A Motor Control Framework for Many-Axis Interactive Robots

by

Matthew D. Hancher

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Science and Engineering

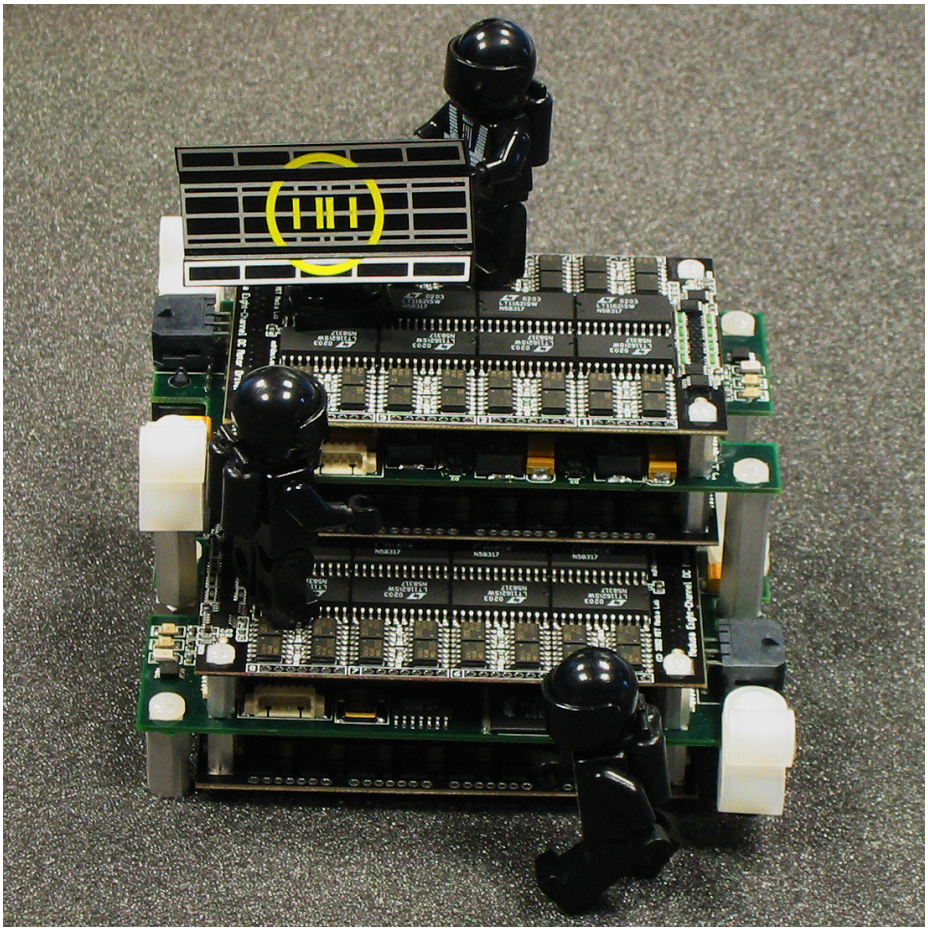
and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 20, 2003



A Motor Control Framework for Many-Axis Interactive Robots

by

Matthew D. Hancher

Submitted to the Department of Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology on May 20, 2003,
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A motor control system has been developed to address the specific needs of many-axis interactive robots. It is based on a modular collection of motor control hardware which is capable of driving a very large number of motors in a very small controller volume. These controllers support simultaneous absolute position and velocity feedback, allowing good dynamic performance without the need for a lengthy calibration phase at power-up. Example firmware has been developed which supports accurate position estimation and PD control to a continuously-updated target position. The control system is highly flexible, allowing alternative control algorithms to be developed with ease. A generic software library has also been developed to provide a clean interface between high-level control code and low-level motor hardware, as has a generic network protocol, known as the Intra-Robot Communications Protocol, which provides a simple and extensible framework for inter-module communication within a complex robot control system.

Thesis Supervisor: Cynthia Breazeal
Title: Assistant Professor of Media Arts and Sciences

Acknowledgments

This work would not have been possible without the help of many people.

Thanks to my thesis advisor, Prof. Cynthia Breazeal, for supporting me as a graduate student at the Lab, to Prof. Neil Gershenfeld for supporting me as an undergraduate before her, and to all my lab-mates past and present for making my time here so much fun.

Thanks to all my friends and family, and especially Vanessa, for enduring the crazy antics and long disappearances of a guy trying to finish his thesis.

Thanks to Stan, Richard, J.D., Lindsay, and everyone at Stan Winston Studio for taking a leap of faith and building us an incredible machine.

Thanks to Rick Ciliberto and Jon Blum for helping out with assembly and testing, and to Mark, Reuben, and everybody at SPS Tech for happily putting up with my small production runs and peculiar schedules.

Thanks to Patrick Kane and the Xilinx University Program for their generous support of students and faculty everywhere, and of this project in particular.

Lastly, thanks to Matthew Reynolds for teaching me what it means to be an engineer, and to Leila Hasan and Holly Gates for inspiring me to learn.

This work was supported by the DARPA MARS grant NAG-9-1443, by the NSF Center for Bits and Atoms grant CCR-0122419, and by the many sponsors of the MIT Media Lab who make this unique research environment possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The problem of giving a mechanical apparatus the appearance of life has inspired artists, engineers, and tinkerers for centuries. Their creations, from the early mechanical automata of Vaucanson, Jaquet-Droz, and others in the 1700s to the latest Hollywood animatronics, have been enormously successful as tools for entertainment [CD58, SH94]. In recent years much more serious applications for these technologies have emerged. As computers have become more powerful, the demands of human-computer interaction have increased. Users are regularly faced with an overwhelming array of choices and are given little help in deciphering them. One proposed solution has been to adopt a social model: rather than forcing users to learn sophisticated new modes of interaction, program the computers to interact with people in a natural social manner instead. A well-designed socially-interactive robot could then become the ultimate user-friendly interface [ABBS00].

Another application for socially-interactive robots is the study of social interaction in general. A robot can be programmed according to theoretical models of social behavior, and the veracity of those models can be tested by observing how people actually interact with it. Such a robot need not attempt to perfectly model the human form or human interaction. Rather, by modeling some elements and not others it may be possible to learn which factors are important to human interaction and which are not.

The Robotic Life Group at the MIT Media Lab was founded in large part to study robots from these dual vantage points of human-computer interaction and behavioral science. It constructs robotic creatures designed to engage and ultimately learn from the people with whom they interact. This requires developing new sensor and actuator

technologies, modeling social behavior and learning, and then integrating everything into a unified robot control framework. One key element in such a framework is a low-level motion control system; this thesis describes hardware and software that have been developed over the last two years to address the special motor control needs presented by interactive robots of this sort.

These motor control requirements are quite different from those presented by robots elsewhere. Industrial robots, for instance, typically have the minimum number of degrees of freedom required to perform some specific manipulation task. In that context a robot with six or eight degrees of freedom is considered highly articulate. More complex robots can be found in the research community, of which the most sophisticated are the humanoids. Several research institutions have constructed such robots, including the MIT AI Lab's *Cog* [BBM+98], NASA's *Robonaut* [AAA+00], Honda's *ASIMO* [SWA+02], and Sony's *SDR-3*. Each of these has on the order of thirty degrees of freedom except *Robonaut*, which has extremely dextrous hands that bring the total number of joints to almost fifty. These robots are used primarily as platforms for research in motion planning, learning, adaptive control, and simple human-robot interaction. However, none of these robots is capable of life-like expressive motion.

There are two classes of robots more closely related to the present work. On the one hand there there are robots designed to study subtler forms of social interaction, most notably the AI Lab's *Kismet* [Bre02]. *Kismet* is not capable of full-body motion like the larger humanoids, but instead possesses a rich set of social behaviors that take advantage of its highly expressive face and head. On the other hand there are the most lifelike robots of all, those designed by visual effects studios such as Jim Henson's Creature Shop or Stan Winston Studio specifically for the purpose of looking convincingly alive [SH94, Bac97]. Many of these robots posses even more degrees of freedom than the research humanoids, but they are not capable of autonomous operation. Instead, teams of human puppeteers drive these robots in real time or with simple motion playback systems. The robots of the Robotic Life Group aim to bridge this gap between small expressive autonomous robots such as *Kismet* and tele-operated Hollywood animatronics.

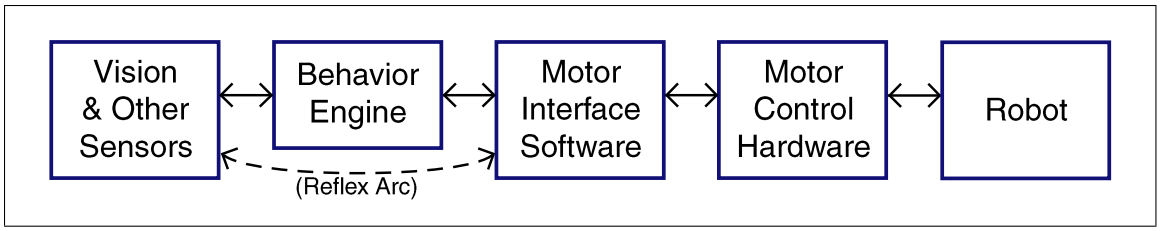


Figure 1.1: Generic robot control structure, emphasizing the motor control components.

1.1 Interactive Robot Control Overview

The more complex robots of the Robotic Life Group all share a common high-level control structure. At the core is a behavior engine developed in conjunction with the Media Lab's Synthetic Characters Group. It is based on the Synthetic Characters C4 codebase, an extremely flexible set of Java character control tools originally created to drive that group's interactive on-screen animated characters. This framework includes generic support for a variety of models of perception, learning, and behavior. It also contains a motion control system which generates appropriate motion data through animation blending. In this scheme professional animators generate a number of sample animations using a 3D robot model. This data set is then interpolated to produce new animations in response to user interaction. Though this system was not originally designed for robot control it is nevertheless quite useful, as many of the issues encountered in interactive robotics are very similar to those that arise in advanced computer graphics [KB99, Joh02]. NASA has used a similar scheme to control *Robonaut*, using basis trajectories generated by tele-operating the robot to perform various tasks.

The basic system structure is shown very simply in Figure 1.1. The behavior engine, represented by a single block in this diagram, in fact consists of several subsystems, including a behavior system as well as a perceptual system and a motion system. This ensemble resides between the sensors and the motors and continuously computes the motor trajectory based on sensor inputs. This motion data is sent to the motor interface software which is in turn responsible for commanding the actual motor control hardware to make the robot move as requested. This mode of operation is quite different from that found in other branches of robotics. For example, in industrial robotics it is customary for a trajectory to be completely pre-computed

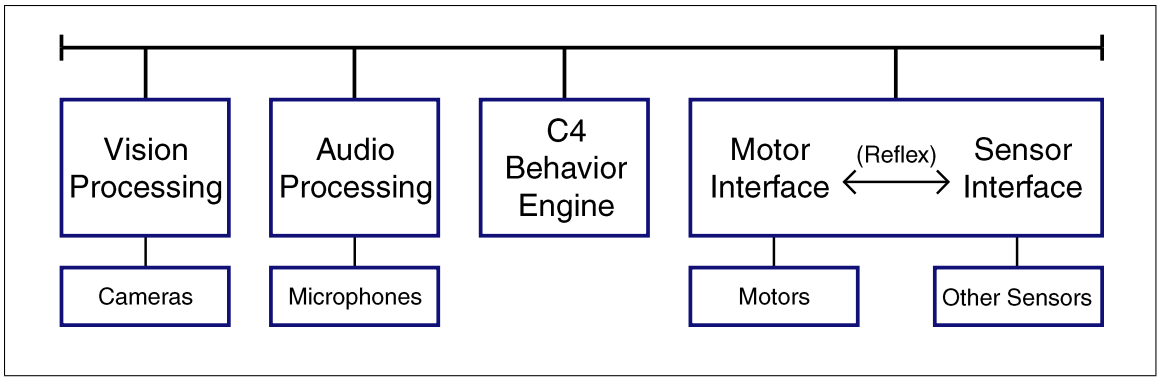


Figure 1.2: A typical division of control tasks across multiple computers. Inter-computer links are the greatest source of latency other than complex sensor processing such as machine vision.

before motion begins. The low-level motor system performs the specified motion and comes to a halt at the end to wait for its next command. This structure makes it easy to optimize the trajectories for some desired quality, such as high accuracy or low power consumption, but is unfortunately entirely inapplicable to interactive robotics. In order to appear life-like an interactive robot must be able to respond to some stimuli nearly instantaneously, both to permit fast reflex responses and to allow tight closed-loop control based on perceptual feedback, and so the motor system cannot ever commit the motors to a pre-specified trajectory.

The Synthetic Characters behavior engine's motion system generates poses at a rate of 20–60 per second. These poses are passed to low-level interface software either through the Java Native Interface (JNI) or via network sockets. The poses may be modified at this stage to enforce constraints such as joint limits or maximum joint velocities, and the data may be up-sampled to provide a higher controller update rate. This up-sampling is particularly important if the behavior engine's update rate is on the low side, because otherwise the discreet updates result in visibly jittery motion. The network sockets interface is considerably more flexible than JNI because it allows the low-level software and behavior engine to be located on different computers. However this introduces network transmission latency into the control loop. To get around this, the low-level sensor interface software should located on the same computer as the motor interface software; these modules can then be programmed to directly implement high-speed reflex actions. This structure, with fast reflex responses initiated by one system and slower responses that require more

computation handled by another, closely resembles the structure found in animals and should make it possible to generate convincing interactive behavior even if some control paths have non-negligible latency. A typical division of labor between multiple computers based on this design philosophy is shown in figure 1.2.

1.2 Low-Level Motor Control

This thesis describes the Medusa line of motor controllers, developed by the Robotic Life Group to fill an important gap in the range of DC servo controllers. Three motor control requirements which are critically important to the Robotic Life Group's interactive robots guided this design. The first is controller density, the number of joints which can be controlled in a given controller volume. This is important because the number of motors in each robot is extremely high. It is not always acceptable to run wires from each of a robot's motors to an external control box, since the associated mass of cabling can restrict the robot's motion and ruin the aesthetic effect. The controllers must therefore be extremely small so that they may be embedded directly inside the robot. The second motor control requirement is support for continuous updates. The controllers must be able to track continuously-varying target positions seamlessly with low latency and small variation in latency across a large number of joints. Some commercially-available systems enforce a trapezoidal motion profile, for instance, and cannot be interrupted until a motion is complete. The last key requirement is support for absolute position feedback. Without this a motor control system must calibrate its motors' absolute positions each time it is turned on, a potentially lengthy and extremely annoying procedure.

No commercially-available control hardware is able to meet these three requirements simultaneously. The detailed specifications of the Medusa controllers are discussed in Chapter 3, but Table 1.1 compares these controllers to a number of alternatives in general terms. Unfortunately, little information is available about the motor control systems used by robots elsewhere, either because they are proprietary (as in the case of robots developed at corporate research institutions) or because they are simply not the focus of research and hence of publication.

Standard industrial controllers are unacceptable first and foremost because they are too bulky to be embedded inside small expressive robots. One of the most compact of the commercial servo controllers is the Animatics RTC-3000, which occupies

Controller Type	Continuous Update	Volume/ Channel	Absolute Position	Max Current	Input Voltage	Single Supply	Interface
Animatics RTC-3000	Yes	2.3 in^3	No	1A	24V	Yes	RS-232 RS-485
ICD Motovator LVDC	Yes	432 in^3	Yes	3.5A	12–48V	Yes	DMX
Logosol LS-173AP	No	14 in^3	Yes*	8A	12–90V	Yes	RS-485
J.R. Kerr PIC-SERVO	No	10 in^3	No	3A	12–48V	No	RS-485
Hobby Servos	Yes	—	Yes	≈1A	4.8-6V	Yes	PCM
Custom distributed	Yes	2–4 in^3	Yes	any	any	any	any
Medusa (Single-Port)	Yes	0.9 in^3	Yes	5A	5–30V	Yes	RS-232
Medusa (Dual-Port)	Yes	0.5 in^3	Yes	5A	0–30V	No	RS-485

*Limited support only.

Table 1.1: Comparison of various motor control technologies.

2.3 in^3 and can drive a DC motor with up to 1A continuous current. The motor controllers described in this thesis support over four times as many motors per unit controller volume, and can provide up to five times as much continuous motor current. Moreover, the Animatics controller, like many commercially available systems, does not support absolute position feedback.

There are some commercial controllers which can be used with absolute position sensors as well as the more traditional velocity encoders. The Intelligent Control Devices Motovator LVDC module is one such controller. Designed for the entertainment industry, this DMX-controlled DC servo controller has specifications very similar to the hardware described in this thesis. Unfortunately each controller occupies a 6"×12"×6" volume per motor! Such a system is completely inappropriate for use with compact many-axis robots. Another commercial controller, the Logosol LS-173AP, is promising in many respects. However close examination reveals that this controller does not take full advantage of the redundant absolute and relative position sensors. Rather, it is only capable of moving at a controlled velocity until the absolute sensor returns a given value. If the sensor is noisy, as potentiometers generally are, this could result in unpredictable behavior. More importantly, this scheme is not compatible with the need for continuous updates.

All other commercial controllers are inappropriate for one reason or another. One possible solution is to use motors with integrated controllers. Many such systems exist

for industrial applications, but these are too bulky to use here. The hobby industry, however, offers a wide range of small integrated packages designed to move the control surfaces on remote-controlled airplanes and boats. Some animatronic creatures, such as the extremely successful robot Teddy produced by Stan Winston Studio and seen in the movie *A.I.: Artificial Intelligence*, have been built using this technology. However, these servos do not allow for the precise control required for object manipulation or machine vision with eye-mounted cameras. In addition they are quite loud, which can ruin the effect of otherwise life-like interaction. For these reasons, standard high-quality DC motors with external controllers are to be preferred.

The hardware presented in this thesis is optimized for controlling a large number of relatively small DC servo motors. In each of the variations a single processor is used to control multiple channels. One alternative architecture which was considered would have used extremely compact single-channel controllers located throughout a robot immediately adjacent to the motors they control. Such a distributed control system could certainly be made to work, but the controller density could never be made as low as is possible with centralized control. The volume associated with the processor, power conditioning circuitry, and connectors at each control board would double the total volume. The most compact version of the controllers described in this thesis, the S-model dual-port controller, can control sixteen motors in a volume of only 0.5 in^3 per motor. Designing a flexible single-channel controller in that volume would be virtually impossible. One advantage to the distributed control technique is that it allows a simple wiring scheme in which a single daisy chain connects all the motors together. However, this too would come at a price: there are far more single points of failure in such a scheme than in the one used here, potentially reducing the overall system reliability. The length and aggregate resistance of a daisy chain with many motors could also introduce significant noise into the power supply and communications lines, decreasing reliability still further.

1.3 About This Text

This document is intended in part as an introduction to the Medusa hardware and related software to be read by future engineers who will be using them. However, there is not nearly enough space here to provide complete documentation, and the firmware and software are being constantly updated. A website is currently being assembled

to provide thorough, up-to-date information about this motor control system. The website can be found at `http://robotic.media.mit.edu/motor/`.

In Chapter 2 we will describe the robots for which this control system was originally intended. The most challenging is a new humanoid robot known as *Leonardo* which was developed in collaboration with Stan Winston Studio. The particular needs of these three robots informed the design of the control hardware, which is described in detail in Chapter 3. This hardware contains many programmable devices; the firmware that was developed for them is discussed in Chapter 4. Such hardware is of course of no use without a mechanism for high-level control software to communicate with it. A general motor control software library and a network interface are described in Chapter 5. The network protocol, known as the Intra-Robot Communications Protocol, was designed as a general-purpose protocol to be used throughout an interactive robot. Chapter 6 offers concluding thoughts and suggests future directions for this work.

Chapter 2

The Robots

The motor control system described in this thesis was designed both as a general framework for future development and as a solution to specific problems presented by three particular robots. In this chapter we will discuss these robots, known as *Public Anemone*, *Leonardo*, and *RoCo*. The first two of these provided important testbeds for early prototypes, and we will discuss those experiences here as well.

2.1 Early Prototypes: Public Anemone

Public Anemone was the first major project of the Robotic Life Group. It was conceived as an exploration in non-anthropomorphic interactive robotics, and was displayed at the Emerging Technologies exhibition at SIGGRAPH 2002 in San Antonio. The robot consists of a tentacle-like body with five smaller tentacle-like fingers and was covered in a synthetic silicone skin for the exhibition, giving it an organic appearance.

The original design called for the *Anemone* to be submerged in a large vat of oil with a layer of glowing goo at the bottom for the robot to play with. The body and fingers were remotely cable-driven in this design, avoiding problems associated with operating motors in oil. However, an early prototype revealed that the cable drive system did not allow sufficiently precise control over the arm's motion, although the movement had a excellent smooth quality. A completely new mechanism was designed which, it was hoped, would improve the controllability of the body without sacrificing its smooth motion.

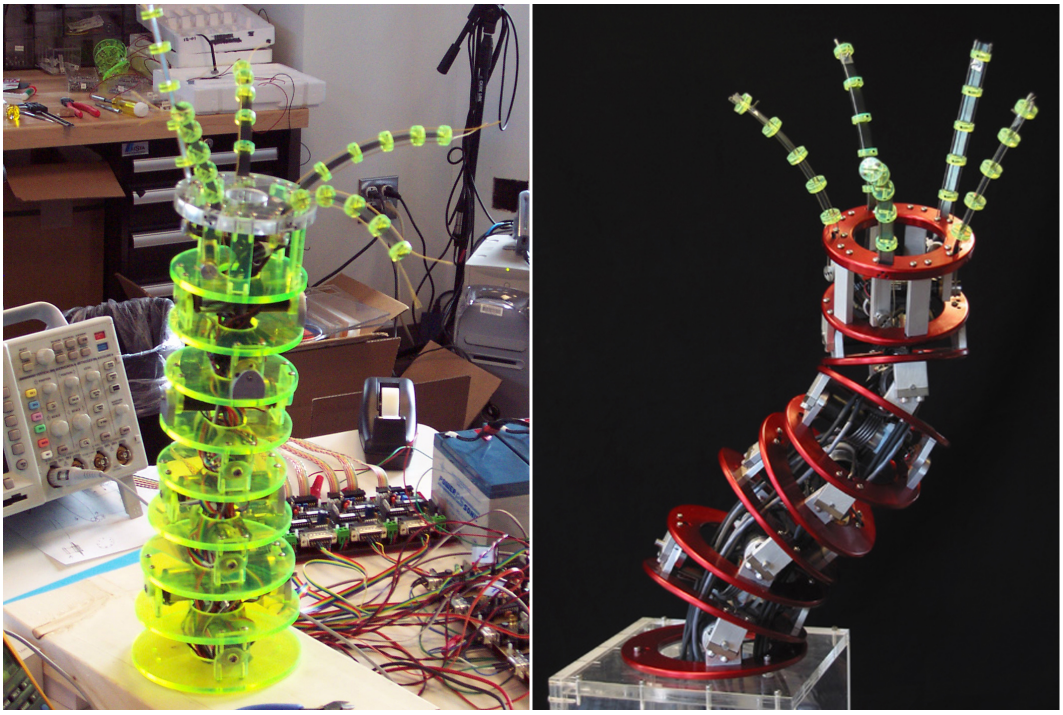


Figure 2.1: Left: Prototype of *Public Anemone*, with assorted prototype control electronics. Right: Final version of *Public Anemone*, as shown at SIGGRAPH 2003.

The new design was based on an eight-stage segmented arm with embedded DC gear motors directly driving each stage. The fingers were still cable-driven, but the drive motors were moved to the tip of the arm, thereby minimizing cable length. Each of the direct-drive motors was outfitted with a potentiometer to measure its position, while the finger motors had traditional quadrature encoders. Encoders were not used in the direct-drive stages for two reasons. First, the encoders that were available at the time added considerable length to the motors and could not fit in the desired robot volume. Second, using encoders as position sensors would have required a calibration phase at power-up (since quadrature encoders can't sense absolute position) and would have left the body joints susceptible to encoder drift.

The challenge of this design was developing a control system capable of achieving a life-like quality of motion. A commercial single-channel motor driver, the J.R.Kerr PIC-SERVO controller, was chosen to drive each of the fingers. These boards feature an RS-485 daisy-chain communications system which appeared to make them quite flexible. However, they only support encoder feedback and so could not be used to control the *Anemone*'s body. Therefore the first of the custom Robotic Life

motor controllers was designed, closely based on the design of the PIC-SERVO but with a 16-bit A/D converter in place of the quadrature decoding hardware. Five of the J.R.Kerr boards and eight of the custom boards were used to drive the *Public Anemone* prototype, shown at left in Figure 2.1.

This design suffered from a number of major problems. First, the body motors required large gear-down ratios so they could generate enough sufficient torque to lift and move the hefty *Anemone* body. Planetary gear boxes of this sort have a backlash region on the order of two degrees wide. As a result, when the *Anemone* passed near the vertical configuration several motors in series would pass uncontrolled from one side of the backlash region to the other. This problem was exacerbated by the fact that the data from the potentiometers was so noisy that the position control gains had to be kept quite low. Electronic damping was virtually impossible, since this requires a velocity measurement which could only be obtained by differentiating the already extremely noisy pot signals. The noise was caused both by the pots themselves—most potentiometers have terrible dynamic performance—and by the motor PWM signals coupling to the adjacent high-impedance pot return lines.

To address the backlash problem torsion springs were added at each stage to preload each joint, forcing the gear box to one side of the backlash region at all times. The controller performance was also improved with the addition of quadrature encoders to each motor. These Maxon MR-series encoders are based on a new magneto-resistive technology which is more sensitive than the traditional Hall-effect sensing technique and consequently permits smaller magnets and a considerably thinner total package. The encoders were installed in addition to the existing potentiometers, which were left in place to allow the robot to determine its absolute position at all times. The final mechanism, incorporating these changes and re-built in aluminum, is shown at right in Figure 2.1.

Another important lesson that came out of the early experiments is that the complexity and overhead associated with using a separate microprocessor for each motor can become problematic as the number of motors becomes large. A more sophisticated communications system could certainly address this problem successfully, but the simple RS485 system used by the J.R.Kerr boards turned out not to be particularly scalable. It was designed for occasional position updates in an industrial environment, and with a large number of channels it could not support the constant trajectory updates that were required.

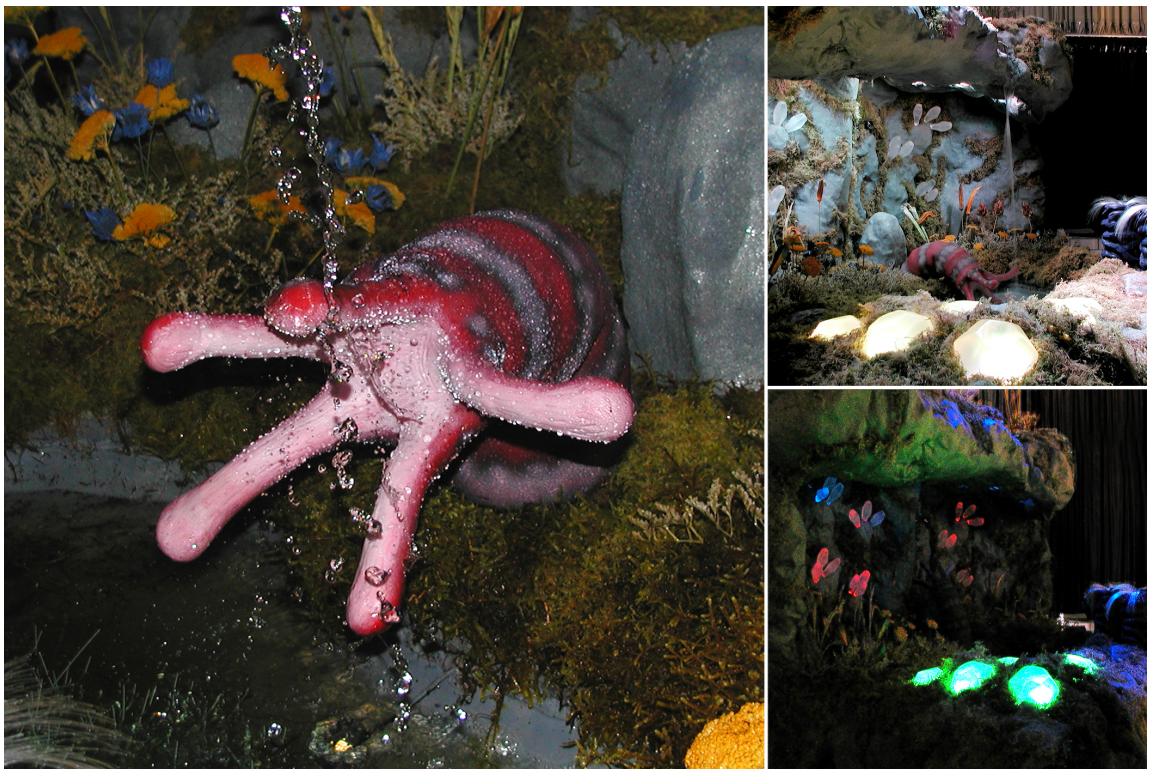


Figure 2.2: Left: *Public Anemone*, with its red silicone skin, playing in the waterfall at SIGGRAPH 2002. Above Right: Full view of the SIGGRAPH terrarium. Below Right: The "night" phase of the terrarium's five-minute cycle.

One more problem arose with the exhibit design: neither the *Anemone*'s potentiometers nor its motors were able to operate properly for extended periods while submerged in oil. Since there were also obvious logistical concerns surrounding the maintenance of an oil-submerged robot, the oil tank was eliminated from the exhibit design in favor of a cave-like terrarium. This environment was also populated with a number of smaller interactive creatures which operated during a "night" phase once every few minutes. The *Anemone* went to sleep during this phase, allowing its motors to cool down and thereby helping it sustain eight hours a day of continuous operation for five days at SIGGRAPH. The final terrarium as shown in San Antonio is pictured in Figure 2.2, featuring the *Public Anemone* with its red silicone skin.

In order to control the final *Public Anemone* a control system was needed that could take proper advantage of the redundant data from the potentiometers and quadrature encoders. In addition, it needed to support a position update rate greater than 30Hz, and preferably twice that, ideally using a standard PC serial port. This

meant that the serial system could not use a poll-and-response protocol such as the J.R.Kerr protocol to communicate with each individual joint; the I/O delays in the computer would limit such a system to around a 10Hz update rate for the eight-stage body. An integrated eight-channel controller was developed to address these problems. It was based on the same design principles as the Medusa hardware described in the next chapter, and served as an early prototype for that line. The J.R.Kerr PIC-SERVO was still used to control the fingers, but with custom firmware that better supported continuous position updates. Unfortunately the quadrature decoding hardware on those boards is particularly susceptible to encoder drift, and this was a recurring problem at SIGGRAPH. In the near future *Public Anemone* will be re-installed as a research robot at the Media Lab and will operate under the full control of the hardware described in this thesis.

At its highest level, the robot was controlled by the Synthetic Characters C4 behavior engine described in Chapter 1. Professional animators were hired to produce a variety of motion animations covering the full range of desired behaviors. As discussed, the control system then used an animation-blending scheme to produce novel motions based on the provided key-animations. The uppermost two body stages were controlled using a separate procedural algorithm that allowed the *Anemone* to orient accurately towards people in the audience. At SIGGRAPH the robot's only inputs were two stereo camera pairs, one located above the terrarium and one hidden in the cave behind. These gave the behavior engine information about the location of the people observing the robot, which the robot used to look at them or to back away in fear if people got too close. When it was not tracking on-lookers, the robot performed a a variety of idling tasks, such as playing in the terrarium's waterfall or watering the nearby flora.

In this installation the behavior engine's motor system communicated with the low-level motor control software via the Java Native Interface (JNI). This was a relatively efficient means of communication, but it required the behavior engine to be fully reloaded for each change in the low-level motor system and required the motor system and control hardware to be shut down for each change made to the behavior engine. This represented a significant impediment to rapid system development, particularly since the Synthetic Characters behavior engine takes a considerable amount of time to initialize. The network-based communications protocol described in Chapter 5 was originally developed primarily as a solution to this problem.

Figure 2.3: *Leonardo* with almost complete skin and fur.

2.2 Primary Application: Leonardo

Much of the design of the Medusa hardware described in this thesis was inspired by the needs of one robot, *Leonardo*. *Leonardo* is a humanoid robot with passive legs being developed in a collaboration between Stan Winston Studio and the Robotic Life Group. Unlike *Public Anemone*, *Leonardo* was designed for traditional social interaction. Stan Winston Studio is a Hollywood visual effects studio with expertise in building convincing live-action animatronic characters. However, their characters are teleoperated and have no autonomous behavior capability. They designed the look and mechanism of *Leonardo* and provided him to the Media Lab for use as a testbed in autonomous character control.

Leonardo is a young and somewhat mischievous-looking furry creature standing roughly two feet tall, with a highly expressive face and upper body. The robot features 61 distinct points of motion, including 32 in the face alone. This makes *Leonardo* the most sophisticated such expressive humanoid robot in existence. He is covered with a silicone- and foam-based skin and a mix of furs from real animals, all crafted using

the most advanced Hollywood special effects techniques. *Leonardo* is not a mobile robot: his body is fixed to a platform, and a number of degrees of freedom are cable-driven from beneath. He was instead designed to push the boundaries of life-like robot behavior in close interaction. Figure 2.3 shows Leonardo with almost finished skin and fur; the hands in particular are incomplete in this picture.

This research is of great interest to the entertainment industry because complex animatronic characters require so many puppeteers to operate that improvisation is essentially impossible. Each sequence of moves must be precisely rehearsed many times so that all parts of the robot move in concert. In the future an autonomous behavior system could be used to control the robot's basic functions while a single puppeteer issued high-level commands. For instance, the puppeteer might direct the robot's attention by selecting an object of interest on a screen, and the robot would then be capable of moving its eyes, head, and body in a lifelike way to look at the object. If the object moved machine vision could be use to track it, freeing the puppeteer from the need to make continuous adjustments. This is but one of many possible scenarios in which a computer-controlled animatronic could allow for a more expressive and improvisational performance, either on the set or in any other interactive setting.

The Robotic Life Group is primarily interested in *Leonardo* as a tool for research in human-robot cooperation. This research extends the work done by Prof. Brezeal and others at the MIT AI Lab, and especially the work on the expressive face and head robot known as *Kismet*. This research is in collaboration with the NASA *Robonaut* project, which is concerned with developing a robotic assistant for astronauts in space. A comfortable, intelligent social interface could make such a robot vastly more efficient for its human teammates. Each of the institutions collaborating on the *Robonaut* project has an upper-body humanoid of some sort, either real or simulated; the Robotic Life Group's focus is social interaction and cooperation, and so *Leonardo* is the perfect platform for this work.

Each of *Leonardo*'s 61 degrees of freedom is controlled by a DC servo motor with a magneto-resistive quadrature encoder and a shaft potentiometer. This design was inspired by the earlier experiences with *Public Anemone*. Some of these motors are embedded in the robot and directly drive the joints, while others are located in the base and drive the joints remotely with cables. This mechanism can be seen in Figure 2.4, which shows *Leonardo* with most of his skin and fur removed. The robot

Figure 2.4: *Leonardo* with his skin and fur removed, revealing the internal mechanism.

features many degrees of freedom that are not normally found in humanoid robots. For instance, in addition to two rotational degrees of freedom each shoulder is capable of a linear "shrug" motion. This ability would be irrelevant in an industrial robotic arm, but is it crucial to generating convincing life-like arm motion. The robot can tilt and lean at both the base and the upper torso, and can also rotate and sway side-to-side, giving the lower body considerable life and freedom despite being firmly attached to the base.

The most expressive part of the robot by far, however, is the face and head. The eyes, eyelids, and brows are all fully actuated, and the ears can move around, perk up, and fold down. More importantly, seventeen degrees of freedom are reserved for the mouth and surrounding face. This provides enough flexibility to generate most of the motions associated with normal speech; only fricatives cannot be faithfully reproduced. Moreover, all these degrees of freedom combine to give *Leonardo* tremendous range of emotional expression.

Thirty of the degrees of freedom in the head are directly-driven. If the controllers for these motors were placed in the base, then thirty sets of motor and sensor cables would have to be passed through the neck. This is not even remotely feasible; rather, these motor controllers must be embedded inside the head as well. No existing commercial controllers meet these extreme density requirements. Fortunately, most of these joints do not bear a heavy load, and so high-power drivers are not required. Meeting this demand was a major driving force behind the development of the hardware described here.

Because *Leonardo* has a more complex low-level motor system than *Public Anemone*, he presents greater programming challenges. For instance, his motors are driven by four separate motor control packages, two in the head and two in the base. The unified motor control software library described in the first half of Chapter 5 was originally developed to provide a clean programming interface to the behavior system despite this fact. This library shields the high-level motion code from the details of maintaining the low-level hardware and coordinating motion across multiple control modules. In *Leonardo* this code runs on a separate computer from the high-level behavior code, one which will also eventually host a large array of sensors measuring parameters such as skin pressure or proximity. This will allow some relatively fast reflex-like behaviors which could either help protect the robot or enhance the quality of interaction.

Much like *Public Anemone*, *Leonardo* is controlled at the highest levels by a modified version of the Synthetic Characters C4 behavior engine. Professional animators have created an animated model and a range of example animations for use with the animation blending system. Other motion generation systems are being developed, including a motion capture suit and a phoneme-based lip-synchronization system. Animation blending will not be the only control technique used with *Leonardo*; a number of procedural control schemes will be used as well. Cameras located in the eyes will allow him to accurately locate and track objects in his visual field. Precise arm and body motions generated from inverse-kinematic models will be required for simple object-manipulation tasks. Designing a common framework to support these tasks while incorporating blended animations for expressivity is one of the primary immediate research projects for *Leonardo*.

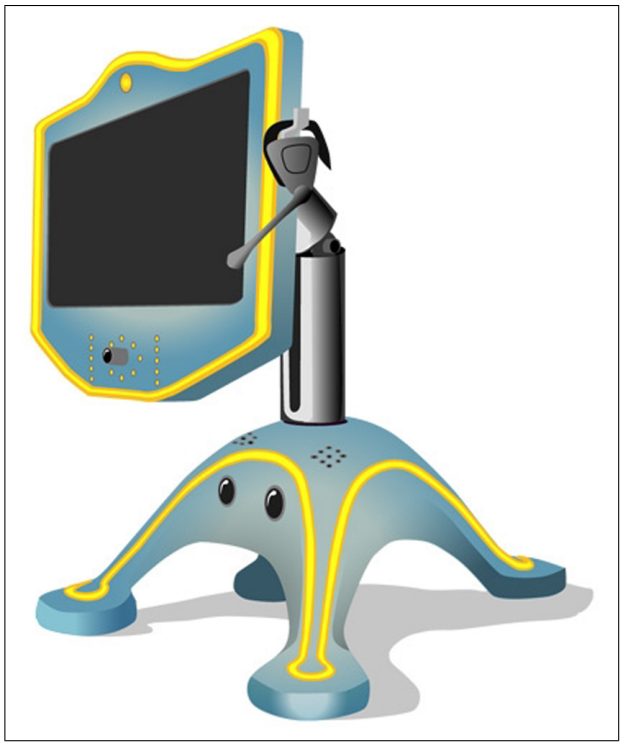


Figure 2.5: Artist's rendition of *RoCo*, still in the early design stages.

2.3 RoCo, A Robotic Computer

A third robot which is still in the design stages nevertheless played a part in guiding the design of this motor control system. It is known as *RoCo*, short for "Robotic Computer". Inspired by Apple advertisements featuring an animated *iMac* computer interacting with passers-by, it will consist of a flat-panel LCD display on a robotic arm. An artist's rendition of the robot, showing its basic structure, is shown in Figure 2.5. Though the mechanism is still under construction, the robot's behaviors have been explored using a 3D model.

This robot has fewer degrees of freedom than than the other robots discussed here. However its control system must be small enough to be embedded inside the controlling computer; a rack of adjacent equipment would spoil the effect. A small, easy-to-use general purpose motor driver package, designed to control up to eight degrees of freedom, was developed to address the needs of simpler robots such as this one.

Chapter 3

Medusa Hardware

The robots described in the previous chapter have somewhat unusual motor control demands which existing motor control systems, such as those discussed in Chapter 1, cannot properly address. Both *Leonardo* and, in the ideal configuration, *RoCo* need motor drivers small enough to be embedded entirely inside them. While the number of motors in each robot is relatively high, especially in *Leonardo*, the demands placed on those motors are relatively low. For instance, many of *Leo*'s joints are purely expressive and see hardly any load at all. Traditional industrial motor controllers are much larger than *Leo*'s volume constraints permit and much more powerful than required.

Somewhat smaller commercial feedback controllers do exist, but they have other failings. For instance, most use quadrature velocity encoders attached to the motors as the only position sensors. While encoders are excellent velocity sensors, they have one important drawback as position sensors: they do not know their position at power-up. Robots based on this design must use some special technique to determine or reset their configuration when they are first turned on, often by gently driving each motor until it reaches a mechanical stop or triggers a limit sensor. A complex robot, such as the MIT AI Lab's *Cog*, may spend several minutes calibrating itself each time it is turned on or reset. To avoid this, the robots for which the hardware described here was designed use a combination of encoders and potentiometers. With these redundant sensors the robots can achieve the good dynamic performance that is possible with encoder feedback but can also determine their position virtually instantaneously.

Position Sensing	Combined potentiometer and encoder
Channel Density	Less than 1 in^3 per channel
Motor Voltage Range	5V–24V
Max Continuous Motor Current	5A
Peak Instantaneous Motor Current	50A
Target Position Update Rate	Greater than 100Hz
Controller Update Rate	Greater than 1kHz
PWM Frequency	Greater than 30kHz (inaudible)
Mode of Operation	Continuously updated target positions

Table 3.1: Summary of Medusa controller specifications (all versions).

Unfortunately only bulky, highly-configurable industrial control systems are capable of supporting this non-standard feedback design. Since no existing motor controllers simultaneously offer the high densities and controller flexibility these robots require, a line of custom high-density controllers was designed specifically to control them. The most challenging design requirements are provided by *Leonardo*, and in particular by *Leonardo*'s head. There the space constraints are so tight that the motor controllers must be extremely compact, with wires coming out of all sides. This image, of a motor controller with wires flowing out snake-like in all directions, gave this series of controllers its name: Medusa.

In this chapter we will describe the hardware that has been developed to date for the Medusa motor controller line. We will begin with a general overview of the system design, and then consider in detail each of the components that has been designed so far. A single set of core specifications, listed in Table 3.1, guided all of these designs. However the three motor control packages that can be assembled from these boards are each tailored to a specific set of needs. One, designed for general use in small robots, is pictured in Figure 3.1. The second, designed for expressive robots with more degrees of freedom, is shown in Figure 3.2. The last package, designed specifically to meet the high control density requirements inside *Leonardo*'s head, is shown in Figure 3.3.

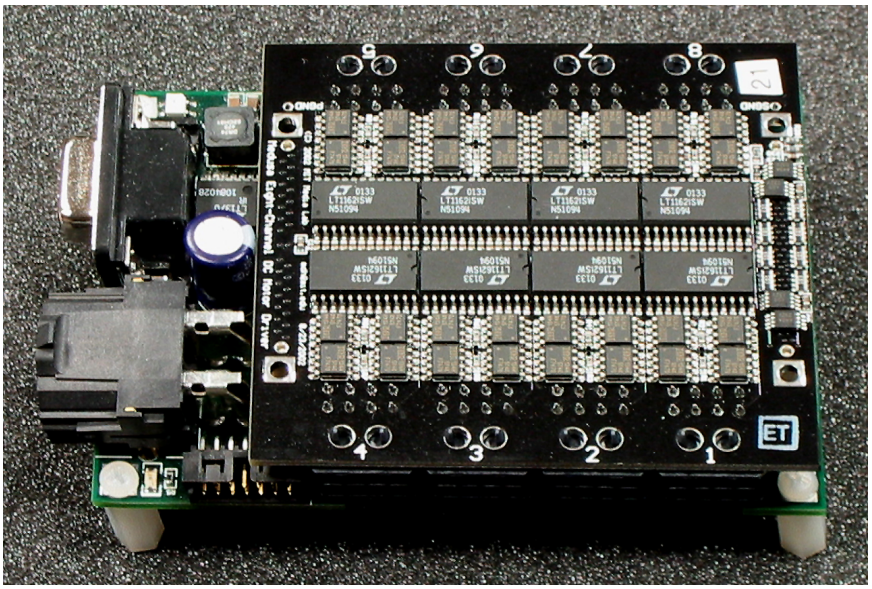


Figure 3.1: A Medusa PIC-based single-port control board with an eight-channel motor driver card attached.

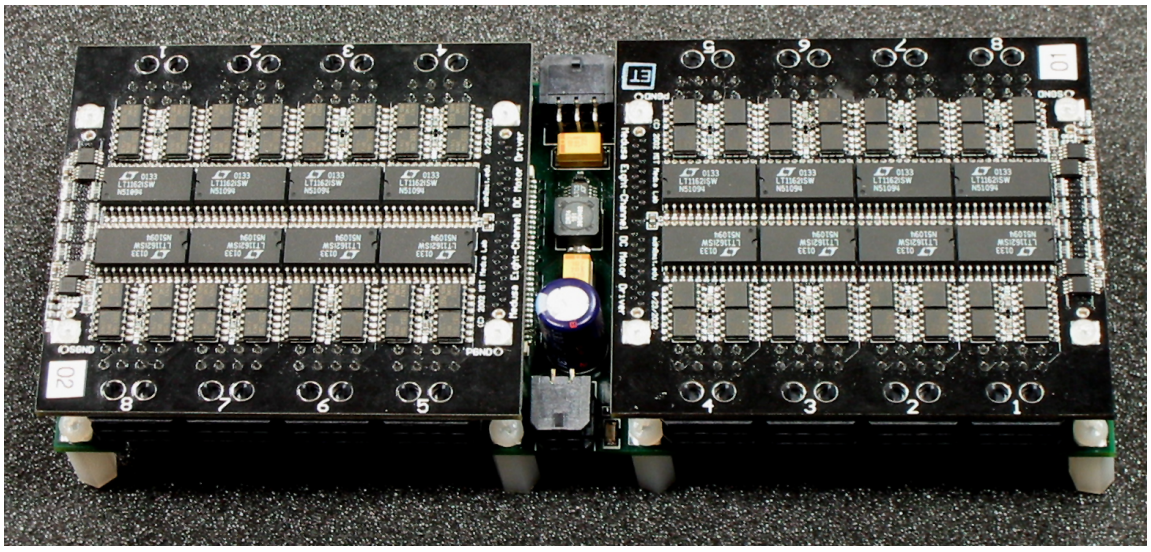


Figure 3.2: A Medusa FPGA-based dual-port control board with two eight-channel motor driver cards attached.

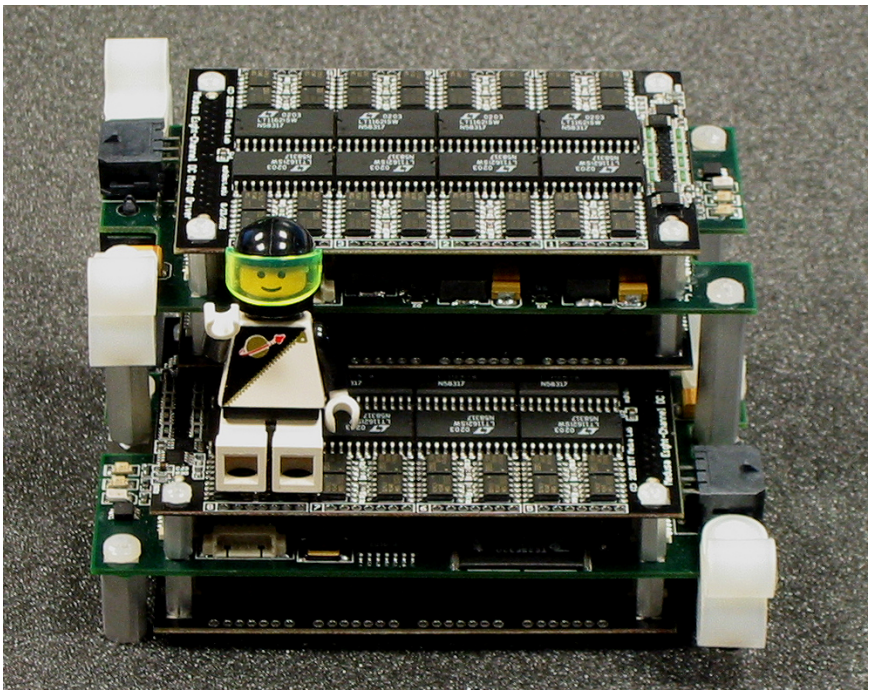


Figure 3.3: Two Medusa FPGA-based dual-port control boards each with two eight-channel motor driver cards, designed for use in *Leonardo*'s head. Lego guy included for scale.

3.1 Design Overview

Though these motor controllers were originally intended for use with the robots described in Chapter 2, it was also hoped that they would be useful for other projects in the future. This prompted a modular design. Each motor control package consists of a control board and one or more motor driver boards. The motor boards contain the power amplifiers, analog sensing circuitry, and other hardware used to interface to the robots themselves. The control board contains the digital electronics which implement control algorithms and communicate with the host computer, and also contains power conversion electronics to generate the various supply voltages that are needed around the system. Ideally, any control board may be configured for use with any motor driver boards to meet the needs of each new project.

The interface between the control and driver boards is very simple, consisting of several power rails and a large number of general-purpose digital I/O lines. This affords the motor driver designer great flexibility. Power and data are passed over two separate 64-pin board-to-board mezzanine connectors (Molex series 71436). The

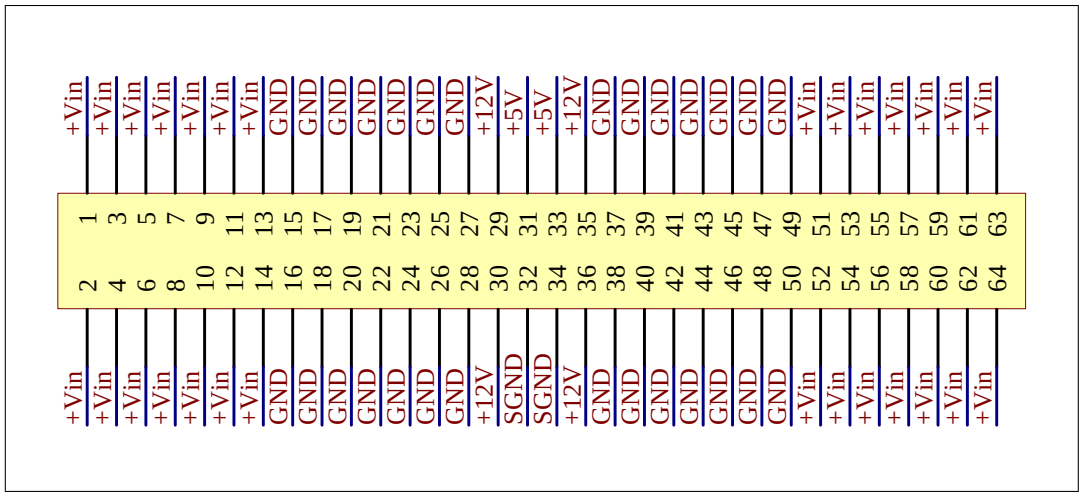


Figure 3.4: Pinout of the Medusa power stacking connector.

64 digital I/O pins are completely unassigned, and the 64 power pins are assigned as shown in Figure 3.4. By ganging together a large number of pins for the motor power supply rails ($^{+}V_{in}$/GND) this design supports up to roughly twenty amps of motor current provided to each driver board. The connector also supports over one amp each for the motor driver supply ($^{+}$12V/GND) and control circuitry supply ($^{+}$5V/SGND). The two grounds, GND and SGND, should be connected together at an appropriate star point on the control board.

Only one type of driver board has been designed for this system at the present time, though it comes in two versions. This board, described in Section 3.2, can control eight motors with moderate-current drivers. However, it would be straightforward to design new driver boards to meet the special needs of future robots. In particular, a driver board with a smaller number of higher-current drivers would be a very useful addition to the Medusa line. One version of this board is intended for general use and the other is a connectorless version intended for use in *Leonardo*'s head.

Two different control boards have been designed so far, one of which is available in two form factors. The first board, designed for general use in small projects, uses a standard PIC microcontroller as its main processor and accepts one driver board. This controller, discussed in Section 3.3, features a standard RS-232 serial port and additional power electronics which make it simple to use. The other control board, discussed in Section 3.4, is designed for more sophisticated expressive robots such as *Leonardo*. It features a more powerful soft-core-based controller architecture and

accepts two driver boards. However this control board is not based on a well-known processor like the PIC, requires dual regulated power supplies, and only supports a total of 10A continuous aggregate motor current. These drawbacks render this board less appropriate for general use.

3.2 Eight-Channel Driver Boards

These eight-channel motor driver boards each provide eight channels of motor interface hardware and can connect to any of the Medusa motor control boards. The motor interface hardware consists of the motor driver itself and associated position-, velocity-, and current-sensing circuitry. There are two versions of this board: the standard model, which includes connectors for each motor, and a smaller model, which uses pigtails to reduce connector volume. The smaller model was designed specifically to address the extremely tight volume constraints faced inside the head of *Leonardo*. The standard model also features one general-purpose user I/O line per motor. Full schematics of both the standard model (part "SWMOT8") and the smaller model (part "SWMOT8s") are provided in Appendix A.

3.2.1 Motor Drivers

The main purpose of this board is of course to drive DC motors, reversibly and with adjustable voltage. This function is performed by a standard FET H-bridge. A simplified schematic of the driver circuit is shown in Figure 3.6. Power MOSFETs Q1–Q4 form a bridge, which can connect the motor to the motor supply voltage $^{+}V_{in}$ with either polarity or force the motor voltage to zero. When transistors Q1 & Q4 are on and transistors Q2 & Q3 are off the motor sees the supply voltage with one polarity; we shall refer to this, and the opposite configuration with Q1 & Q4 off and Q2 & Q3 on, as the "enabled states" of the bridge. The motor voltage is forced to zero when transistors Q2 & Q4 are on while Q1 & Q3 are off; we shall refer to this as the "braking state," since it can be used to resistively brake the motor. By rapidly switching between these configurations the bridge can present any effective average voltage in the range $\pm V_{in}$ to the motor. This process is known as pulse-width modulation, or PWM, and is the cornerstone of most DC motor control.

The four digital control inputs (L/RHIGH & L/RLOW) cannot directly drive the gates of the FETs, and so four gate drivers A1–A4 are needed. These are powered by

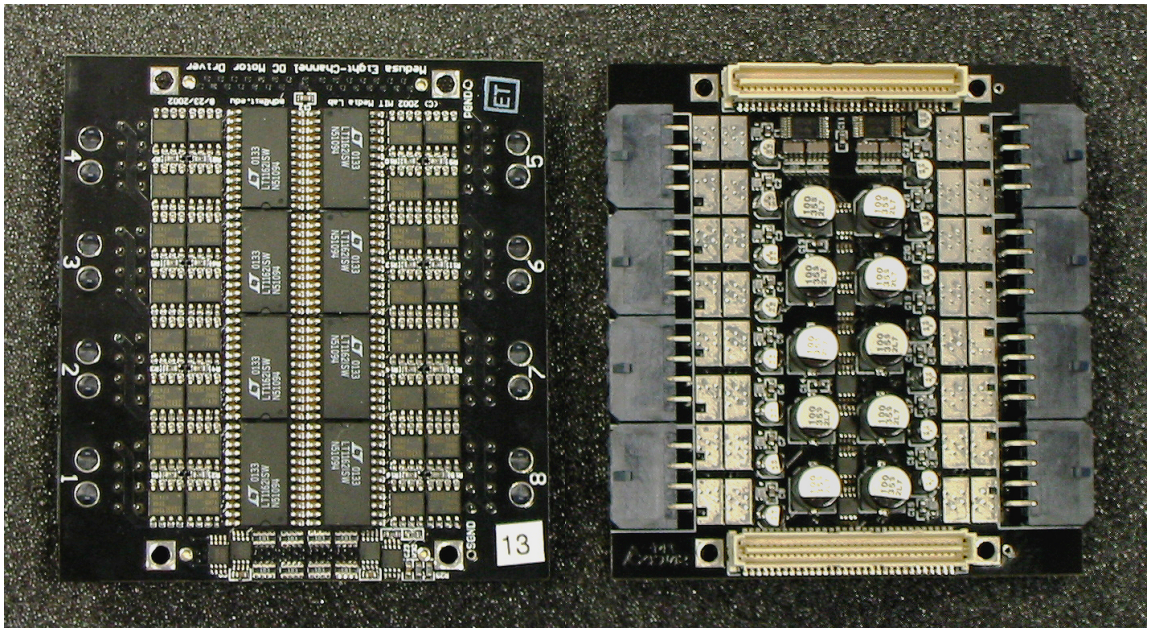


Figure 3.5: Top and bottom views of the Medusa eight-channel motor driver board.

a separate gate voltage supply ${}^{+}V_g$, in this case 12V. Since ${}^{+}V_{in}$ may be higher than ${}^{+}V_g$, additional circuitry is needed to generate an even higher gate voltage with which to turn on the high-side FETs Q1 & Q3. This is handled by bootstrap capacitors C1 & C2 and diodes D1 & D2. Considering the left side of the bridge, this works as follows. When Q1 is off and Q2 is on capacitor C1 is charged to approximately ${}^{+}V_g$ relative to ground through diode D1. Then when Q2 is off and Q1 begins to turn on the capacitor makes sure the positive supply of A1 remains roughly ${}^{+}V_g$ higher than the source of Q1. As long as the value of C1 is sufficiently large the gate-to-source voltage of Q1 can be held at roughly ${}^{+}V_g$ even when the source of Q1 is at ${}^{+}V_{in}$, i.e. when transistor Q1 is fully on.

Resistors R1 & R2 are current sensing resistors. Outputs LSENSE & RSENSE run to a difference amplifier, and when the bridge is in either enabled state the output of this amplifier will be proportional to the motor current. Current sensing will be discussed in more detail in the next section.

On this board, the functions of gate drivers A1–A4 are provided by a single monolithic bridge driver chip, the Linear Tech LT1162. This chip additionally ensures that transistors Q1 & Q2 are never on simultaneously, nor are transistors Q3 & Q4, since those conditions would result in an effective short-circuit between power and ground and would have destructive consequences. Although this chip is very

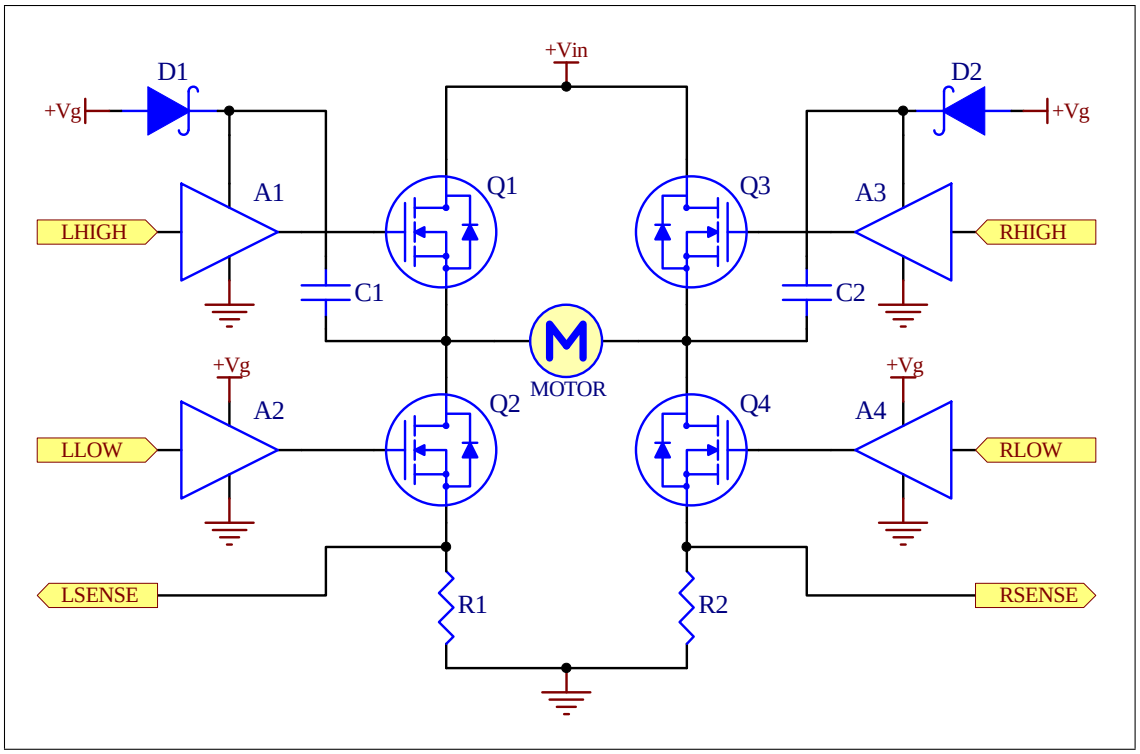


Figure 3.6: Simplified schematic of the H-bridge motor driver topology used in the eight-channel motor driver boards.

convenient and relatively compact, it does have one drawback: it is fabricated on a legacy bipolar process, and so has very high quiescent current consumption. Since the supply voltage requirements are relatively high, this results in a great deal of wasted power, which presents a significant thermal problem for enclosed robots. Future versions of this board should almost certainly replace this chip with a cooler-running alternative, possibly based on a custom design using discrete components if no better monolithic solution can be found.

There are several ways to drive a bridge of this sort. As discussed above, we want to switch rapidly between two states with varying duty cycle. Which two states? Clearly the first state must be the enabled state with the desired output polarity. There are three reasonable possibilities for the other state: the opposite-polarity enabled state, the disabled state (all transistors off), and the braking state. The disabled state is not desirable because it forces the motor current to flow through the FETs' parasitic diodes, which will consume more power than switched-on FETs. Using the opposite enabled state has the advantage that the H-bridge is then always enabled, and therefore the current-sensing differential amplifier is always properly

measuring the motor current. However, switching to this state requires switching all four transistors, whereas switching to the braking state requires switching only two transistors. Since the FET gate capacitances are large and the required gate voltages are high, this represents a significant difference in switching power and a corresponding difference in heat generation. Perhaps more importantly, when switching between the two enable states it is only possible to achieve exactly zero output voltage through a precise balancing of the two states. The braking state is clearly the preferred second state, though it forces a more complicated current measurement, as discussed in the next section.

For these boards, the International Rectifier IRF7470 HEXFETs were chosen as the power transistors because of their low $R_{DS,on}$, their high current ratings, and their small SO8 package. Sense resistors of 0.1Ω were installed; these resistors provide decent current-sensing resolution at low currents, as discussed below. However, at higher currents the voltage drop across these resistors will become undesirably large, and since they are $^1/_{16}$W resistors they will be destroyed if a current of more than an amp is applied for too long. If higher average currents are needed then these resistors can be reduced in value, or they can be replaced with jumpers if current sensing is not required. The next factor limiting the motor current is the motor connector, a Molex MicroFit 3mm connector rated for 5A. Sustained motor currents higher than this are not supported by these boards.

3.2.2 Sensor Electronics

Each eight-channel motor driver board supports sixteen channels of analog-to-digital conversion. One channel is wired to each of the eight motor connectors, and is intended to be used for position feedback from a potentiometer, though these channels could be used for other purposes if potentiometer feedback is not used in a particular application. The other eight channels are connected to the current-sensing circuitry of the eight motor drivers.

This current-sensing circuitry is diagrammed in Figure 3.7. The sense inputs RSENSE & LSENSE come from the H-bridge, as shown in Figure 3.6. These inputs are sent to a differential amplifier which generates a 2.5V-centered single-ended current signal. This signal is then filtered by a simple RC low-pass filter (R1 & C1) to isolate the remaining sensing circuitry from switching fluctuations and transients and is then routed to an eight-channel multiplexer (A2). At sampling time, the filtered

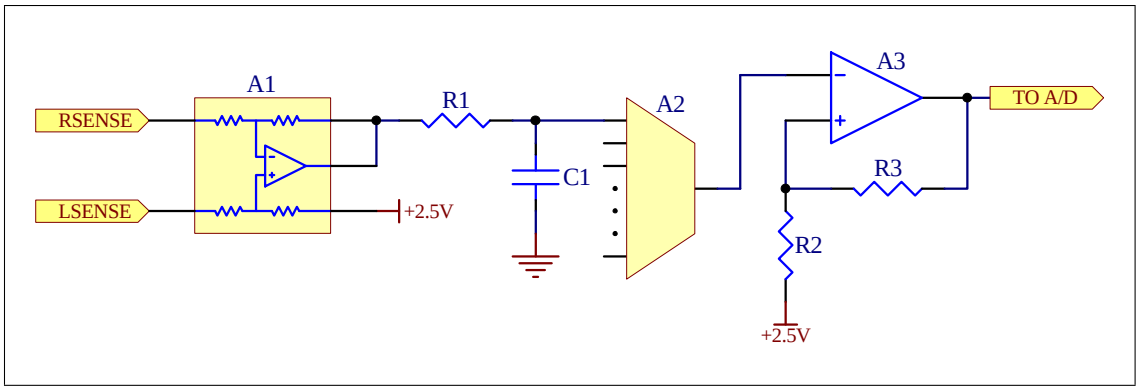


Figure 3.7: Simplified schematic of the current-sensing circuitry used in the eight-channel motor driver boards.

signal from the selected channel is amplified by an inverting amplifier (A3, with a gain of $\frac{R3}{R2}$) and sent to the A/D for conversion.

As mentioned above, since the full motor current passes through the current sense resistors (R1 & R2 in Figure 3.6) these resistors must be kept small or they will introduce significant losses. However, they must not be made *too* small or the signal will be swamped by noise and bias currents in the sensing circuitry. For the present robots (*Public Anemone* and *Leonardo*) no individual motor is expected to consume a current of more than about an amp, and even currents of that size are not expected to last for very long. The sense resistor value of 0.1Ω is therefore acceptable; these resistors produce a voltage of 0.1V at 1A, which is just barely negligible given the 12–24V intended operating voltage range. However, as discussed above, these $^1/_{16}$W resistors limit the allowed sustained current provided to each motor, and must be replaced if higher continuous currents are expected.

In this design, the differential amplifier (A1) is a Texas Instruments INA152, which has unity gain in the configuration shown. The signal that is sent to the low-pass filter therefore has a range of approximately ±0.1V. Capacitor C1, which is actually one element of a high-density capacitor array, was chosen to have value 0.1μF, the largest value available in that package. The time constant of the filter is then set by the value of R1 (which is also an element of a resistor array). In the original design it was expected that the current sense signal would be relatively constant throughout the PWM cycle, except at switching transitions, and so a value of 1KΩ was used, for a filter cutoff frequency of 10KHz. As discussed below, this assumption is not correct in all cases, and these resistors have been increased to 10KΩ (for a cutoff frequency of

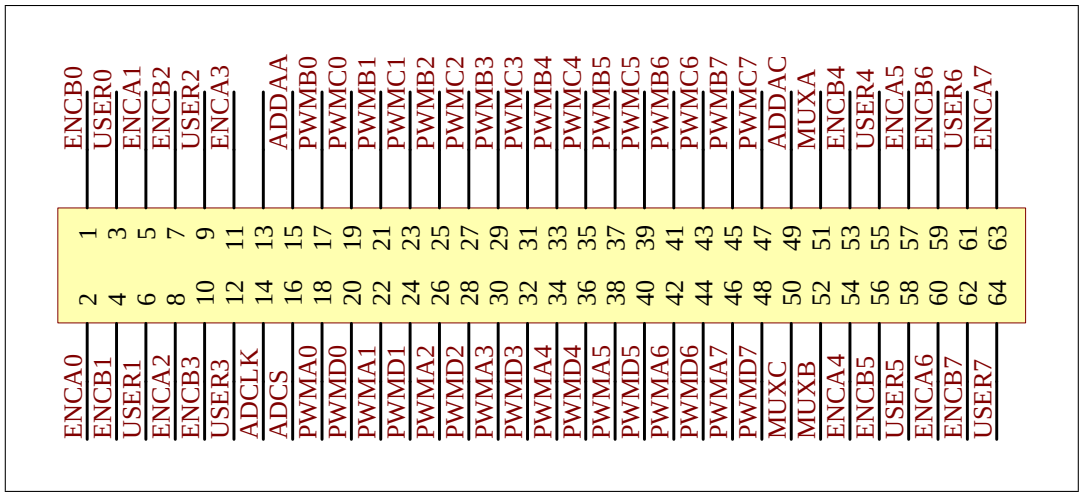


Figure 3.8: Pinout of the SWMOT8/SWMOT8s digital stacking connector.

1KHz) in recent revisions. This provides a relatively clean signal for PWM frequencies larger than 10KHz. The gain of the amplifier A3 can then be adjusted to take full advantage of the 0–5V input range of the A/D converter. For *Leonardo* and *Public Anemone* this gain is set to 20, mapping a current range of $\pm$1A onto an input voltage range of $2.5 \pm 2V$ at the converter.

Depending on the H-bridge control method used by the control board, the converted value may need to be further adjusted in firmware or software before it correctly represents the motor current. If the second PWM state is either the reverse enabled state or the floating state then no further adjustment is needed: one sense resistor always sees the full motor current while the other sees no current, and so the differential amplifier always outputs the correct value. However if the second PWM state is the braking state, as is often desirable, then both sense resistors see the motor current during the second phase. This increases the effective sensed current. The sensed current I_s and actual current I_a are related at a given PWM duty cycle d by

$$I_s = dI_a + (1-d)(2I_a) \quad \Longrightarrow \quad I_a = \frac{1}{2-d} I_s \ .$$

This correction factor can be applied directly in firmware by any controller capable of performing division, or it can be applied in software if the duty cycle being applied at the time is also known.

Much like the eight current sensor signals, the eight potentiometer inputs are low-pass filtered, multiplexed, buffered, and sent to an A/D converter. The only

difference is that the buffer has unity gain instead of large negative gain. A separate A/D converter is used for the potentiometer inputs. The multiplexers are Texas Instruments SN74LV4051, and the A/D converters are Texas Instruments ADS8320. The ADS8320 is a 16-bit serial A/D converter capable of a 100kHz sampling rate. Since each converter is responsible for converting eight channels, this system can achieve a roughly 12kHz update rate per channel. Most controllers will operate at a considerably lower control frequency, and so oversampling can be used to improve the signal-to-noise ratio.

In addition to these analog feedback channels, this board also routes three digital I/O lines from the control board to each of the motor connectors. Two of these are intended to be connected to a quadrature encoder attached to the motor, though they may be reclaimed for other uses if encoders are not required in a particular application. The actual processing of the quadrature encoder signals is left to the digital control board. The third line could be connected to an index encoder or used for any other purpose. This general-purpose line is omitted from the smaller S-model boards for space reasons.

The control signals for all the circuits described above are routed to the digital stacking connector, with the pinout shown in Figure 3.8. Pins PWMxx are the H-bridge control lines, pins MUXx are the analog multiplexer channel-select lines, pins ADCLK and ADCS control the A/D converters, pins ADDAA and ADDAC are the serial outputs from the pot and current A/D converters, and pins ENCAx/ENCBx/USERx are the digital I/O lines. The pinouts of the motor connectors are shown in Figure 3.9. Pins MOTA and MOTB are the motor outputs, the POT pin is for the potentiometer input, pins ENCA and ENCB the digital inputs intended for quadrature encoders, and pin USER (present only on the SWMOT8) is the general-purpose digital I/O pin. The +5V and GND pins provide access to a five-volt power supply with which to power the potentiometer, encoders, or other low-power electronics as needed.

3.3 PIC-Based Single-Port Controller

The PIC-based single-port control board was designed to be easy to use for a wide range of small robotics applications. It features a standard RS-232 serial port with a female DB9 connector for easy connection to a host PC. It also features a a single high-

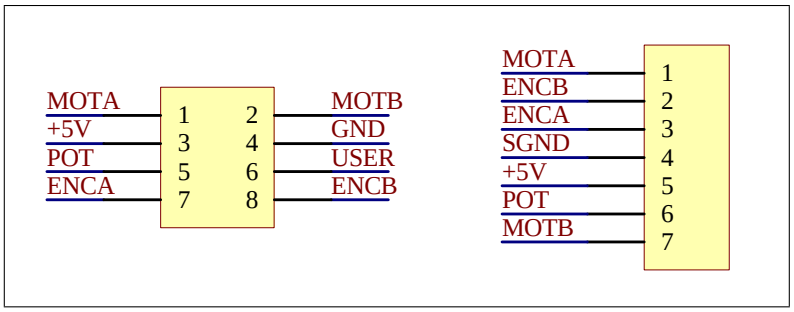


Figure 3.9: Pinout of the SWMOT8 (left) and SWMOT8s (right) motor connectors.

current power connector which accepts any motor voltage in a wide range (roughly 5V to 30V, depending on the specific driver hardware chosen) and generates all internal supply voltages from this one input. At the core of the board is a Microchip PIC16F877, a microcontroller which was selected because it is used frequently around the MIT Media Lab. An FPGA connects the microcontroller to the digital stacking connector, isolating it from the details of managing the hardware on the driver board. This FPGA can also act as a math coprocessor, allowing the use of more sophisticated control algorithms than could normally be implemented in this processor. A full schematic of this board can be found in Appendix A.

3.3.1 Power Conversion

One of the primary design goals for this board was that it support a wide range of motor supply voltages without requiring a separate power supply for the control circuitry. Neither 5V nor 24V DC motors are uncommon, and so this voltage range (with a little headroom on the high end) was chosen as the allowable input range. Any Medusa control board is required to generate a 12V supply rail for the power stacking connector, and so a power converter was needed that could step the motor voltage either up or down. A single-ended primary-inductance converter (SEPIC) is capable of performing exactly this function.

A simplified schematic of the SEPIC converter used in this board is shown in Figure 3.11. When transistor Q1 is on it charges up inductor L1, and when it is switched off that energy is dumped through capacitor C1 and diode D1 onto the output capacitor C2. This is much like the operation of a standard boost converter; however, capacitor C1 breaks the direct current path from input to output and permits

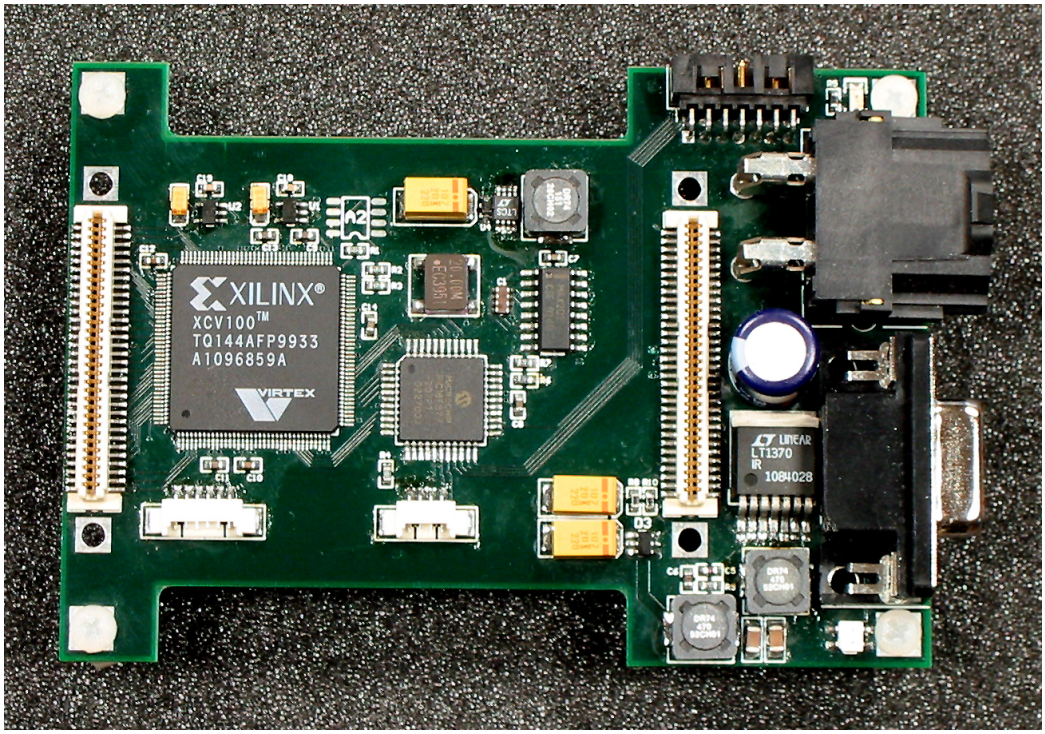


Figure 3.10: The Medusa PIC-based single-port control board.

the output voltage to fall below the input voltage. Inductor L2 is then needed to provide a DC current path to the output. By varying the duty cycle of the SWITCH input, the output voltage can be varied over a large range from well below to well above the input voltage. In particular, it is easy to show that with no load the output voltage is given by

$$\mathrm{V_{out}} = \frac{d}{1-d}\mathrm{V_{in}} \ ,$$

where d is the duty cycle of transistor Q1. The output voltage is measured and the duty cycle is then adjusted to compensate for variations in the load.

In the actual design the switching transistor Q1 is internal to a monolithic SEPIC controller chip, the Linear Technologies LT1370. This part provides all the feedback control needed to generate a steady output voltage using this topology. The inductors are Cooper DR74-470 47μH inductors with a maximum current of 1.1A. This is the limiting factor in determining how much current may be drawn from the 12V supply generated by this converter. If the input voltage is above 12V then the output current may be as large as an amp, but this current is derated for proportionally for lower input voltages.

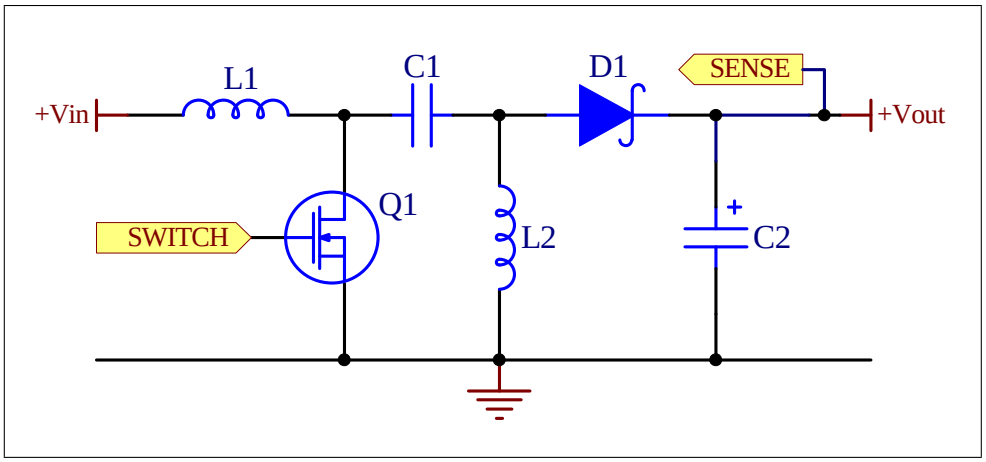


Figure 3.11: Simplified schematic of the SEPIC power converter used in the PIC-based single-port controller.

Once a stable 12V rail has been generated it is then used to generate the other required voltages. First a simple buck converter is used to efficiently generate a 5V rail. A simplified schematic of this circuit is shown in Figure 3.12. As transistor Q1 is switched on and off, diode D1 correspondingly switches off and on. This results in a reduced average voltage which is losslessly filtered by inductor L1 and output capacitor C1. In this circuit, as in the SEPIC converter, the switching transistor is included as part of a monolithic converter package, here the Linear Tech LT1474. The inductor is a Cooper DR74-151 150μH inductor, which sets the maximum allowable output current to about half an amp. However, if this much current were drawn from this converter it would require a current draw of about 1.2A from the SEPIC converter, which is not allowed. Therefore the load on the 5V converter must be kept rather lower than .5A. In practice it will usually be on the order of 100mA, roughly the load presented by the control board's own electronics, since the motor driver board is unlikely to require much current from this supply.

Two linear regulators generate 2.5V and 3.3V rails needed by the FPGA from this 5V rail. The current demands at those voltages are quite low, and so additional DC/DC converters were not required.

3.3.2 Controller Architecture

As described above, the basic controller architecture in this board consists of a PIC microcontroller accompanied by an FPGA. The FPGA configuration data is stored

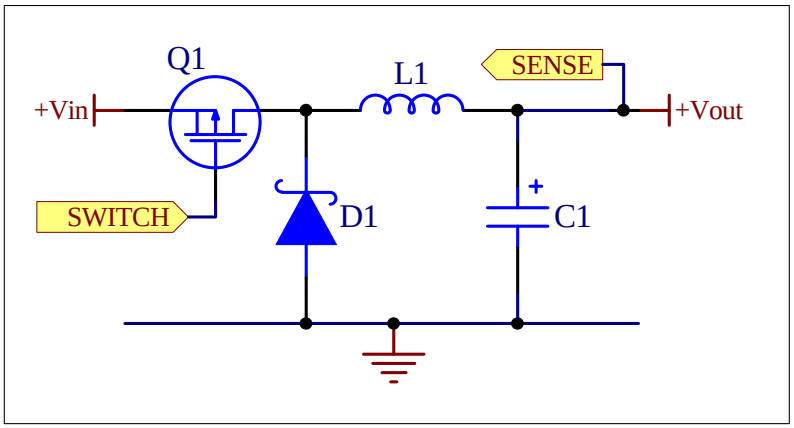


Figure 3.12: Simplified schematic of the buck power converters used in the PIC-based single-port controller and the FPGA-based dual-port controller.

in a one-time-programmable PROM; the intention is for each board to be configured once for use with a particular type of motor driver module. The two boards could then be given to the end user, who need only worry about programming the PIC for the final application. A standard set of PIC libraries could be developed along with each driver board which would allow even a novice PIC programmer to control the FPGA and motor hardware without trouble.

The core microcontroller, a PIC16F877, has 31 user I/O pins. Of these, 18 are connected directly to the FPGA, intended for use as an eight-bit data bus with six address lines and four control lines. If the four control lines are used as Read, Write, Reset, and High/Low signals then this bus has an address space of 64 16-bit words. Two PIC pins are used by the PIC's hardware UART to communicate over the serial port to the computer, two control two status LEDs, and eight of the remaining nine are routed to an external connector for use as general-purpose digital I/O lines and up to five low-resolution analog inputs.

The FPGA is a Xilinx XCV100 Virtex, chosen largely because it was already in stock as a prototyping tool for the FPGA-based control board described in the next section. This FPGA has a low-power 2.5V core, but is configured in this board to use 5V-compatible 3.3V inputs and outputs. This makes it possible for the FPGA to communicate directly with standard 5V parts on the motor driver board as well as with the PIC without additional interface hardware. The 31 control lines from the PIC and the 64 control lines to the motor driver board consume virtually all of this part's I/O pins.

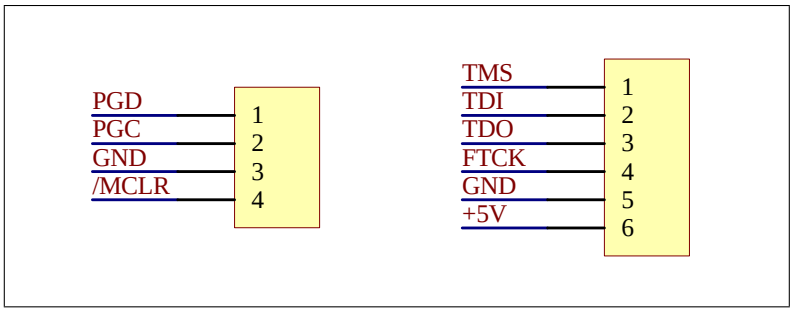


Figure 3.13: Standardized programming connectors for PICs (left) and FPGAs/CPLDs (right).

Both the PIC and the FPGA receive a 20MHz clock from a free-running oscillator. With this clock the PIC operates at an instruction rate of 5MHz, at which speed it takes approximately 1.6μs to execute a 16-bit bus read and 3.4μs to execute a bus write. With the FPGA acting as a math coprocessor, a 16-bit binary operation then takes about 6.6μs. A 16-bit software multiply operation would take roughly ten times as long: using the FPGA as a coprocessor makes it possible to use a single PIC to implement standard control algorithms like PID on multiple channels simultaneously, a feat which would otherwise be possible only at very low control frequencies.

All other details of how to use this board are left up to the firmware designer. An example combination of FPGA and PIC firmware is described in the next chapter. However, this hardware is extremely flexible and could be used in a variety of other ways. In particular it could be operated fully autonomously, with no external computer and with a small amount of sensor data being supplied over the general-purpose I/O connector. The PIC's hardware SPI port is accessible via that connector, and so a relatively sophisticated sensor suite could be connected directly as well. The user would only be required to modify the PIC code in such a design; the FPGA code could be used as-is to manage the motor driver hardware.

Both the PIC and FPGA have in-circuit programming headers; these headers are standard across the Medusa line, and the pinouts are given in Figure 3.13. Both are Molex Micro-Minniature 1.25mm wire-to-board connectors. The FPGA programming header is only to be used for code development; configuration data provided via this header is lost when power to the FPGA is turned off. Once the final FPGA code is complete, it should be burned into a configuration PROM (Xilinx XC17V01) and installed on the board.

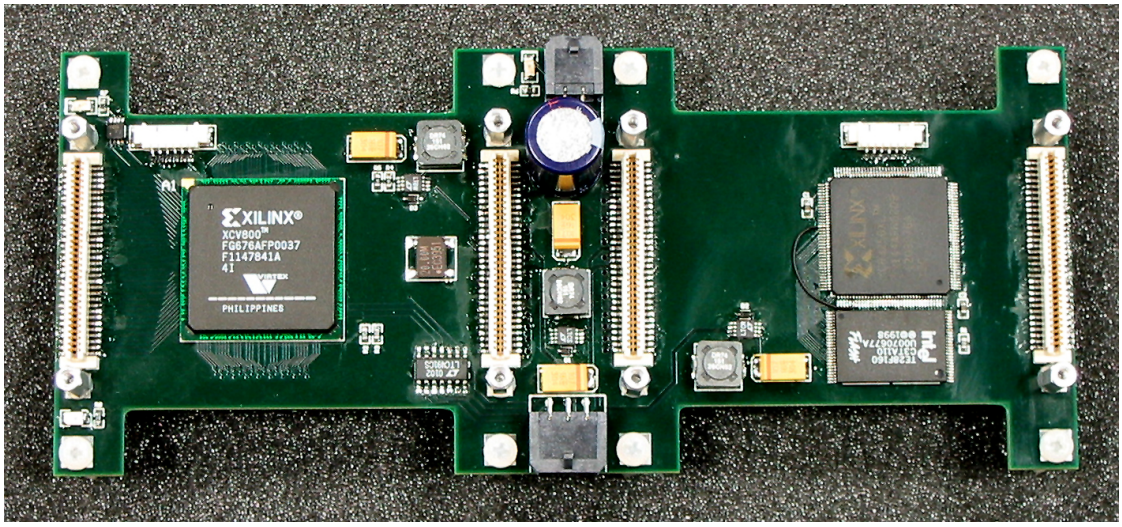


Figure 3.14: The Medusa FPGA-based dual-port control board (standard model).

3.4 FPGA-Based Dual-Port Controller

The other control board which has been developed so far was designed specifically to meet the unusual needs of *Leonardo*. It can control a large number of motors in a small volume with flexible control logic, but it does not support high aggregate currents. The design is similar in many respects to the design of the PIC-based board discussed in the previous section. A Xilinx Virtex FPGA (here an XCV800) connects to the digital stacking connectors, and the FPGA again has a 2.5V core with 5V-compatible 3.3V I/Os. The 5V, 3.3V, and 2.5V supply rails are all generated from the 12V rail using buck converters just like the one described in the previous section.

There are however several important differences between the two control boards. This board can accept two motor driver modules, allowing it to control twice as many motors as the previous controller. Further, this board uses RS-485 instead of RS-232, allowing for reliable higher-speed communication with the host PC or intermediate hardware. The board's power and communications connectors were chosen to be small enough to pass through the tight spaces in *Leonardo*'s neck; as a result, the total continuous motor current supported by this control board is limited to only 10A. (This is not a problem for *Leonardo*, which uses four of these controllers and draws less that ten amps in its entirety.) Since the volume and complexity of a SEPIC converter was undesirable in *Leo*'s head, this board does not have one; instead, it must be provided with a regulated 12V power supply. The pinouts of the two connectors (both Molex MicroFit 3mm connectors) are shown in Figure 3.15.

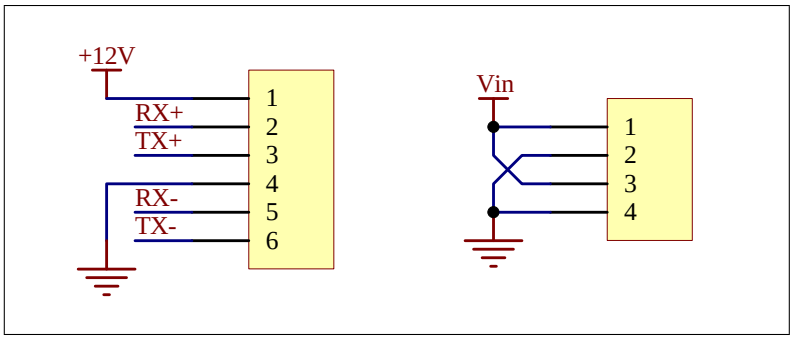


Figure 3.15: Connector pinouts for digital power and RS-485 (left) and motor power (right) for the FPGA-based dual-port controllers.

By far the most important difference between the two boards is that there is no traditional microcontroller present in this design. Instead the FPGA is quite large, allowing the firmware designer to use a soft-core processor appropriate to the application. Though a high-performance commercial processor could certainly have been used instead, the board needed a large FPGA to manage the large number of digital I/O pins on the two stacking connectors anyway, and the benefits of added flexibility were deemed greater than the drawbacks associated with the increased FPGA design complexity. One example controller , based on a simple custom soft-core processor, is described in the next chapter; this controller firmware is presently being used to with *Leonardo*. As open-source soft cores improve in reliability, performance, and usability, these could be introduced and modified to suit the needs of motor control.

The configuration data for this large FPGA consumes over four megabits, and so it will not fit in a standard serial EEPROM. Instead, this board uses flash memory (Intel 28F160C3) to store the configuration data. This memory has a 16-bit parallel data bus, and so some glue electronics is needed to configure the FPGA with it at power-up. An auxiliary CPLD, which is in many ways similar to a small FPGA but which retains its configuration permanently after programming, performs this function. The CPLD can clock the configuration data out of the flash and present it to the FPGA via the FPGA's 8-bit SelectMAP programming interface. Both the FPGA and the CPLD also have JTAG programming ports, with the standard pinout shown in Figure 3.13. When the board is first configured, both the FPGA and CPLD must be programmed via these connectors. The user may then upload the desired

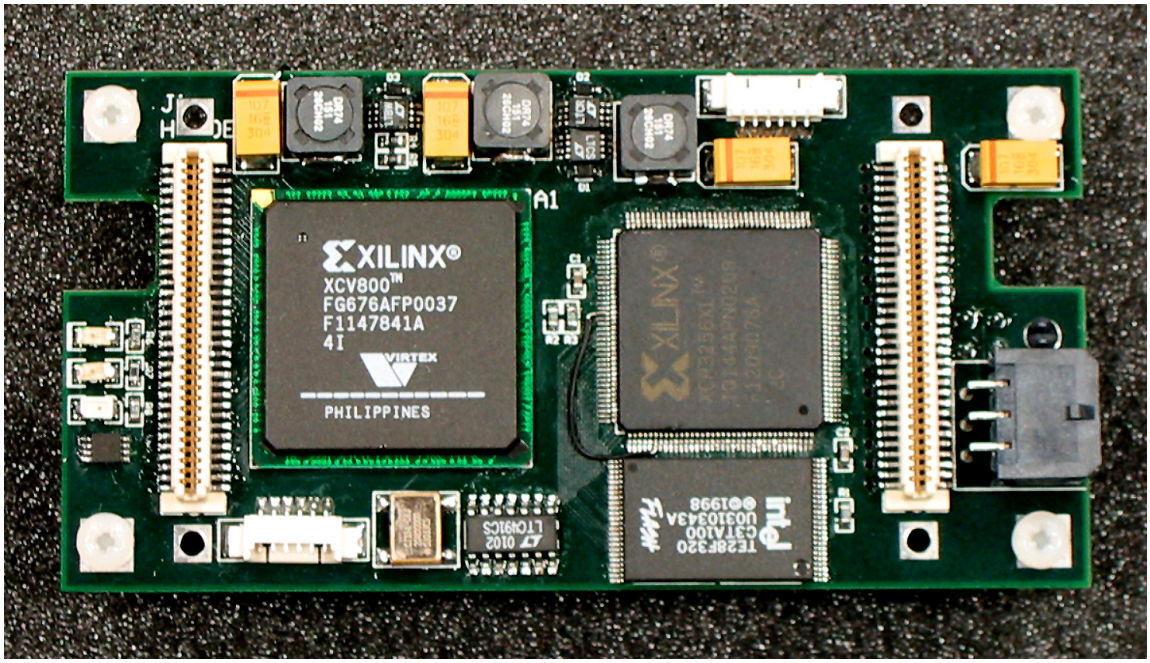


Figure 3.16: The Medusa FPGA-based dual-port control board (smaller S-model).

FPGA configuration via the serial port; the FPGA receives this data and instructs the CPLD to load it into flash. This port also offers the FPGA developer a much faster method of testing FPGA code than uploading it to the flash each time.

This board is available in two form factors. The first, shown in Figure 3.14, is a larger design which was developed originally for testing and which is easier to work with in situations where volume is not an issue. It accepts both motor driver modules side-by-side on the top of the board. This larger version has been used for testing to control all of *Leonardo*, and will continue to be used to control *Leo*'s body and arms. The second version, shown in Figure 3.16, is much smaller. It is accepts one motor driver module on each side in a sandwich configuration. This board can be used in conjunction with the smaller S-model eight-channel motor drivers to achieve the extremely high controller and driver densities needed inside *Leo*'s head.

Chapter 4

Medusa Firmware

This chapter describes the firmware which has been written to date to control the hardware described in Chapter 3. This code was developed for three reasons: to test the hardware, to make it possible to put the hardware to immediate use, and to provide examples for future firmware authors. Early versions of the firmware described here were used to control the *Public Anemone*, and the most recent version is currently being used inside *Leonardo*.

This chapter describes a large amount of code written for five devices. There is far too much code to describe it in full detail here; this chapter is intended only as an introduction. For more complete documentation of this code and the complete source of the most recent version see the project website, which can be found at `http://robotic.media.mit.edu/motor/`. More importantly, this code is continuously being improved; the website will also document the most recent versions.

4.1 PIC-Based Controller

The PIC-based controller described in Section 3.3 has two devices that must be programmed for it to operate, the PIC microcontroller itself and the support FPGA. The FPGA must be programmed to interface to whatever motor driver hardware is going to be attached, and it may also be programmed to provide additional math co-processing features. The PIC must then be programmed according to the needs of the particular application.

In this section we will present example firmware for both devices. The FPGA firmware discussed here is designed for use with the eight-channel motor driver board

- `main` — Peripheral bus controller.
  - `analog` — Controls SN74LV4051 muxes and ADS8320 A/D converters.
  - `decoders` — Contains eight quadrature decoders and bus logic.
    - `simpledec` — Quadrature velocity decoder module.
  - `pwmmod` — Contains eight PWM generators and bus logic.
    - `pwmgen` — PWM generator module for H-bridge control.
  - `mult16` — A 16-bit hardware multiplier core.

Table 4.1: Structure of the example Verilog code for the support FPGA on the PIC-based single-port motor control board.

described in Section 3.2. It also provides a hardware multiplier for use by the PIC. The PIC firmware implements eight general-purpose PD-controllers under the control of a host computer via the serial port.

4.1.1 Support FPGA

The support FPGA must perform a variety of tasks to manage the attached eight-channel motor driver board. Most importantly it must generate the control signals for the H-bridge driver chips, operate the analog multiplexers and A/D converters, and interpret the signals coming from the motor encoders. In order to program an FPGA to perform these sorts of logic tasks, one must first describe the tasks using a Hardware Description Language (HDL). This example code was developed in Verilog HDL (not to be confused with the Ada-like alternative language VHDL). Like any HDL, Verilog allows the programmer to define a logic module and then to instantiate that module any number of times in the definition of other modules. Each module can then be kept relatively simple and easy to debug. A top-level module, instantiated once, connects the other modules to each other and to the outside world. The module hierarchy for the support FPGA code described here is shown in Figure 4.1.

The `analog`, `simpledec`, and `pwmgen` modules each provide an interface to some of the control hardware on the motor driver board. For example the simplest module is the `pwmgen` module which generates the PWM control signals used to control an H-bridge as described in Section 3.2.1. We shall discuss this module in detail to provide an example of how such a module may be constructed. The code for the module is shown in Figure 4.1. The main inputs are a clock, a 10-bit counter rolling over at the

```
module pwmgen(CLOCK,RESET,COUNTER,PWMVAL,DIR,TPWMA,BPWMA,TPWMB,BPWMB);

    input CLOCK;              // Clock input
    input RESET;              // Active-high reset line
    input [9:0] COUNTER;      // 10-bit counter rolling over at PWM freq.
    input [9:0] PWMVAL;       // 10-bit desired PWM value
    input DIR;                // Desired PWM direction
    output TPWMA;             // Controls left high-side FET
    output BPWMA;             // Controls left low-side FET
    output TPWMB;             // Controls right high-side FET
    output BPWMB;             // Controls right low-side FET

    reg [9:0] pwmreg;         // 10-bit PWM value being used this cycle
    reg dirreg;               // PWM direction being used this cycle
    reg negpwm, pospwm;       //
    wire pwm;

    /* Latch PWM value during the dead time */
    always @(posedge CLOCK or posedge RESET) begin
        if(RESET) begin
            pwmreg <= 0;
            dirreg <= 0;
        end
        else if(COUNTER == 10'h3F0) begin
            pwmreg <= PWMVAL;
            dirreg <= DIR;
        end
    end

    /* Compare on both clock edges to increase resolution */
    always @(negedge CLOCK) negpwm <= (COUNTER<pwmreg && !&COUNTER[9:5]);
    always @(posedge CLOCK) pospwm <= (COUNTER<pwmreg && !&COUNTER[9:5]);
    assign pwm = negpwm & pospwm;

    /* Set the four H-bridge control lines accordingly */
    assign BPWMA = dirreg ? 1 : ~pwm;
    assign TPWMA = dirreg ? 0 : pwm;
    assign BPWMB = dirreg ? ~pwm : 1;
    assign TPWMB = dirreg ? pwm : 0;

endmodule
```

Figure 4.1: Verilog PWM generator for the FPGA `pwmgen` module.

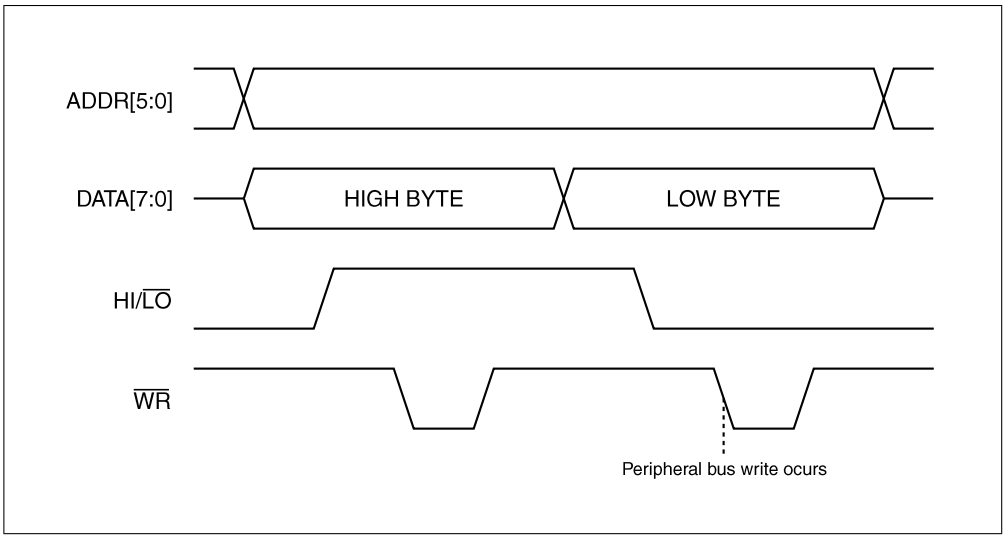


Figure 4.2: PIC-FPGA host bust 16-bit write cycle.

desired PWM frequency, a 10-bit value indicating the desired PWM duty cycle, and a bit indicating the direction of output. The module then generates the a PWM signal by constantly comparing the counter to the requested duty cycle and uses the result of this comparison, along with the direction indicator, to generate the four control signals needed by the bridge driver.

The H-bridge driver design used by the driver boards requires that the bridge be turned off for at least a brief interval once per cycle; if it is held on for too long the bootstrap capacitor will discharge and the bridge will fail to operate properly. The maximum duty cycle that this module will generate is 0x3E0/0x400≈97%. During the dead time the module also latches the values of duty cycle and direction that it will use for the next cycle. This prevents output glitches even when the inputs are changed mid-cycle.

Another potential source of output glitches is the comparator itself. There may be glitches on the comparator output each time the counter value increases, because of differences in combinatorial path delays. Therefore the output cannot be used directly as a PWM signal, but must be synchronously sampled instead. This sampling is done on both rising and falling clock edges, allowing the counter to increment as fast as twice the clock frequency and supporting correspondingly high PWM frequencies. The counter must be provided as an input instead of simply being generated in the module so that multiple instances of the `pwmgen` module may be used without counter duplication. While it is likely that the synthesis or implementation tools

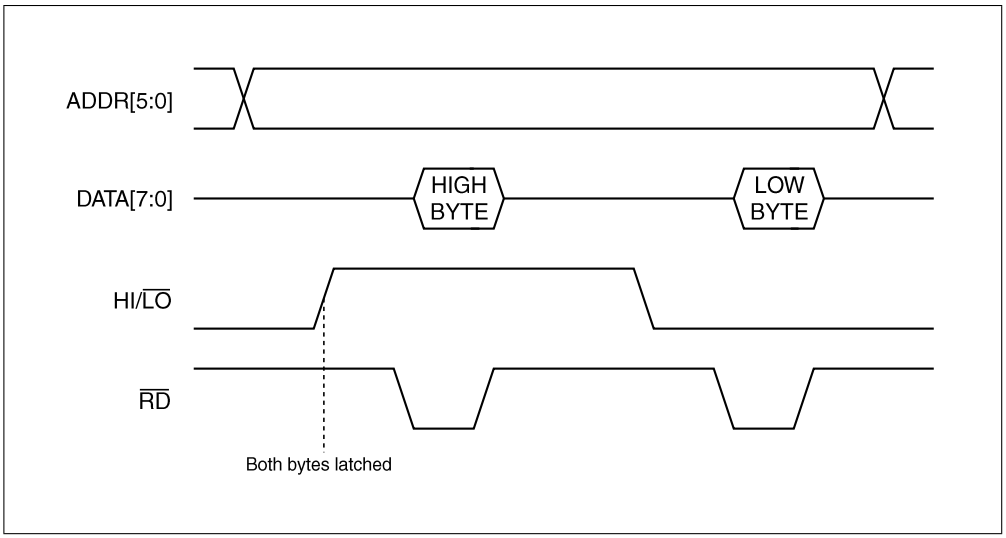


Figure 4.3: PIC-FPGA host bust 16-bit read cycle.

would optimize out such duplicate counters, optimizations of this sort are easy to include directly in the design.

The `simpledec` module is a similarly-simple module which measures a motor's velocity using the signals from a quadrature encoder. The `decoders` and `pwmgen` modules provide bus-style access to eight `simpledec` modules and eight `pwm` modules respectively, one of each for each motor channel. The `analog` module controls the analog multiplexers and A/D converters, sampling the motor positions and currents using 8× oversampling. The `mult16` module describes a multiplier taking two 16-bit multiplicands and generating the 32-bit product. This multiplier is intended to be used by the PIC to improve its computational performance. Modules of this sort are easy to generate using the Xilinx CORE Generator software tool.

Each of the internal modules was designed with a 16-bit peripheral bus interface, while the PIC uses an 8-bit host data bus. Therefore the top-level module (`main`) implements an 8-to-16-bit bus converter. The host uses control pin HI/$\overline{\text{LO}}$ to select whether it is accessing the high or low byte. For read operations the converter latches the data to be read on the rising edge of HI/$\overline{\text{LO}}$, while for write operations the high byte is buffered in the converter and a peripheral bus write can be initiated only the when HI/$\overline{\text{LO}}$ is low. Thus all host bus operations must proceed high-byte-first. The timing diagram for a host bus write is shown in Figure 4.2, while the timing diagram for a bus read is shown in Figure 4.3.

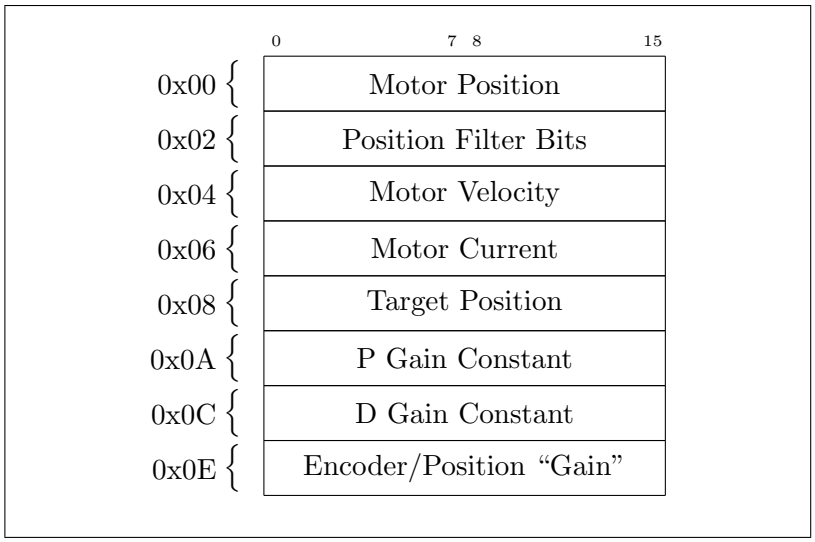


Figure 4.4: PIC communications memory layout.

4.1.2 PIC Firmware

The example PIC firmware implements a PD controller for each motor as well as position estimation filters and code to communicate with the host PC via the serial port. The code can be loosely divided into three sections: FPGA interface code, serial communications code, and the main filter and control loop. At present all three pieces have been written directly in PIC assembly language. However it should be possible to incorporate the first two into an assembly library for use with a PIC C compiler so that the main control code can be rewritten (and later modified by end users) in C.

The FPGA interface code implements the bus protocol shown in Figures 4.2 and 4.3. It provides a number of macros to simplify bus access using both direct and indirect PIC register addressing. Each 16-bit word must be stored across two PIC registers since the PIC is an 8-bit processor; these libraries adopt a big-endian storage convention. The library also includes a number of other basic routines for manipulating 16-bit values within the PIC and performing some simple 16-bit mathematical operations. See the library sources for documentation of these utility macros.

The serial communications code is responsible for communicating with the host computer via the asynchronous serial port. It uses the PIC's hardware UART to relieve the processor of the burden of handling each individual bit. This code resides

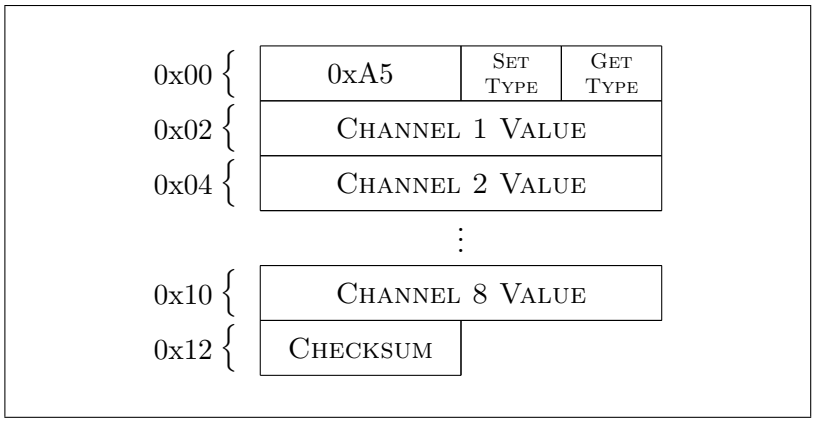


Figure 4.5: PIC communications packet format.

entirely within the PIC's interrupt service routine, providing a clean separation between the serial code and the main control code. Communications between the two modules is handled through a block of shared memory. When the control code wishes to perform a non-atomic access to this memory block, such as any 16-bit access, it must disable interrupts for the duration of the operation. The control code must be designed with the assumption that the contents of the shared memory block may change at any time.

The shared memory consists of eight eight-word slices, one for each motor channel. Communication with the host PC takes the form of read and write operations on a all 16-bit words at a particular offset into each slice. For instance, the host PC may send the controller a command simultaneously updating the first two bytes of each shared memory slice and requesting that the contents of the third and fourth bytes of each slice be sent back in response. The code assigns a particular meaning to each word position within a slice; these assignments are shown in Figure 4.4. As the main control code measures and calculates the motor positions, velocities, and currents it stores these in the shared memory in the given locations. Likewise at each tick the code uses the given values for the target positions and controller gains.

The commands and responses are sent over the serial port using the simple 19-byte packet format shown in Figure 4.5. A header byte (with fixed value 0xA5) and a checksum allow each receiver to reliably separate the received byte stream into packets. A type byte contains both the write offset Set Type and the read offset Get Type. These specify at what offset into each slice the given values are to be written and from what offset the response values are to be read. The PIC echoes the

type field in the response packet. The 8-bit CHECKSUM is a simple twos-complement sum of the remaining bytes of the packet, excluding the fixed header byte. After the host PC has configured the various control gains, normal communications will likely consist primarily of packets of type 0x08, updating the controllers' target positions and requesting measured motor position values in return.

The PIC's serial UART is configured for operation at 56 kbaud with eight data bits, one start bit, one stop bit, and no parity bit (57600:8N1). At full speed this configuration supports roughly 300 serial packets per second per direction. However, if the host PC were to attempt to communicate at the full rate it is likely that received packets would be periodically corrupted due to clock drift between the PC and the PIC. To prevent this, a host PC wishing to send back-to-back packets should insert one padding character with any value other than 0xA5 after each packet. This increases the effective transmit packet size to 20 bytes, and so the maximum reliable communications rate is reduced to roughly 288 packets/sec. Higher baud rates are in general not reliably supported with RS-232.

Moving now to the main filter and control code, we will first discuss the position estimation filters. These are relatively simple filters which estimate the motor position using the redundant data provided by the potentiometers and encoders. Potentiometers are notoriously noisy when in motion. They also have an inherently large output resistance, and so the switching noise associated with both the motor PWM and the quadrature encoders couples efficiently onto the pot sensor lines unless they are *very* carefully shielded. Encoders, on the other hand, provide excellent velocity measurement but cannot measure absolute position at all and are susceptible to drift when used as relative position sensors.

The filter works in the following way. The potentiometers yield a measured position trajectory $q_m[t]$ which is noisy. The encoders yield a measured velocity trajectory $\dot{q}_m[t]$ which is quite good. The estimated position trajectory $\overline{q}[t]$ is computed according to

$$\overline{q}[t+1] = (1-\alpha)(\overline{q}[t] + \beta\dot{q}_m[t+1]) + \alpha q_m[t+1] \; . \tag{4.1}$$

This filter has two parameters, α and β. If we set the measured velocity to zero then this filter reduces to a simple first-order IIR low-pass filter on the measured position data and the parameter α adjusts the cut-off frequency. However, in order to greatly attenuate noise the cut-off frequency must be placed well into the frequency range of interest, around 1-10Hz, which would render virtually any simple controller unstable.

This situation is easy to remedy: if we assume we knew the position at the previous time step, and if we assume that our velocity measurement is accurate, then we can accurately predict the position at the present time step through the introduction of the $\beta\dot{q}_m$ term. The parameter β is of course set equal to the sampling time T. We now have a predictive filter, and the parameter α adjusts our relative faith in the prediction (derived from the encoders) and the direct measurement (derived from the potentiometer).

The position and velocity are measured in incompatible units derived from the properties of the potentiometer and the encoder. The parameter β must therefore also include a conversion factor between the two unit systems. This value must be configured by the user much like the controller gains, and so it is referred to in the code as the "Encoder/Position Gain" or "E Gain" even though it is not strictly a feedback gain. It is straightforward to determine the correct value empirically: if the value is too high then the controller will undershoot at first, and if it is too high the controller will overshoot. The parameter α has hard-coded value of 2^{-8} in this design, giving the filter a cut-off frequency around 4Hz. With α fixed at a power of two the entire filter can be implemented using only a single true 16-bit multiplication, namely the multiplication of the measured velocity by β.

Now that the controller has an accurate estimate of both the motors positions ($\overline{q}$) and the velocities ($\dot{q}_m$), we can implement a control law which moves the motor to the desired target position q_t. The example code implements a simple PD controller of the form

$$y[t] = K_p(q_t[t] - \overline{q}[t]) - K_d\dot{q}_m \ , \tag{4.2}$$

where y is the signal to be sent to the motor and K_p and K_d are adjustable gain constants. This is a standard controller type which is often used to control systems which are reasonably well-behaved. It lacks an error-integrating term, and so it cannot fully compensate for steady-state external forces, such as gravity. Nevertheless it is often quite sufficient in situations where extreme precision is not a design requirement.

If the main control code were to loop as quickly as possible, beginning a new cycle as soon as the previous one finished, then minor variations in loop execution time would cause variations in controller performance and potential system instability. To prevent this, the example code uses one of the PIC's hardware timers to trigger the start of a control cycle every approximately 750μs. Some of this time will be spent in the interrupt service routine handling received serial bytes, but the filter

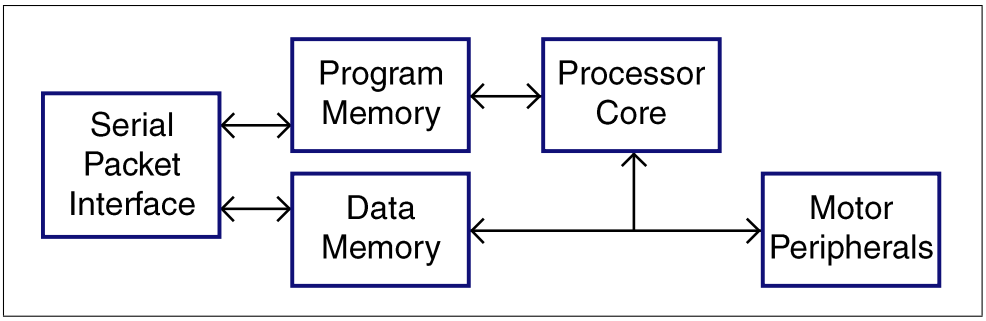


Figure 4.6: High-level processor and memory architecture used in the example FPGA code for the FPGA-based dual-port motor control board.

and control code will certainly be allowed 500μs per cycle. The code must access the measured position, velocity, and current values, write the pwm values, and perform three hardware-assisted multiplications per channel; bus access therefore occupies consumes almost half of the control loop's processing time. There is just enough time leftover to consider simple extensions to the control law, such as an integrating control term. Substantially more complex control laws cannot be supported by this control framework at such fast update rates.

4.2 FPGA-Based Controller

The FPGA-based dual-port control board described in Section 3.4 has no microcontroller. Instead, it has a single large FPGA which is responsible for all aspects of the board's operation. This provides the user with an extremely flexible control framework with potentially high computational performance, but it requires a considerably more sophisticated FPGA program than the one described in the previous section.

In addition to the FPGA, the board has a smaller CPLD whose responsibility is programming the FPGA with configuration data stored in flash memory. This part must also be programmed in an HDL such as Verilog, just like the FPGA. However this code is extremely simple by comparison: it need merely read data bytes out of the flash one at a time and present them to the FPGA, incrementing an address counter until programming is complete. We will not discuss the details of this CPLD code any further here.

- `main` — Top-level processor architecture.
  - `memory` — Dual-port data and program memory.
    - `packetrx` — Processes received data packets.
      - `serialrx` — Serial byte receive module.
    - `packettx` — Generates data packets for transmission.
      - `serialtx` — Serial byte transmit module.
  - `periphmod` — Peripheral bus controller.
    - `analog` — Controls SN74LV4051 muxes and ADS8320 A/Ds.
    - `decoders` — Contains sixteen quadrature decoders and bus logic.
      - `simpledec` — Quadrature velocity decoder module.
    - `pwmmod` — Contains sixteen PWM generators and bus logic.
      - `pwmgen` — PWM generator module for H-bridge control.
  - `procalu` — Processor Arithmetic Logic Unit (ALU).

Table 4.2: Structure of the example Verilog code for the main FPGA of the FPGA-based dual-port motor control board.

4.2.1 FPGA Code Overview

There are many possible structures for a motor controller implemented in an FPGA. For instance, a naïve design might route the sensor inputs directly to multipliers, adders, and so forth, and then route the results of that computation directly to the PWM outputs. Unfortunately, such a design would rapidly consume all the resources of an FPGA—even one as large as the Virtex XCV800 present on these boards—because of the large size of complex computational units such as multipliers. Moreover, in a design of that sort any change in the control structure would require re-synthesis and re-implementation of the code, which may take on the order of an hour for a complex design.

To address the first problem, a design can incorporate only a limited number of computational units, and the data can be routed between them as needed. This technique is known as *microsequencing*. To solve the second problem, the user can be allowed to specify the manner in which the data is routed between the computational units at run-time. A flexible microsequenced architecture which interprets user-specified sequencing commands one after the other is nothing other than a microprocessor. That is, the most sensible FPGA code structure for flexible motor control consists of a processor surrounded by peripherals which operate the hardware. This is

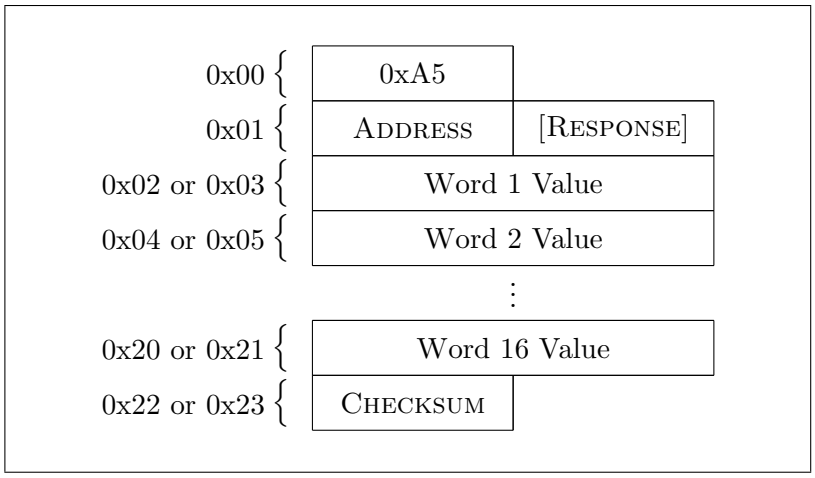


Figure 4.7: FPGA communications packet format. The Response byte exists only in packets sent *to* the FPGA.

essentially the same structure as that used in the PIC-based control board discussed earlier, except the processor has been moved inside the FPGA and the slow 8-bit bus has been eliminated. The example design discussed here adopts this structure.

The parallels between this design and that developed for the PIC-based controller go beyond broad structural similarity. The peripheral code developed for the PIC-based controller's support FPGA is used almost as-is in this design; it has been modified to support two eight-channel motor control boards instead of only one, but this modification is straightforward and we will not discuss these modules any further here. Communications with the host PC is also handled similarly in the two designs. The packet reception and transmission modules operate entirely independently of the main processor, exchanging data with the processor through a large shared memory block. Even the internal processor architecture, discussed in the next section, is patterned after the architecture of the PIC microcontroller line.

The module hierarchy for this code is given in Figure 4.2. The communications modules `packetrx` and `packettx` implement packet reception and transmission using a serial UART consisting of a a reception module `serialrx` and a transmission module `serialtx`. The `periphmod` module and the other modules contained in it correspond to the support FPGA in the PIC-based design, though without the additional math coprocessing features which are not needed here. The remaining modules (`main`, `memory`, and `procalu`) contain the processor itself and associated memory and bus control logic. The basic system design, shown in Figure 4.6, consists

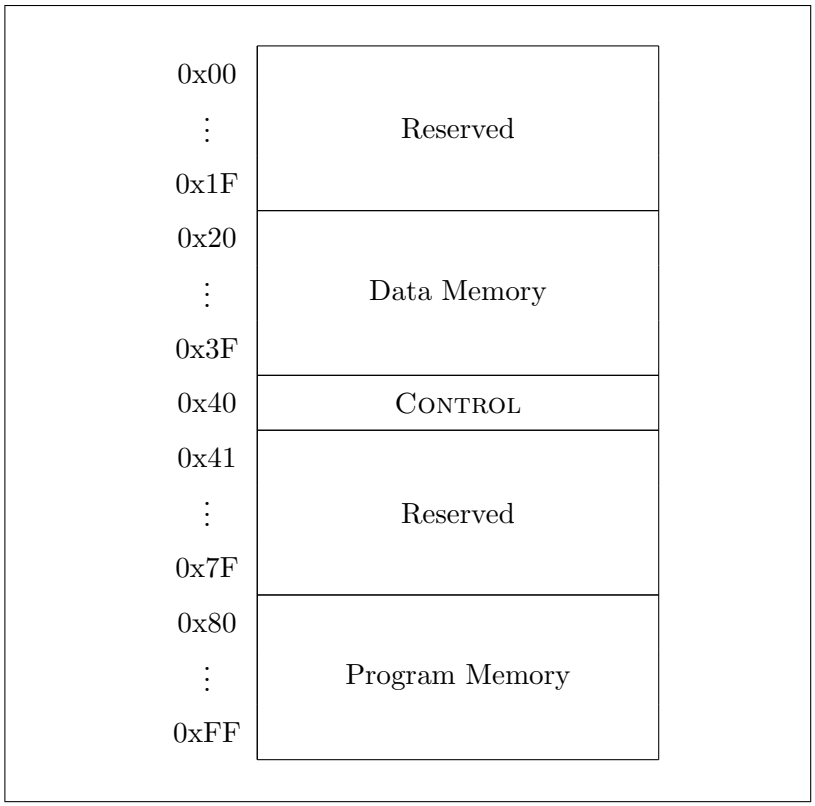


Figure 4.8: FPGA serial communications address space.

of a processor attached to separate program and data busses. This is known as a Harvard architecture, and offers higher and more predictable performance than the alternative von Neumann architecture based around a single memory bus. Many DSP and embedded controllers, including the PIC microcontrollers, use a Harvard architecture for this reason.

The Xilinx Virtex FPGAs, such as the XCV800 used here, include blocks of dual-port memory known as SelectRAM. The `memory` module contains a four-kilobyte region of these blocks to be used for program memory and a one-kilobyte region to be used for data memory. One port of each block is provided to the processor, via the program and data memory busses, and the other is provided to the serial communications modules. The host computer can therefore modify the controller's program code, configure various parameters in data memory, or read the contents of that memory, all without interrupting the processor in any way.

Communications over the serial port uses a packet format very similar to the one used by the PIC-based controller. This format is shown in Figure 4.7. The packet

format is slightly different for the two directions of communication. When the host PC sends a packet to the FPGA, it specifies the address range of memory to write (Address) as well as the address range of memory which the FPGA should send back in a response packet (Response). When the FPGA issues the response it specifies the address range of the data contained in the packet, but the Response field is missing. The response packets are therefore one byte shorter than the originally-transmitted packets, avoiding the problems encountered in the PIC-based design associated with PC and controller clock drift. It is important to note that the word values appear in the packet low-word-first, but that each value is represented with big-endian (high-bit-first) bit order.

The address space seen by the host computer consists of 256 blocks each containing sixteen 16-bit words. A map of this address space is shown in Figure 4.8. Here program and data memory appear as part of a single address space, though the processor internally accesses them via two separate busses. The Control word allows the host PC to exert high-level control over the processor; at present the only bit used is the low bit, which acts as a processor enable bit. This bit is used to halt the processor's execution when the host PC wishes to modify the contents of program memory. This is especially important at power-up, when it prevents the processor from executing until the control program has been loaded.

The processor sees this memory in an entirely different way. The data memory space is of course separate from the program memory space as far as the processor module is concerned. Further, the address space of the processor's data bus must contain not only the shared data memory but also the various motor peripherals. The memory bus is sixteen bits wide, and therefore requires an 11-bit address bus. The data bus is also sixteen bits wide, and has a 10-bit address bus. The upper half of this data memory space is general-purpose RAM shared with the communications system, while the lower half contains the peripherals. This processor data address space is shown in Figure 4.9. For simplicity, the peripherals may also be thought of as occupying the first reserved block of the serial communications address space as well, in which case the processor data address space corresponds precisely to the first quarter of the serial communications address space. However it is not actually possible to access these peripherals directly from the communications modules.

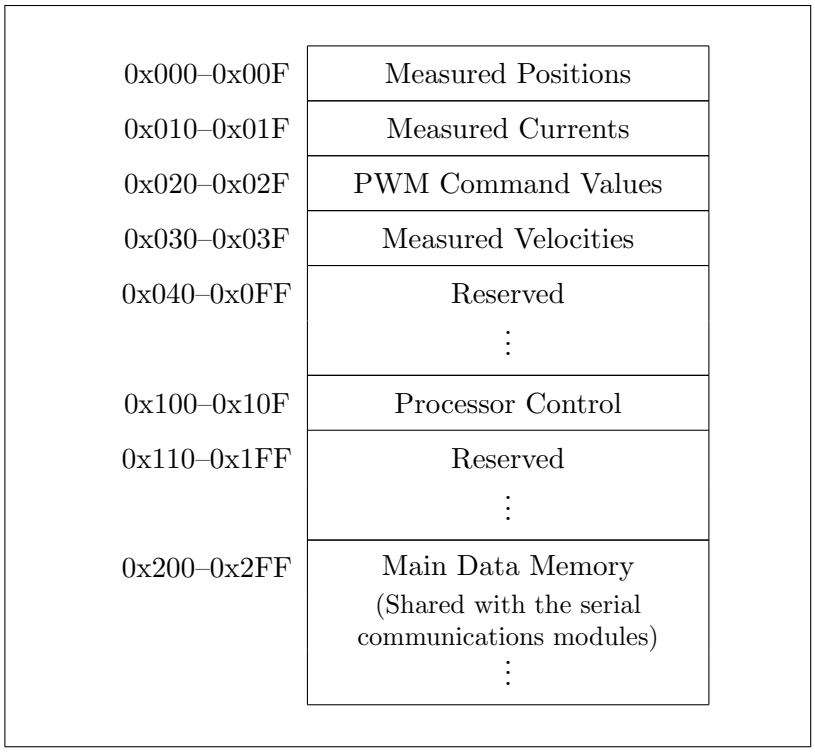


Figure 4.9: FPGA processor data address space.

4.2.2 Soft-Core Processor

The processor at the heart of this architecture is based loosely on the PIC microcontroller line. This similarity should help ease ease the transition from one to the other for future programmers. The processor has a simple RISC architecture; the execution of each instruction is straightforward, but the instruction set is powerful enough to allow sophisticated computation. A diagram of the processor architecture is shown in Figure 4.10. This simplified diagram omits the control lines running from the output of program memory to virtually every part of the processor. In particular, various bits of each instruction are used to control the three multiplexers, the Arithmetic Logic Unit (ALU), and the write enable lines for the registers and data memory.

Readers familiar with PIC microcontrollers will recognize many features of this architecture. Like all processors, it includes a special Program Counter register (PC) which increments once per instruction unless instructed otherwise. The output of this register selects the address of the next instruction to be executed. In addition to the control bits mentioned above, the instruction specifies an address in data memory and may also specify a literal value to be passed directly to the ALU. Each instruction

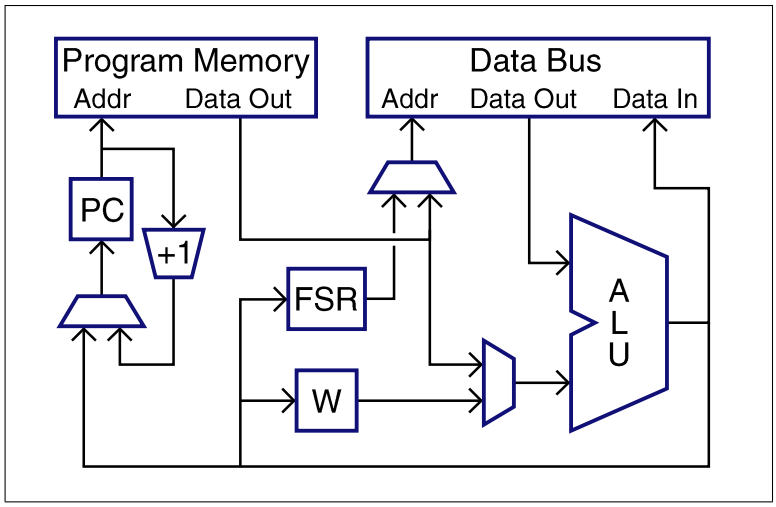


Figure 4.10: Simplified diagram of the processor architecture used in the example FPGA code for the FPGA-based dual-port motor control board.

only contains one memory address, and so normally binary memory operations must take advantage of a special register, known as the Accumulator (W). The first input of the ALU is always the output of data memory, and the second input can be either the contents of W or a literal value included in the instruction. The ALU then performs a chosen arithmetic operation on the two values and presents the result at its output, which is routed back to data memory and to the accumulator among other places. One of those destinations will in general have its write enable activated for each instruction.

An additional special register, known as the File Select Register (FSR), is provided to allow indirect memory addressing. The value contained in this register can be used as the address of a location in data memory to access. Program jumps, table look-ups, and other such operations can be implemented by writing the value computed by the ALU directly to the Program Counter. The special function registers can be accessed by reading or writing to locations in the Processor Control block of data address space in the range 0x100-0x10F. The exception to this rule is the accumulator W, which can always be accessed directly regardless of the address specified in the instruction.

A map of the Processor Control address space is shown in Figure 4.11. As mentioned above, the Program Counter (PC) contains the address of the next instruction to execute. The File Select Register (FSR) contains an address of memory to be

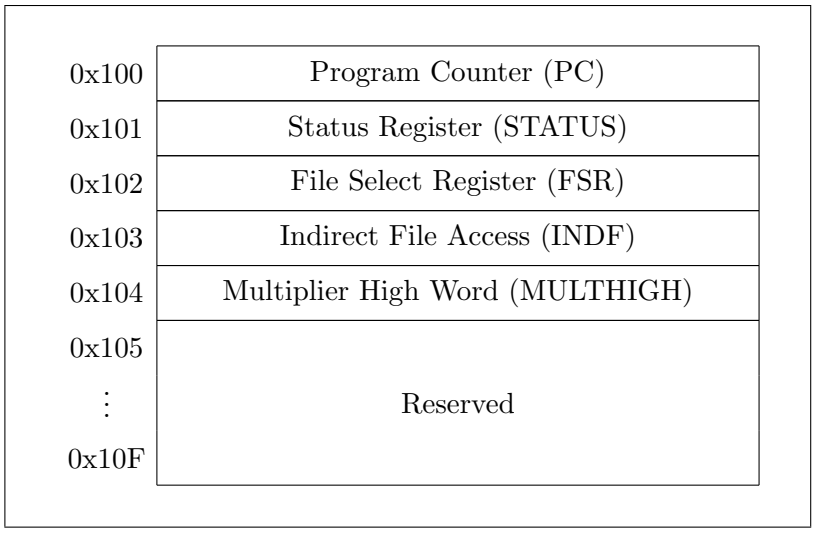


Figure 4.11: Processor Control register map.

accessed indirectly, and the Indirect File Access (INDF) "register" provides that access. That is, when an instruction specifies INDF in its address field, the processor switches the data bus address multiplexer so that the value of FSR is used instead.

The Status Register and the Multiplier High Word register both provide access to several extra outputs of the ALU. The Status Register (STATUS) contains the Zero and Carry bits, as shown in Figure 4.12. The Multiplier High Word register (MULTHIGH) contains the high word of the result of the most recent multiplication operation. The product of two 16-bit numbers is of course a 32-bit number; the multiply instruction itself returns the low sixteen bits, as is the standard behavior for 16-bit processors, but this mechanism provides access to the high bits which would otherwise simply be thrown away.

There are several key differences between this processor and the PIC microcontroller. First, this is of course a 16-bit processor rather than an 8-bit processor. Also, in place of the PIC's four-stage instruction pipeline, this processor uses a simpler two-stage pipeline. During the first stage the processor decodes the current instruction and fetches the required data, and during the second stage it computes and writes the result while fetching the next instruction. Unlike the PIC, this processor's instructions are the same width as the data bus, sixteen bits. Instructions cannot therefore contain literal values in the same way as the PIC, in which the value takes the place of the address field. Instead, literal instructions are twice as long as other instructions; the second word contains the literal value itself. When the processor decodes the first

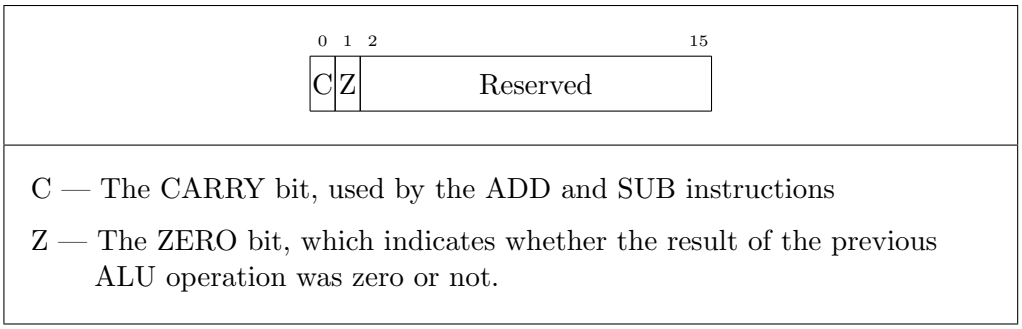


Figure 4.12: The Status Register (STATUS), located at data address 0x101.

instruction during the first execution stage, it checks to see whether the instruction has a literal value attached. If it does, it reads the second word from program memory during the second stage and passes it to the ALU for processing. In this case it must increment the Program Counter during both stages instead of only once, so that at the end of the second stage it still points to the next instruction. Though literal instructions therefore consume twice as much program memory as other instructions, they execute in the same amount of time.

The ALU has one additional output, which is used to implement code branching. If the instruction being executed is a bit-test instruction then the ALU computes a SKIP output, which tells the processor whether or not to skip the next instruction. If the ALU asserts SKIP then next instruction is in fact still executed but all write enables lines are forced low. This prevents the instruction from having any effect; it as though the instruction had never executed, but one instruction cycle has nevertheless passed. The processor has a Sleep mode which the user may enter by executing a special SLEEP instruction. This halts all execution of the processor until it is awakened by an external 1.3kHz timer signal generated by the motor peripheral module. This can be used to enforce a fixed control loop speed and to resynchronize the processor with the motor peripherals, resulting in more predictable control loop behavior.

4.2.3 Instruction Set

The instruction set of this processor is in many respects simpler than that of the PIC16F877 microcontroller used in the PIC-based control board, but it is in other respects more powerful as well. It lacks the special function-handling instructions CALL and RETURN, and in fact has no hardware stack at all. This is not a problem, however, because the processor has enough memory to implement a software stack to

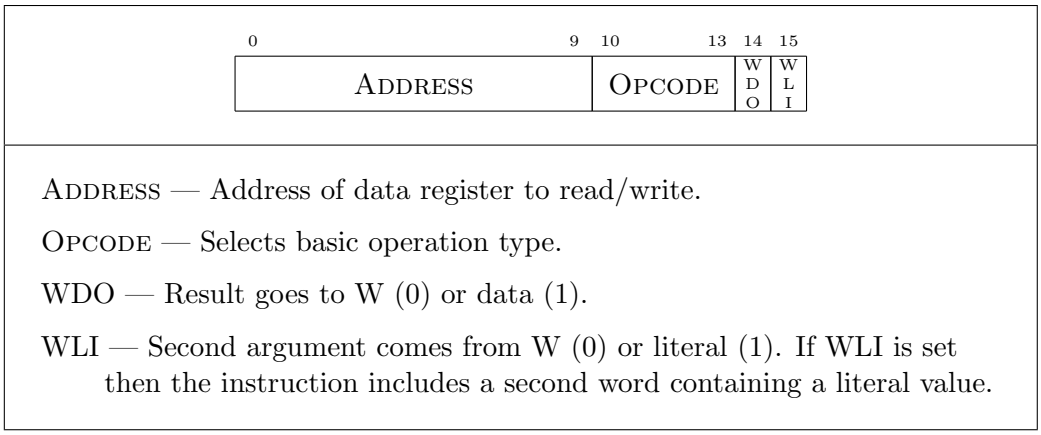


Figure 4.13: Processor instruction format.

achieve the same functionality in a more flexible way. Even the GOTO instruction does not appear here as a separate instruction; it can be implemented by using the other instructions to write directly to the Program Counter. An assembler for this processor might choose to provide GOTO as a mnemonic for either the MOVFLF or the ADDFLF instruction, and the reference assembler that was developed to test the processor does exactly that.

One advantage of this instruction set over the PIC's is that because literal values are not stored in the same place as data addresses within the instruction it is possible to have instructions which access data and include a literal argument simultaneously. For instance, this processor can initialize a register to an arbitrary literal value with a single instruction; this operation would require two instructions in the PIC, one to load the literal into the W register and a second to write the value to data memory. Additionally, the instruction format allows any arithmetic operation to take a literal argument, whereas the PIC only allows select literal operations. Lastly, this processor supports more powerful mathematical operations than the PIC; in particular it has a single-cycle multiply unit and a barrel shifter.

The instruction format is shown in Table 4.13. The Opcode field and WDO and WLI bits specify the instruction type, while the Address field specifies the address of the data register that is to be read and/or written by the instruction. The WDO and WLI bits are primarily responsible for routing data to and from the ALU, while the Opcode field is primarily responsible for selecting a particular ALU operation. However there are certain exceptional cases which handle the few special-function instructions that the processor does support.

Opcode	Mnemonic	Description
0x0	SLEEP	Put the processor in sleep mode.
0x1	XOR	Logical exclusive-or (ARG1 $\otimes$ ARG2).
0x2	MOV1	Copy first input (ARG1).
0x3	MOV2	Copy second input (ARG2).
0x4	AND	Logical and (ARG1 $\wedge$ ARG2).
0x5	IOR	Logical inclusive-or (ARG1 $\vee$ ARG2).
0x6	BSR	Barrel-shift right (ARG1 $\gg$ ARG2).
0x7	BSL	Barrel-shift left (ARG1 $\ll$ ARG2).
0x8	ADD	Twos-complement sum (ARG1 + ARG2). (Carry bit goes in CARRY.)
0x9	SUB	Twos-complement difference (ARG1 $-$ ARG2). (Borrow bit goes in CARRY.)
0xA	MUL	Unsigned product (ARG1 $\times$ ARG2). (High word goes in MULTHIGH.)
0xB	—	Reserved.
0xC	BTSC	Skip if bit ARG2 of ARG1 is clear.
0xD	BTSS	Skip if bit ARG2 of ARG1 is set.
0xE	—	Reserved.
0xF	NOP	No operation.

Table 4.3: Interpretation of the OPCODE instruction field. (ARG1 and ARG2 are the instruction arguments, the first always coming from data memory and the second coming from the W register or an instruction literal depending on the value of the WLI bit.)

Table 4.3 describes the various possible values of the OPCODE field. The values 0x1–0xA (XOR–MUL) all represent normal arithmetic operations: the ALU performs the selected operation on the given inputs and the result is stored in the given output location. The special instruction SLEEP places the processor in Sleep mode, as described in the previous section. The special instruction BTSC and BTSS take their inputs in the usual way but do not generate outputs; instead they instruct the processor to skip the following instruction if the specified bit is set or clear. The special instruction NOP is intended as a dummy operation and has no effect. The

two reserved opcodes are both treated as NOP but should not be used to allow for code-compatible extensions of this processor architecture. In particular, the opcode value 0xB is reserved for a hardware divider, the next logical improvement.

Each arithmetic operation has four variants, corresponding to the four combinations of values of the WDO and WLI bits. The assembly mnemonics for these variations have a simple form, consisting of one letter for each of the two inputs and one letter for the output: FWW, FWF, FLW, and FLF. In all four cases the first argument comes from the register file (F), but the second argument may come from the accumulator (W) or a literal (L), and the output may be sent to the accumulator (W) or registers (F). This convention mirrors that of the PIC assembly language, but features a higher level of symmetry: any arithmetic operation can have any input/output combination.

Since the MOVA and MOVB operations each ignore one of their arguments completely, it does not make sense for them to have the same assembly syntax as the other arithmetic instructions. Further, the WLI bit has no effect on the MOVA operation. Therefore the reference assembler combines the eight MOVA/MOVB instruction variants into six different MOV instructions. Likewise, the bit test operations BTSC and BTSS do not produce an output, and so the WDO bit does not affect them; therefore there are only four distinct bit test instruction types instead of eight. The complete instruction set supported by the reference assembler, which adopts both these simplifications, is shown in Table 4.4.

4.2.4 Control Firmware

An example sixteen-channel motor control program was developed in this assembly language, both to test the hardware and for immediate use controlling *Leonardo*. The features of this program are very similar to those of the PIC control program developed for the PIC-based control board as described in Section 4.1.2. It implements sixteen position filters of the form given in equation 4.1 and sixteen PD position controllers of the form given in equation 4.2. Communications between the processor and the host PC is of course facilitated by the hardware in the design; no special serial communications code must be written to use it. The program's memory layout is shown in Figure 4.14; this is all the host PC needs to know in order to control the program.

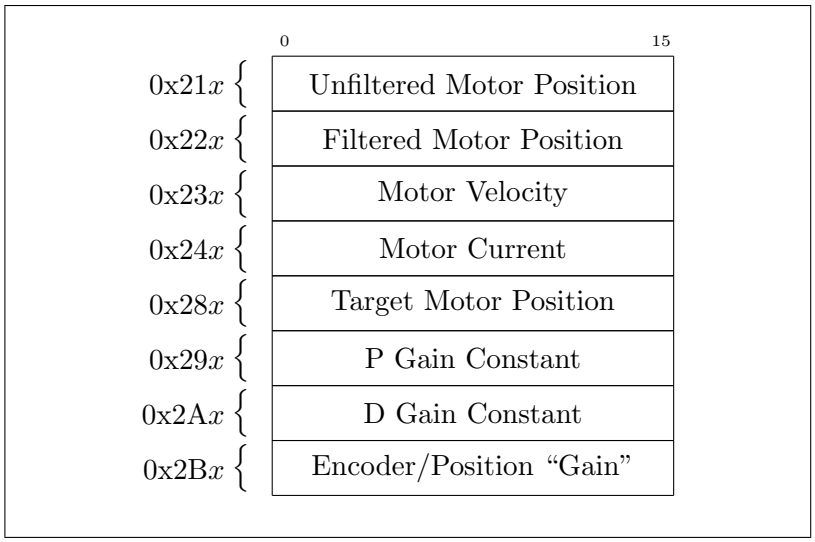


Figure 4.14: FPGA control program shared memory layout. (Channel number x.)

The only interesting difference between this code and the corresponding PIC code is that this code is considerably shorter and simpler. This is because the motor peripherals are located directly on the processor data bus, because the processor supports multiplication directly, and because the serial interface is handled in hardware as well. This is the great advantage to integrating a soft-core processor directly into a design: the processor and its peripherals may be optimized for the tasks the processor is expected to perform. This substantially simplifies program development.

XORFWW	r	Bitwise exclusive-or: W = *r $\otimes$ W (Sets ZERO bit)
XORFWF	r	Bitwise exclusive-or: *r = *r $\otimes$ W (Sets ZERO bit)
XORFLW	r,l	Bitwise exclusive-or: W = *r $\otimes$ l (Sets ZERO bit)
XORFLF	r,l	Bitwise exclusive-or: *r = *r $\otimes$ l (Sets ZERO bit)
MOVFW	r	Memory move/copy: W = *r (Sets ZERO bit)
MOVFF	r	Memory move/copy: *r = *r (Sets ZERO bit)
MOVWW		Memory move/copy: W = W (Sets ZERO bit)
MOVWF	r	Memory move/copy: *r = W (Sets ZERO bit)
MOVLW	l	Memory move/copy: W = l (Sets ZERO bit)
MOVLF	r,l	Memory move/copy: *r = l (Sets ZERO bit)
ANDFWW	r	Bitwise and: W = *r $\wedge$ W (Sets ZERO bit)
ANDFWF	r	Bitwise and: *r = *r $\wedge$ W (Sets ZERO bit)
ANDFLW	r,l	Bitwise and: W = *r $\wedge$ l (Sets ZERO bit)
ANDFLF	r,l	Bitwise and: *r = *r $\wedge$ l (Sets ZERO bit)
IORFWW	r	Bitwise inclusive-or: W = *r (Sets ZERO bit)
IORFWF	r	Bitwise inclusive-or: *r = *r $\vee$ (Sets ZERO bit)
IORFLW	r,l	Bitwise inclusive-or: W = *r $\vee$ (Sets ZERO bit)
IORFLF	r,l	Bitwise inclusive-or: *r = *r $\vee$ (Sets ZERO bit)
BSRFWW	r	Bit-shift right: W = *r $\gg$ W (Sets ZERO bit)
BSRFWF	r	Bit-shift right: *r = *r $\gg$ W (Sets ZERO bit)
BSRFLW	r,l	Bit-shift right: W = *r $\gg$ l (Sets ZERO bit)
BSRFLF	r,l	Bit-shift right: *r = *r $\gg$ l (Sets ZERO bit)
BSLFWW	r	Bit-shift left: W = *r $\ll$ W (Sets ZERO bit)
BSLFWF	r	Bit-shift left: *r = *r $\ll$ W (Sets ZERO bit)
BSLFLW	r,l	Bit-shift left: W = *r $\ll$ l (Sets ZERO bit)
BSLFLF	r,l	Bit-shift left: *r = *r $\ll$ l (Sets ZERO bit)
ADDFWW	r	Twos-complement sum: W = *r + W (Sets ZERO and CARRY bits)
ADDFWF	r	Twos-complement sum: *r = *r + W (Sets ZERO and CARRY bits)
ADDFLW	r,l	Twos-complement sum: W = *r + l (Sets ZERO and CARRY bits)
ADDFLF	r,l	Twos-complement sum: *r = *r + l (Sets ZERO and CARRY bits)
SUBFWW	r	Twos-complement difference: W = *r - W (Sets ZERO and CARRY bits)
SUBFWF	r	Twos-complement difference: *r = *r - W (Sets ZERO and CARRY bits)
SUBFLW	r,l	Twos-complement difference: W = *r - l (Sets ZERO and CARRY bits)
SUBFLF	r,l	Twos-complement difference: *r = *r - l (Sets ZERO and CARRY bits)
MULFWW	r	Unsigned product: W = *r $\times$ W (Sets ZERO bit and MULTHIGH word)
MULFWF	r	Unsigned product: *r = *r $\times$ W (Sets ZERO bit and MULTHIGH word)
MULFLW	r,l	Unsigned product: W = *r $\times$ l (Sets ZERO bit and MULTHIGH word)
MULFLF	r,l	Unsigned product: *r = *r $\times$ l (Sets ZERO bit and MULTHIGH word)
BTSCFW	r	Skips next instruction if Wth bit of *ris clear
BTSCFL	r,l	Skips next instruction if lth bit of *ris clear
BTSSFW	r	Skips next instruction if Wth bit of *ris set
BTSSFL	r,l	Skips next instruction if lth bit of *ris set
GOTO	label	Implemented with MOVLF (absolute goto) or ADDFLF (relative goto)
SLEEP		Places the processor in Sleep mode until next timer tick.
NOP		No operation

Table 4.4: FPGA soft-core processor assembler instruction summary.

Chapter 5

Support Software

Powerful and flexible hardware is of no use without equally powerful and flexible software to control it. The details of high-level behavior and motion-planning software are beyond the scope of this thesis; however it must be easy for high-level software programmers to access the important features of the hardware without getting bogged down in details. The software libraries and protocols described in this chapter are designed to make programming robot control systems as easy as possible.

There are three main requirements for a general-purpose motor control software layer for interactive robots. At the lowest level, it must be straightforward to add driver-level support for new motor control hardware. One level up, it must be easy to configure a particular collection of hardware for use with a given robot. At the top, there must be a clean interface to the chosen high-level motion-planning and behavior system. Each of these three interfaces should be kept as isolated as possible from the others; most importantly, the high-level interface should shield the behavior system from the inner details of the hardware and drivers.

The `motor_system` motor control library described in Section 5.1 aims to satisfy these requirements. It has been implemented in C++ to achieve high performance in a clean object-oriented framework. This library may be accessed directly by other C++ code. However in many situations that is not desirable; for instance, the high-level behavior system may be written in a language such as Java that cannot conveniently link against C++ libraries. Therefore a second network-based interface has been provided. Network-based inter-process communication is useful in many situations in interactive robotics. The Intra-Robot Communications Protocol (IRCP), described in Section 5.2, is an extensible protocol designed to be used throughout an interactive

robot, not just at the motor-control layer. By standardizing on a single protocol, programmers can focus on developing new behaviors instead of writing network code. Moreover, having a single inter-process communications framework allows remote debugging tools to be developed that can monitor the state of the entire robot with ease, even when it is distributed across many computers running software developed by different programming teams.

This document is only intended to provide an overview of the `motor_system` library and IRCP. The most up-to-date and complete documentation can be found on the project website at `http://robotic.media.mit.edu/motor/`.

5.1 Motor System Software Layer

The `motor_system` library described in this section provides a clean programming interface to complex motor control hardware. It shields the high-level programmer from specific hardware details while allowing the engineer to adjust those details with ease. Adding support for new hardware is straightforward, though only the hardware described in the previous chapter is supported at the present time.

5.1.1 Motor System Overview

The primary purpose of this general-purpose library is to simplify the interface between the programmer and the engineer. To the programmer, a robot should look like a single motor system consisting of a collection of joints with simple descriptive names and standard properties such as position and velocity. The programmer should be able to develop motion and behavior algorithms for the robot in these terms alone, even if in reality the robot uses several different types of motor drivers and actuators configured in a variety of different ways. Further, the engineer should be able to adjust or reconfigure the motor system without bothering the programmer.

The high-level (public) interface to this library has been kept as simple as possible. All the details of low-level configuration are handled automatically through an ASCII configuration file provided by the engineer. This configuration file describes the attached hardware, including hardware-specific configuration parameters, and assigns a name and other properties to each joint. Since this file is processed at run-time, the engineer can completely reconfigure the robot's hardware without the programmer even having to recompile. The engineer may also make fine adjustments while the

Function	Description
`motor_system(string&)`	Constructor, takes configuration file name.
`motor_system::iterator find(string&)`	Find a motor by name.
`motor_system::iterator begin()`	Points to the first motor in the tree.
`motor_system::iterator end()`	Points beyond the last motor in the tree. (Returned by `find()` for invalid motor name.)

Table 5.1: Public methods of the `motor_system` class (partial listing).

robot is running via the library's network interface without the programmer having to take any action at all.

5.1.2 High-Level (Behavior) Interface

At the core of the library is the `motor_system` class. The only parameter passed to the constructor of this class is the location of a configuration file to use; all other low-level details are hidden from the user. This class is treated (at this level) as a container class, containing some number of `motor` objects. The class provides the standard set of methods provided by other container classes in the standard library, and the motor objects are accessed by means of iterators of type `motor_system::iterator`. The most important public methods of this class are listed in Table 5.1.

Each motor object has a simple interface consisting of easy-to-use and self-explanatory functions such as `set_target_position()`, `get_velocity()`, and `enable()`. All function parameters and return values are given in real-world units chosen by the author of the configuration file. Since not all joints may have the same semantics, each motor object also provides methods to determine what functions it supports. For instance, the drive motors in a wheeled robot may not support a notion of absolute position, and this would be indicated by a certain return value of functions such as `supports_set_target_position()`. Since the programmer should be able to operate the robot without an engineer present, certain key low-level features (most notably the ability to enable and disable the motors) have been exposed at this layer. The most important public methods of this class are listed in Table 5.2.

Function	Description
`void enable()`	Enable the joint
`void disable()`	Gracefully disable the joint
`void kill()`	Immediately disable the joint
`void set_target_position(float)`	Set the joint's target position
`float get_position()`	Get the joint's actual measured position
`float get_velocity()`	Get the joint's actual measured velocity

Table 5.2: Public methods of the `motor` class (partial listing).

5.1.3 Mid-Level (Configuration) Interface

As described above, all configuration of the motor system library is performed through a single configuration text file. This file has a hierarchical structure: each section contains simple name/value pairs and other (sub)sections. The top level of the hierarchy contains parameters which apply to the entire motor system, such as a brief name and description of the configuration or the definition of an IRCP interface. The file then contains one section corresponding to each piece of motor control hardware. The details of the format of these sections depend on the type of hardware being described; after being parsed, the contents of each section is simply passed to the corresponding low-level driver. However in general each section will contain parameters configuring the hardware unit as a whole and subsections corresponding to each motor the unit controls.

Broadly speaking, there are two types of data contained in the configuration file. One the one hand there are low-level configuration parameters, such as specific controller gains, which are specific to the low-level drivers being used and which should be invisible to the upper layers. On the other hand there are parameters such as joint names and units definitions which define the interface to the behavior system. In this sense, this configuration file, and the `motor_system` class as a whole, provide a buffer layer between the low-level drivers and the high-level behavior code.

A simple fully-functional example configuration file illustrating these features is shown in Figure 5.1.

```
name = "TestMotorSystem"
description = "Simple 1-DOF Motor System"

MotorDriverA {                          // Define a motor driver
    type = "SerialMedusa16"             // Specify hardware type
    description = "Sixteen-channel Serial Medusa"
    serial_device = "/dev/cua0"         // Specify serial port
    controller = control.bin            // Other hardware parameters...

    MotorA1 {                           // Define a motor
        channel = 1                     // Hardware channel number
        name = "TestMotor"              // The motor's name
        description = "A Test Motor"    // A simple description
        pgain = 1000                    // Controller's P gain
        dgain = 1000                    // Controller's D gain
        egain = 110                     // Other controller params...
        max_position = 50000
        min_position = 10000
        center_position = 30000         // Define the zero and scale
        ticks_per_unit = 20000          //  of the position units
    }
}
```

Figure 5.1: Example `motor_system` configuration file for a trivial 1-DOF robot.

5.1.4 Low-Level (Driver) Interface

Many details of each low-level hardware driver will, of course, be specific to the hardware being driven. However, the motor system library makes it simple to design a driver which may be configured by the standard configuration system. All configurable objects are derived from the abstract base class `configurable_object`, which provides a standard mechanism for setting and querying object parameters by name. This class has two important subclasses, `motor` and `abstract_motor_container`.

To create a driver for a new piece of hardware, you must create two new classes. The bulk of the driver code belongs in a class derived from `abstract_motor_container`. This class would generally be responsible for spawning a separate thread that runs the actual control code and communicates with the hardware. The details of each particular motor should be stored in a class derived from `motor`; this class is re-

sponsible for implementing the simple interface functions for a single joint in a thread-safe manner. Programmers interested in adding support for new hardware should familiarize themselves with the existing drivers for the Medusa hardware; the techniques used in that code should be applicable to most motor control hardware.

5.1.5 Abstract Tree Structure

The `motor_system::iterator` class must be able to iterate over the entire motor tree without regard to the fact that different `motor` objects may in fact have different types. More importantly, those `motor` objects may be stored in containers that are themselves of different types, rendering traditional iterators unusable. To address this problem, the motor system library is based on a highly-abstract tree container type called `abstract_tree<T>`.

Three special types of iterator are provided to support this framework. The first, `abstract_iterator<T>`, provides a common interface for all iterators which point to objects derived from a given abstract base class `T`. No single normal iterator can iterate across different container types, but it is possible to use a pointer to `abstract_iterator<T>` for this purpose. The associated iterator creation and destruction can be messy, and so a second iterator type, `meta_iterator<T>`, is provided. This class contains a pointer to `abstract_iterator<T>` and hides all memory management issues from the user. A single iterator of type `meta_iterator<motor>` is thus capable of pointing to any motor in the motor system tree. The third iterator type, `indirect_iterator<T>`, is provided so that containers which contain pointers-to-`T` can have iterators which dereference to `T` instead; it simply provides an extra level of dereference.

The `abstract_tree<T>` template class is a tree container which takes advantage of these abstract iterator types. The branches of the tree may be members of arbitrary subclasses of `abstract_tree<T>`, and the leaves may be members of arbitrary subclasses of the base type `T`. In addition to the standard branch and leaf iterators, the primary iterator class, `abstract_tree<T>::iterator`, is capable of iterating across all the leaves of the tree even if the tree's branches have heterogeneous type. The abstract type `abstract_motor_container`, described above as the base class for all motor driver classes, is simply an empty class derived from `abstract_tree<motor>` and `configurable_object`.

5.2 The Intra-Robot Communications Protocol

The control system for a complex interactive robot often consists of many modules running on multiple computers. Inter-module communication can become extremely complicated in this environment. Early control systems for *Public Anemone* and *Leonardo* used a mixed bag of statically-linked, dynamically-linked, and one-off network-based techniques. This design approach, in which is a new chunk of inter-module communications code is written for every new communications link in the system, is fundamentally not scalable. The number of links in a system increases roughly as the square of the number of modules; soon the code base would be dominated by communications code.

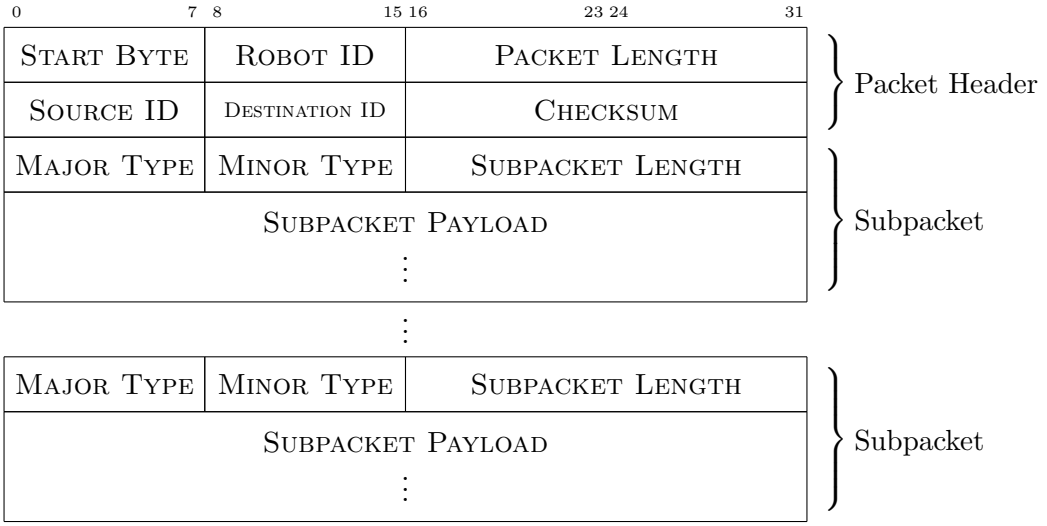
To address this problem, a standard inter-process communications protocol was developed, known as the Intra-Robot Communications Protocol, or IRCP. This protocol is sockets-based, so that it may be used both for communication between multiple processes on a single computer and between different computers. It is general enough to be used throughout a robot control system, but it is structured enough to allow considerable code reuse.

5.2.1 IRCP Overview

The Intra-Robot Communications Protocol (IRCP) was designed with three principles in mind: extensibility, reusability, and simplicity. The protocol is *extensible* in the sense that it is easy to add support for new channels of communication, with new syntax and new semantics, as needed. It is *reusable* because it suggests a hierarchical code structure which makes code reuse painless. Its *simplicity* will hopefully be immediately apparent.

At the heart of IRCP is the IRCP packet format. In a sense, this format is the beginning and the end of the IRCP specification. IRCP packets may be sent over any channel, and may carry virtually any sort of payload. However, for the protocol to be successful in simplifying inter-process communication users should adhere to certain guidelines in deciding how to use it. These guidelines will be discussed below; first, we present the packet format itself, shown in Figure 5.2.

Users of IRCP must agree on several values. First, each of the robots that may coexist on a given network should be assigned a unique Robot ID in the range 0–255. Then the various control processes for each robot should be broken down into



Start Byte — A fixed value (0xA5) signifying the start of an IRCP packet. Allows IRCP packets to be sent over a byte stream.

Robot ID — An identifier specifying the robot to which this packet applies. Prevents inter-robot interference if multiple robots are coexisting on a single network, and allows one process to control multiple robots.

Packet Length — The total packet length in bytes. Must be a multiple of four.

Source ID — An identifier specifying the module sending the packet. Intended primarily for use in system debugging.

Destination ID — An identifier specifying the module to which the packet is being sent. Intended primarily for use in system debugging.

Checksum — A 16-bit checksum used to verify packet integrity. The 16-bit twos-complement of the sum of all other 16-bit words in the packet.

Major Type — Identifies the major type of the subpacket.

Minor Type — Identifies the minor type of the subpacket. The meaning of this byte depends on the major subpacket type.

Subpacket Length — The subpacket length in bytes.

Subpacket Payload — The meaning and size of this field is determined by the subpacket type. The subpacket payload must be padded to the smallest multiple of four bytes greater than or equal to the given Subpacket Length.

Figure 5.2: Intra-Robot Communications Protocol packet format.

Major Type	Description
0	Low-level motion (joint-space)
1	Body sensations (skin/touch)
2	Vision
3	Audition and speech recognition
4	Vocalization and speech generation
5	High-level motion (gestural/postural)

Table 5.3: Currently assigned major subpacket types for the Intra-Robot Communications Protocol.

modules, each of which is assigned a Module ID from the same range. The Robot ID and the Source and Destination Module IDs are included in the header of each IRCP packet. In many situations this may seem unnecessary; there is often no ambiguity about who is sending a packet, or to whom it is being sent. Nevertheless these values should be included and checked, both to prevent miscommunication and to allow remote debugging tools to monitor and interpret the communication with ease.

The IRCP packet header also contains a start byte, the total packet length, and a checksum field. These fields may also be redundant in many situations, such as when the IRCP packet is sent via UDP. These fields exist primarily to allow IRCP packets to be sent over a byte-stream communications channel, such as a serial port, UNIX pipe, or TCP (as it is traditionally interpreted). These fields should be set and checked regardless of the underlying transport method used in order to reduce the chance misinterpreting erroneous packets. Ideally the IRCP packet-handling code would be kept entirely independent of the transport code, increasing code modularity and reusability.

The payload of each IRCP packet then consists of any number of IRCP subpackets. The type of each subpacket determines its internal format and semantics and is divided into a Major and Minor Type. This allows the IRCP semantics for, say, vision processing to be reworked without risk of disrupting other robot subsystems. Each subpacket header contains the total subpacket length in bytes as well as the packet type, so that unrecognized subpackets can be easily skipped over. A small number of major packet types have been defined at this time; these are listed in

Table 5.3. At the end of this section we will describe some minor types that have been defined for major type 0; the minor types associated with other major types are outside the scope of this thesis.

5.2.2 IRCP Subpacket Formats

The subpacket payload format associated with each subpacket type may be completely independent from the format used by other types, and is completely up to the user. Some subpackets may contain image data, others may contain joint position data, and others may require no payload at all. However, when defining a new minor type the programmer should be careful to look and see if an existing type uses a payload format that would be appropriate to the new type as well. In this case the code to format and interpret the subpacket payload may be reused immediately. If no existing packet type has an appropriate payload format, then the programmer must of course define a new one. Nevertheless, this should still be done with an eye towards code reuse. The packet format should be defined in as general a manner as is appropriate, and the packet-formatting code should be written modularly so that future packet types may take advantage of it as well.

The reference IRCP implementation, currently being completed in C++, provides a clean mechanism for subpacket code reuse. Each subpacket format is implemented in an abstract class, and each individual packet type using that format is simply derived from this class. This reference implementation presently supports several packet formats, many of which are used by the low-level motion commands of major type 0 described in the next section. These formats are listed in Table 5.4.

5.2.3 IRCP Major Type 0: Low-Level Motion Commands

Low-level motion control is the first application for which a full range of IRCP subpacket types has been defined. These all have major type 0; the full list of minor types which have been defined so far is provided in Table 5.5. Only some of the types listed in the table have been used to date in a real motor system. The rest have been defined to provide a more complete example of how to use IRCP.

By convention the low-level motor module operates in a slave mode only: it does not send any packets unless they are specifically requested by some other module, which we shall refer to as the master. One minor type, Request Response, is reserved

Type	Payload Description
No Payload	This subpacket format contains no payload. Note that the subpacket length is therefore four, not zero, because of the subpacket header.
Integer	A single 32-bit integer, transmitted in standard network byte order.
Float	A single 32-bit floating-point number in the standard IEEE format, transmitted in standard network byte order.
String	A single null-terminated ASCII string, padded with 0–3 zeroes to a multiple of four bytes.
Optional Integer	Interpreted either like type Integer or type No Payload depending on the subpacket length.
Optional Float	Interpreted either like type Float or type No Payload, depending on the subpacket length.
Optional String	Interpreted either like type String or type No Payload, depending on the subpacket length.
Integer Array	A collection of four-byte Integers. The number of values is determined from the packet length. A packet type may place restrictions on how many values may be present.
Float Array	A collection of four-byte Floats. The number of values is determined from the packet length. A packet type may place restrictions on how many values may be present.
Float Array	A collection of Strings. The number of values can only be determined by parsing and counting the strings. A packet type may place restrictions on how many values may be present.
Indexed Integer Array	A collection of pairs of Integers, alternating between index and value. Allows a few values of an Integer array to be sent without transmitting the entire array.
Indexed Float Array	A collection of Integer/Float pairs, alternating between index and value. Allows a few values of a Float array to be sent without transmitting the entire array.
Indexed String Array	A collection of Integer/String pairs, alternating between index and value. Allows a few values of a String array to be sent without transmitting the entire array.

Table 5.4: IRCP subpacket formats supported by the reference implementation.

Minor Type	Description
0x00	Request Response (payload: Integer). Sent to a motor module to request information. The payload is the minor type of the desired response.
0x01	Motor System Information (payload: Indexed String Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains the names of the joints controlled by the motor system, indexed by joint number.
0x02	Enable Motors (payload: Integer Array). Sent to a motor module to enable some or all of the motors. The payload contains the joint numbers of the motors to enable; an empty array indicates *all* motors.
0x03	Disable Motors (payload: Integer Array). Sent to a motor module to gracefully disable some or all of the motors. The payload contains the joint numbers of the motors to disable; an empty array indicates *all* motors.
0x10	Set Target Positions (payload: Indexed Float Array). Sent to a motor module to update some joints' control target positions.
0x11	Set Target Velocities (payload: Indexed Float Array). Sent to a motor module to update some joints' control target velocities.
0x12	Set Target Accelerations (payload: Indexed Float Array). Sent to a motor module to update some joints' control target accelerations.
0x13	Set Target Forces (payload: Indexed Float Array). Sent to a motor module to update some joints' control target forces/torques.
0x14	Set Target Stiffnesses (payload: Indexed Float Array). Sent to a motor module to update some joints' control target stiffnesses.
0x15	Set Target Viscosities (payload: Indexed Float Array). Sent to a motor module to update some joints' control target viscosities.
0x20	Set All Target Positions (payload: Float Array). Sent to a motor module to update all joints' control target positions.
0x21	Set All Target Velocities (payload: Float Array). Sent to a motor module to update all joints' control target velocities.
0x22	Set All Target Accelerations (payload: Float Array). Sent to a motor module to update all joints' control target accelerations.
0x23	Set All Target Forces (payload: Float Array). Sent to a motor module to update all joints' control target forces/torques.

Continued on next page....

Table 5.5: IRCP Minor Types defined for Major Type 0: Low-level motion.

Table 5.5 continued from previous page....

Minor Type	Description
0x24	Set All Target Stiffnesses (payload: Float Array). Sent to a motor module to update all joints' control target stiffnesses.
0x25	Set All Target Viscosities (payload: Float Array). Sent to a motor module to update all joints' control target viscosities.
0x30	Actual Positions (payload: Indexed Float Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains actual (sensed) joint positions.
0x31	Actual Velocities (payload: Indexed Float Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains actual (sensed) joint velocities.
0x32	Actual Accelerations (payload: Indexed Float Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains actual (sensed) joint accelerations.
0x33	Actual Forces (payload: Indexed Float Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains actual (sensed) joint forces/torques.
0x40	Actual Currents (payload: Indexed Float Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains actual (sensed) actuator currents.
0x41	Actual Temperatures (payload: Indexed Float Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains actual (sensed) actuator temperatures.
0x50	Joint Descriptions (payload: Indexed String Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains human-readable joint descriptions.
0x51	Joint Capabilities (payload: Indexed Integer Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains a bitfield describing the capabilities of each joint. See Figure 5.3 for a description of this field.
0x52	Joint Status (payload: Indexed Integer Array). Sent by a motor module in response to an appropriate Request Response subpacket. Contains a status bitfield for each joint. See Figure 5.4 for a description of this field.
0xFF	Emergency Stop (payload: No Payload). Initiates an emergency shut-down of the motor system.

for this purpose. The master may send any number of these subpackets at a time to request a variety of types of information from the motor system. In most cases all modules that communicate with a motor system will have agreed on a set of joint names *a priori*. The master will then request a Motor System Information subpacket from the motor system, which will list the names of the joints supported by the motor system and assign each one an integer index. The master should verify that the joint names are as expected and should take note of the indices. All future communication with the motor system will refer to each joint by its index for efficiency.

The master may now begin controlling or monitoring the state of the motor system. Two minor types allow the master to enable and disable the active control of each joint. There are also a range of minor types for controlling the target position, velocity, acceleration, force, stiffness, or viscosity. Not all of these parameters may be controllable in every joint, and any joint which is not capable of controlling for a particular parameter should ignore attempts to set that parameter. If a parameter value is given which is outside the range supported by a joint, the joint should clip the value where appropriate and ignore it otherwise. Some joints may support multiple control modes; for instance, a force-feedback actuator may be able to operate both in a position-control mode and in a force-control mode. In this case issuing a Set Target Positions or Set Target Forces command for that joint could cause it to enter the appropriate mode.

The Set Target Velocities and Set Target Accelerations types are not only intended for use with joints under pure velocity or acceleration control. They may also be supported for position-controlled joints as a way for the higher motion planning layers to provide hints to the low-level motion controllers about how the target position is expected to change in the near future. This is particularly important when the target position update rate is slow relative to the system dynamics. In this situation the low-level controller must use a smoothing filter to avoid the jerky quality of motion that is usually associated with low update rates, and velocity and acceleration hints can be used to alter the smoothing filter's behavior. (This feature is not implemented in the firmware described Chapter 4 because no high-level motion systems currently in use are capable of generating these hints.)

Two methods for setting these joint control values are provided. The first allows the master to set a parameter for only a given set of joints, by providing an indexed array of values, while the second allows the master to set the value of a particular

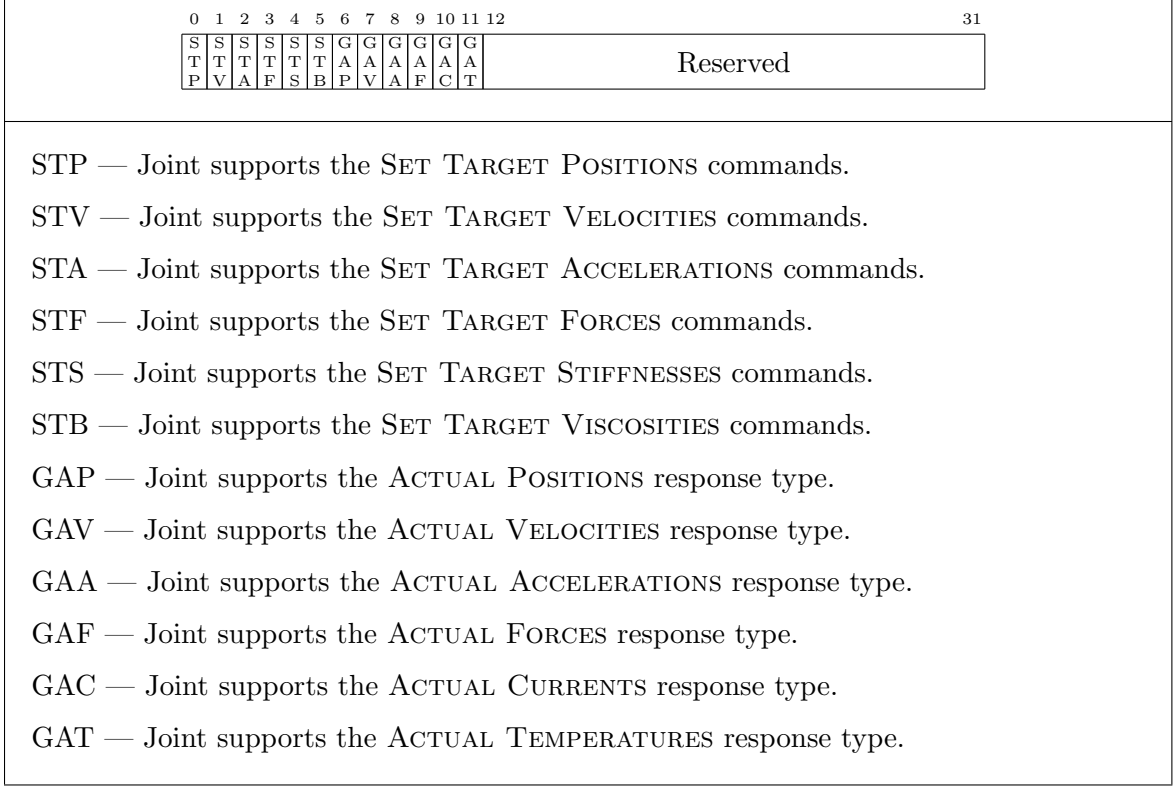


Figure 5.3: IRCP low-level motion Joint Capabilities bitfield format.

parameter for all joints at once. This second interface is provided as a convenience for use in relatively simple robots, and can of course be used only when the same parameter can be safely set for all joints at that time. The first interface can be used to allow two separate master modules to each have control of part of a robot. Of course, no two modules should attempt to control the same joint simultaneously, and so any two such modules must have some side-channel for arbitrating joint control.

The master controlling a particular joint would usually be expected to know the basic properties of the joint in advance. For instance, a robot that lacked force sensors would not generally be asked to control joint forces. Nevertheless, it is possible to learn more about each motor. Three minor types, which can be requested by a Request Response subpacket, provide additional information. This information might be particularly useful for general-purpose motor and robot debugging tools. The Joint Descriptions type provides a human-readable description of the function of each joint. These descriptions must be specified by the engineer in the configuration file of the motor system. The Joint Capabilities type specifies which parameters can

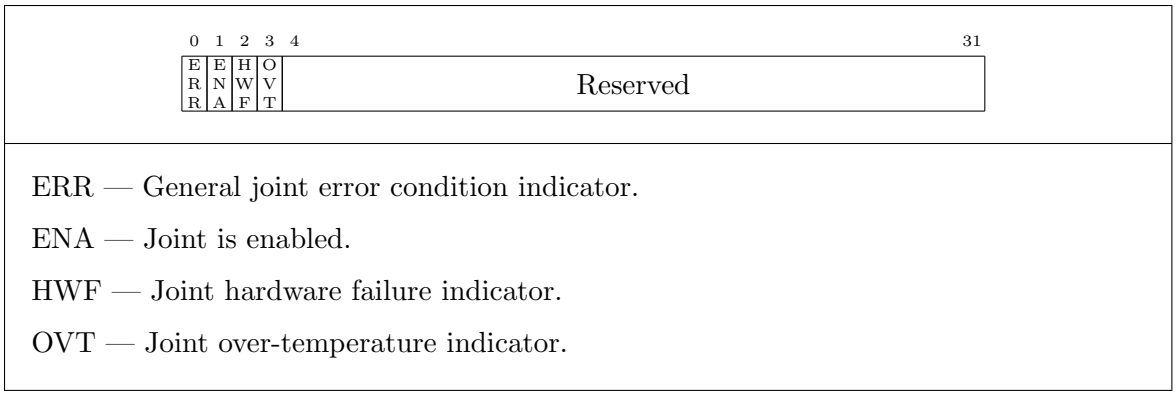


Figure 5.4: IRCP low-level motion JOINT STATUS bitfield format.

be controlled by each joint; the data is encoded in a bitfield as shown in Figure 5.3. Lastly, the JOINT STATUS type provides real-time status flags for each joint; this data is also encoded in a bitfield, as shown in Figure 5.4.

Chapter 6

Concluding Thoughts

There are a number of important areas for future work relating to this thesis. The simple uncoupled PD control algorithm that is currently used by all the controllers could be replaced with any of a wide range of more sophisticated control laws. The soft-core processor used in the FPGA-based control boards would need to be redesigned for the control law to be made considerably more complicated. Alternatively it could be replaced with an open-source or commercial core. In the long run, the best option would likely be to design a new control board based on one of the new FPGAs with an integrated hard processor core, such as many in the Xilinx Virtex II-Pro series.

A true multi-axis control system would also need some mechanism for multiple control boards to communicate directly. The RS-485 bus used on the FPGA-based control board could be used for this purpose, but appropriate hardware would be needed to interface this network to the host computer. If a higher performance communications scheme were used instead then it would be possible to use this same system to communicate with various sensors. A common high-speed sensorimotor network would also facilitate extremely rapid reflex responses.

Finally, other motor driver cards should certainly be designed to extend the Medusa line. In addition to higher-current drivers, support for brushless DC motors would be very useful. Added safety features, such as over-current protection and motor temperature sensing, would be useful in improving long-term robot reliability. Looking further ahead, force-feedback will almost certainly play an important role in future robots and requires special control hardware. A force-feedback system based on series-elastic actuators, for instance, requires an additional position sensor at each motor [PW95]. In the spirit of the Medusa line, the best solution would probably be

to add support for a large number of additional channels of analog and digital I/O which could be configured to suit the needs of each individual application.

All the hardware and software described in this thesis has been tested and used to some degree. However, no part of this system has been tested nearly enough to offer any real sense of its long-term reliability. I have great confidence in the core motor driver design, which has remained essentially unchanged since the first prototypes of *Public Anemone*. However this system contains a very large number of parts, and so it will be impossible to feel confident in its total reliability until after a long and arduous burn-in. For instance, the motor drivers for *Leonardo* contain a total of 256 IRF7470 MOSFETs, and this part has been found to have a non-negligible off-the-shelf failure rate. While it is possible to catch many problems on the test bench before the hardware makes it into a robot, others problems do not emerge until after extended use.

Now that the design, manufacturing, programming, and early testing are complete, the goal of the next few months will be to permanently install the final control hardware in both *Public Anemone* and *Leonardo*. This will also be the first chance to install the extremely compact S-model motor controllers inside *Leonardo*'s head, as the robot was sent back to California for modification and repair before the manufacture of those modules was complete. This will be an exciting culmination of the original work that developed into this thesis.

Appendix A

Board Schematics

The following figures show the complete Protel schematics used to generate the boards described in this thesis.

A.1 Eight-Channel Driver Pack

A.2 PIC-Based Single-Port Controller

A.3 FPGA-Based Single-Port Controller

A.4 FPGA-Based Single-Port Controller (Model S)

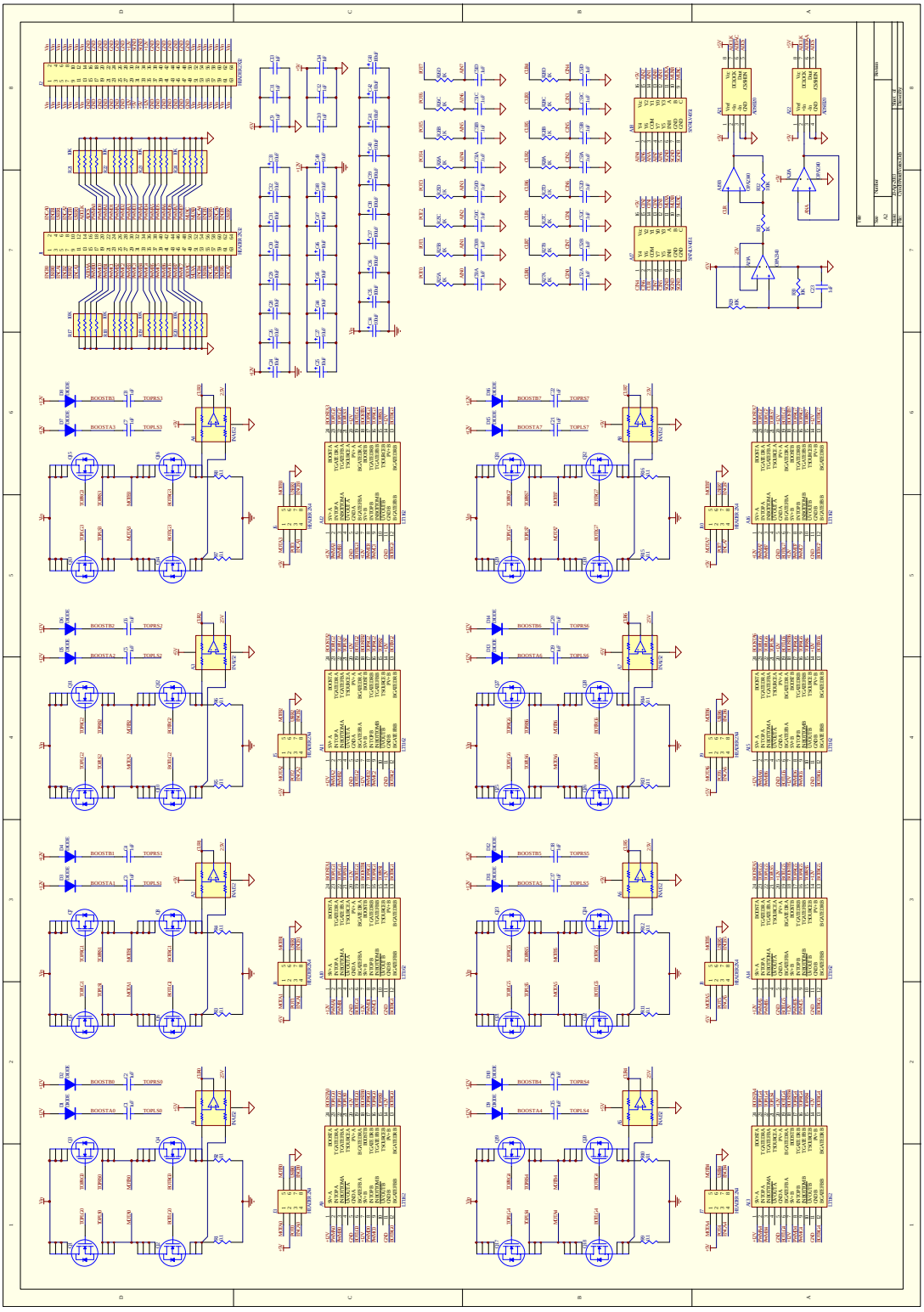


Figure A.1: Schematic of the Medusa Eight-Channel Driver Pack.

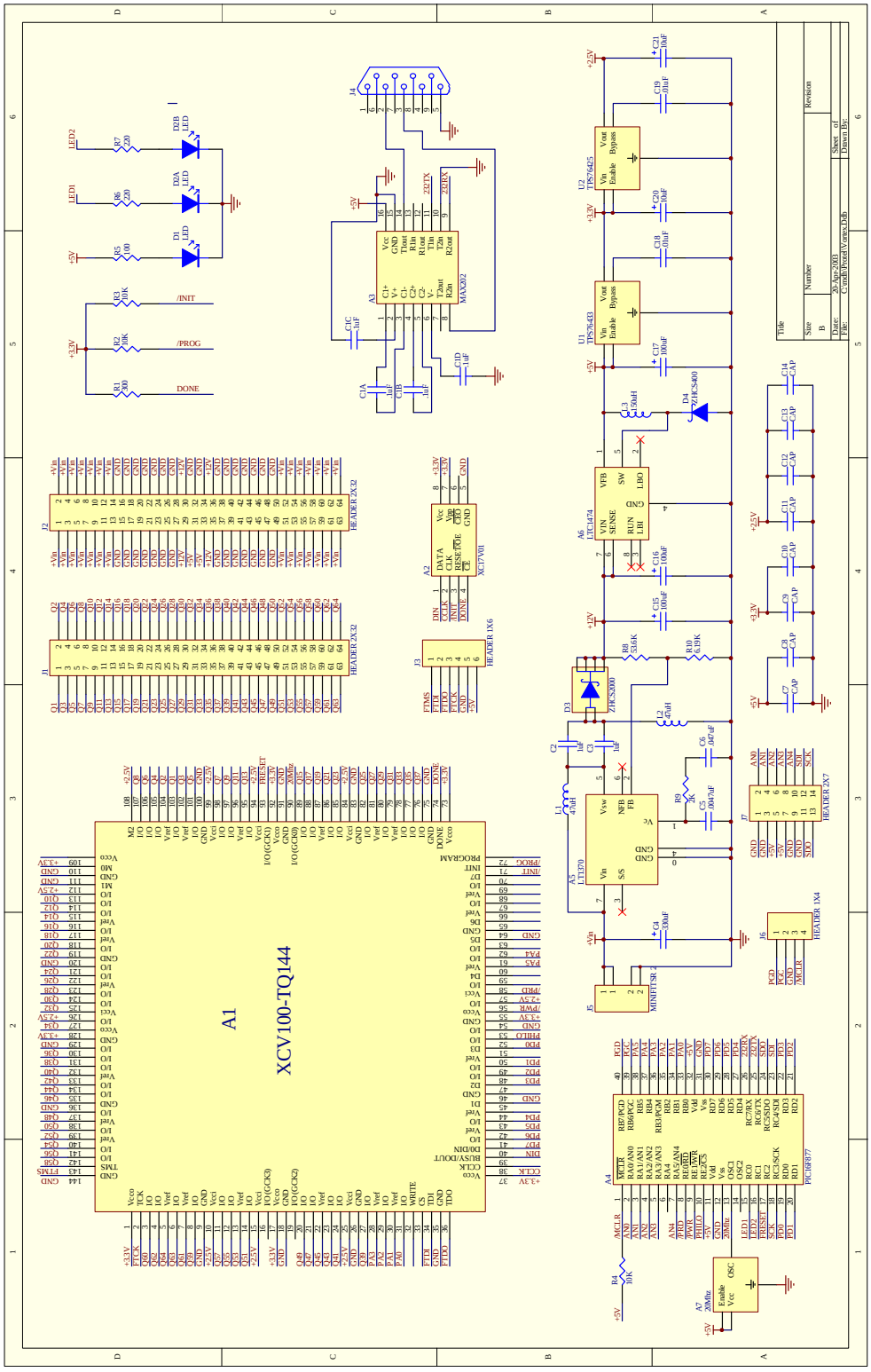


Figure A.2: Schematic of the Medusa PIC-Based Single-Port Controller.

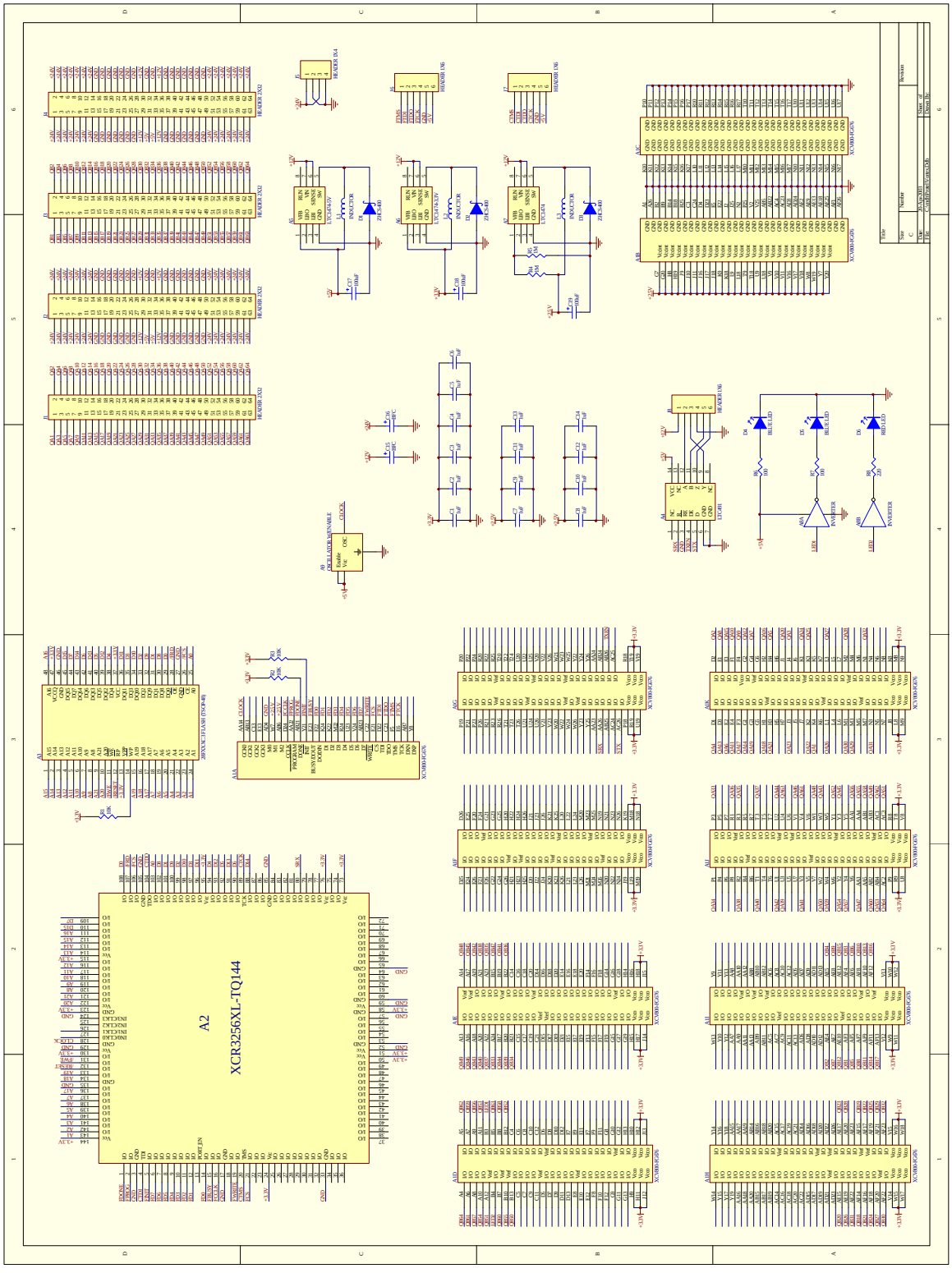


Figure A.3: Schematic of the Medusa FPGA-Based Dual-Port Controller.

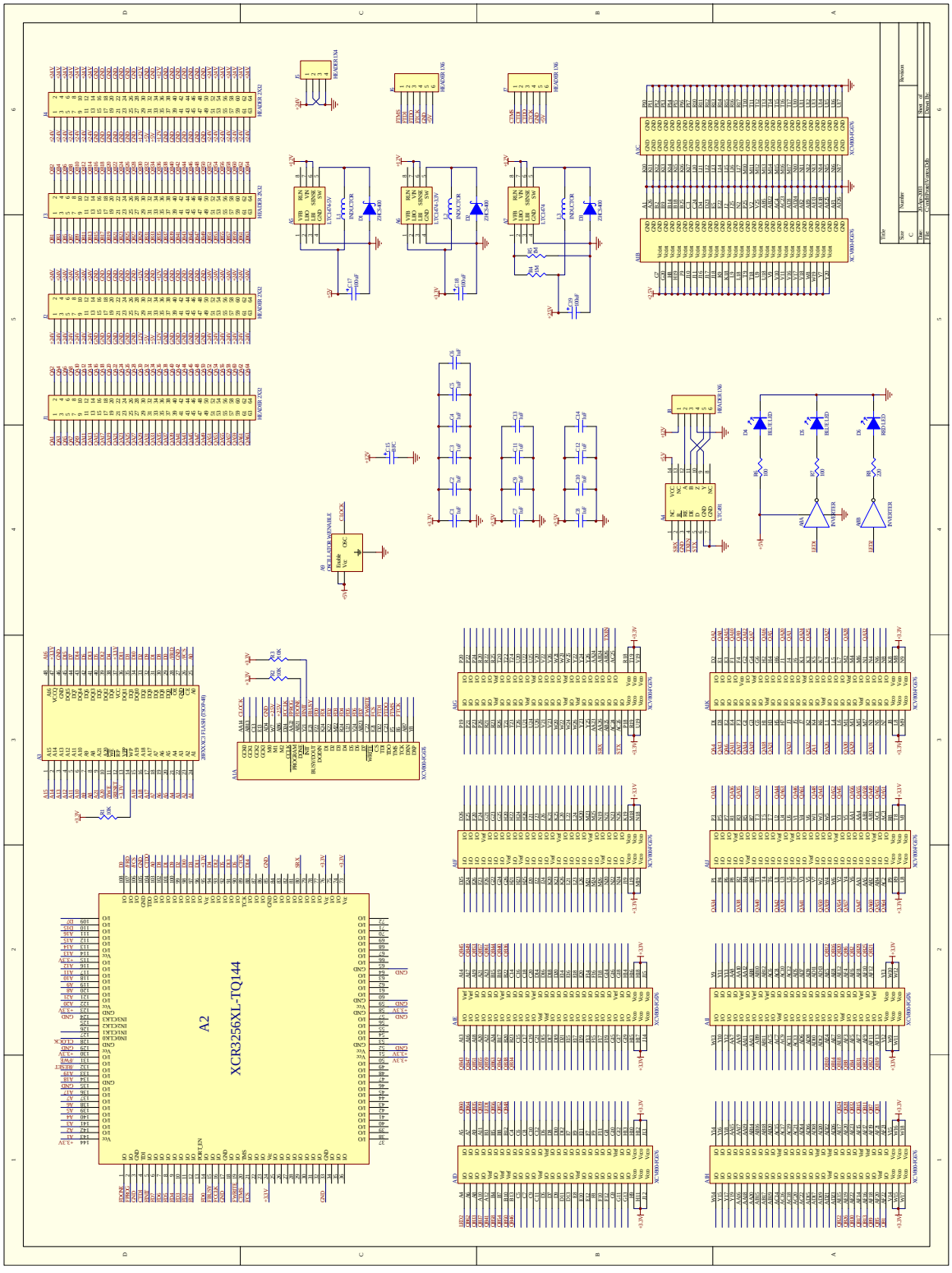


Figure A.4: Schematic of the Medusa FPGA-Based Dual-Port Controller (Model S).

Bibliography

[AAA+00] R.O. Ambrose, H. Aldridge, R.S. Askew, R.R. Burridge, W. Bluethmann, M. Diftler, C. Lovchik, D. Magruder, and F. Rehnmark. "Robonaut: NASA's Space Humanoid". *IEEE Intelligent Systems*, 15(4):57–63, Jul/Aug 2000.

[ABBS00] B. Adams, C. Breazeal, R. Brooks, and B. Scassellati. "Humanoid Robots: A New Kind of Tool". *IEEE Intelligent Systems*, 15(4):25–31, Jul/Aug 2000.

[Bac97] M. Bacon. *No Strings Attached: The Inside Story of Jim Henson's Creature Shop*. New York: Macmillan, 1997.

[BBM+98] R.A. Brooks, C. Breazeal, M. Marjanović, B. Scassellati, and M. Williamson. "The Cog Project: Building a Humanoid Robot". In C. Nehaniv, editor, *Computation for Metaphors, Analogy, and Agents*. Berlin: Springer-Verlag, 1998.

[Bre02] C. Breazeal. *Designing Sociable Robots*. Cambridge, MA: MIT Press, 2002.

[CD58] A. Chapuis and E. Droz. *Automata: A Historical and Technological Study*. London: B.T. Batsford Ltd, 1958.

[Joh02] Michael P. Johnson. *Exploiting quaternions to support expressive interactive character motion*. PhD thesis, Massachusetts Institute of Technology, 2002.

[KB99] C. Kline and B. Blumberg. "The Art and Science of Synthetic Character Design". In *Proceedings of the AISB1999 Symposium on AI and Creativity in Entertainment and Visual Art*, Edinburgh, Scotland, 1999.

[KNK+01] S. Kagami, K. Nishiwaki, J.J. Kuffner, T. Sugihara, M. Inaba, and H. Inouet. "Design, Implementation, and Remote Operation of the Humanoid H6". In D. Rus and S. Singh, editors, *Experimental Robotics VII*, pages 41–50. Berlin: Springer-Verlag, 2001.

[PW95] G.A. Pratt and M.W. Williamson. "Series Elastic Actuators". In *IEEE/RSJ Int. Conf. on Intelligent Robotics and Systems (IROS)*, 1995.

[SH94] Mark Salisbury and Alan Hedgcock. *Behind the Mask: Secrets of Hollywood's Monster Makers*. London: Titan Books, 1994.

[SWA+02] Y. Sakagami, R. Watanabe, C. Aoyama, S. Matsunaga, N. Higaki, and K. Fujimura. "The Intelligent ASIMO: System Overview and Integration". In *IEEE/RSJ Int. Conf. on Intelligent Robotics and Systems (IR0S)*, 2002.