hardware emergency stop button, which kills power to the amplifiers, etc., in addition to the software *stop* button in the GUI. During the control run, the variables selected for real-time plotting are plotted in the main window of the GUI. After the control run ends, the user may plot any of the variables that were logged.

Sample Control Program

The following sample control program reads an input voltage from A/D channel 3, multiplies it by a scale factor, adds an offset, and writes the result out to D/A channel 3. The user can change the scale factor and offset from the GUI at run time. Note that this program is extremely simple, and is intended for instructional purposes only.

```
#include "qc.def"
#include <math.h>
extern float tsamp;

/* Log Variables */
float VD_Time, VD_InputVoltage, VD_OutputVoltage;

/* Control Variables */
float VF_ScaleFactor, VF_Offset;

init_control() {
}

input() {
  /* Increment time */
  VD_Time=VD_Time+tsamp;
```

```
  /* Read ADC channel 3, convert from*/
  /* A/D hex integer value to */
  /* floating pt. voltage. */
  VD_Input=IToV (G_ADC[3]);
}


control() {
  VD_OutputVoltage=VD_InputVoltage *
                   VF_ScaleFactor +
                   VF_Offset;
}

output() {
  /* Write DAC channel 3, converting */
  /* from voltage to hex integer */
  G_DAC[3] = VToI (VD_OutputVoltage);
}

end_control() {
  /* Zero DAC 3 */
  G_DAC[3] = VToI(0.0);
}
```

Conclusions and Future Work

This paper has compared the traditional multiprocessor Host/DSP architecture used in control applications with QMotor 2.0, an implementation of the single-processor architecture. The single-processor architecture is simpler, cheaper, more flexible, easier to upgrade, and powerful enough to implement complex control algorithms.

QMotor 3.0 is currently under development. It will feature more graphing capabilities, object oriented design (to allow easy extension to more hardware), and support high control frequencies (in the 15KHz range, using a new PCI bus 200KHz A/D board). News about QMotor 3.0 can be found at *http://ece.clemson.edu/crb/qmotor/*.

The authors would like to thank the anonymous reviewer for all of the hard work and time spent in reviewing the original manuscript of this paper (CSM# 97-47). This reviewer generated several pages of constructive comments, including a suggested new structure for the paper, all of which were a great help in the rewriting of this paper. We appreciate his hard work and attention.

References

[1] K. Anderson, Integrator Backstepping Techniques for the Control of Brushless DC Motors: Theory and Experimentation, Masters Thesis, Clemson University 1994

[2] S. Lim, D. Dawson, and P. Vedagarbha, "Advanced Motion Control of Mechatronic Systems via High-Speed DSP and Parallel Processing Transputer Network", *Mechatronics - An International Journal*, Vol. 6, No. 1, pp. 101-122, 1996.

[3] S. Battilotti, and G. Ulivi, "An Architecture for High Performance Control Using Digital Signal Processor Chips", *IEEE Control Systems Magazine*, Vol. 10, No. 6, Oct. 1990, pp. 20-23.

[4] A. Jaritz, and M. W. Spong, "An Experimental Comparison of Robust Control Algorithms on a Direct Drive Manipulator", *IEEE Transactions on Control Systems Technology*, Vol. 4, No. 6, Nov. 1996, pp. 627-640

[5] H. Hanselmann, "Guest Editorial - Introduction to the Special Issue on Digital Signal Processors in Control", *IEEE Transactions on Control Systems Technology*, Vol. 2, No. 4, Dec. 1994, pp. 277-278

[6] M.S. de Queiroz, S. Donepudi, T. Burg, and D.M. Dawson, "Experimental Evaluation of Link Position Tracking Controllers for Rigid-Link Flexible-Joint Robots", *Proc. of the IEEE Conference on Decision and Control*, Kobe, Japan, Dec. 1996, pp. 4092-4097.

[7] D. Hildebrand, "An Architectural Overview of QNX", *Proc. of the Usenix Workshop on Micro-Kernels & Other Kernel Architectures*, Seattle, April, 1992.

[8] http://www.qnx.com/docs/qnx/sysarch/index.html

[9] M. Feemster, D. Dawson, P. Aquino, and D. Haste, "Position Tracking of the Induction Motor without Rotor Velocity or Rotor Flux Measurements", *Proc. of the IEEE Conference on Control Applications*, Trieste, Italy, Sept., 1998, pp 36-40.

[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 IEEE Conference on Decision and Control*, Tampa, FL, Dec 1998, accepted, to appear.

[11] M. de Queiroz, D. Dawson, and M. Agarwal, "Adaptive Nonlinear Boundary Control of a Flexible Link Robot Arm", *Proc. of the IEEE Conference on Decision and Control*, San Diego, CA, Dec. 1997, pp. 1327-1332.

[12] P. Vedagarbha, D. M. Dawson, and M. Feemster, "Tracking Control of Mechanical Systems in the Presence of Nonlinear Dynamic Friction Effects", *Proc. of the American Control Conference*, Albuquerque, NM, June 1997, pp. 2284-2288.