Otto: A Low-Cost Robotics Platform for Research and Education

by

Edwin B. Olson

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering In Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 23, 2001

Signature of Author ……...
Department of Electrical Engineering and Computer Science
May 1, 2001

Certified by ……...
Lynn Andrea Stein
Associate Professor of Computer Science
Thesis Supervisor

Accepted by ……..
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Department of Electrical Engineering and Computer Science

Otto: A Low-Cost Robotics Platform for Research and Education

by
Edwin B. Olson

Submitted to the Department of Electrical Engineering and Computer Science

May 23, 2001

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering In Electrical Engineering and Computer Science

ABSTRACT

We have developed a high-performance, low-cost robotics platform named "Otto" designed for robotics researchers and educators. The platform consists of a microcontroller board as well as software development and runtime tools. Greater computational power allows more complex artificial intelligence and control algorithms to be implemented, enabling more sophisticated robot behavior. Over the course of two manufacturing runs, the platform has evolved significantly. This thesis documents the platform's evolution, lessons learned during its design, and future planned enhancements. In addition, we taught a robotics engineering class at MIT using the Otto platform. The course and the students' feedback on the Otto platform are also discussed.

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor, Electrical Engineering and Computer Science

Table of Contents

List of Figures

List of Tables

Acknowledgements

We are indebted to Hitachi Semiconductor, the MIT EECS department, and PADS Software for funding portions of our work. We would also like to thank Altera, Acroname, ISSI, Maxim, National Semiconductor, and Pulse Engineering for providing discounted or free components.

This work was completed under the supervision of Professor Lynn Andrea Stein whose work is supported by the National Science Foundation under CISE Educational Innovation Grant #99-79859. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We would also like to acknowledge the MIT undergraduate students who gave us invaluable feedback on our platform during the MASLab robotics course in January, 2001.

Collaboration Note

This report documents the work of two MIT students, Edwin Olson and Max Bajracharya. In this document, I will be discussing both the overall project as well as my specific contributions. I highly recommend Bajracharya's Master's thesis, as it is a companion to this one.

1. Introduction

Many researchers and educators are drawn to robotic systems. For artificial intelligence researchers, robots are an excellent bridge from theory to the real world. For example, a research robot may incorporate path-planning and machine vision algorithms in order to explore an environment. Robots must also physically interact with their environment, posing interesting problems for the mechanical engineer and control systems designer. Educators are attracted to robots for two basic reasons: students find the idea of working with robots inherently appealing, and robotics provide a rich union of many disciplines which can enhance and accelerate the learning process.

It is economically infeasible for everyone wanting to build a robot to develop a custom platform. Consequently, there is a need for general-purpose platforms that can be used by many people. Development of a simple platform can require man months of engineering labor, and the unit cost of manufacturing is usually exorbitant for small quantity runs. Of course, it is impossible to meet the demands of *every* user; a user doing stereo vision research might require an array of dedicated DSP chips, an impossible burden on a teacher working with high school students. However, many users seem to have surprisingly similar requirements, which has led to the success of several commercially available general-purpose platforms. These general-purpose platforms save users the cost and effort of developing their own platforms. In addition, the production of general-purpose platforms can take place on a larger scale, resulting in dramatically lower unit costs.

After experimenting with several existing general-purpose platforms, we realized that by using modern components, we could produce a platform with superior capabilities without increasing the cost. For many of these platforms, their limited capabilities are a result of the technology available when the boards were designed—in some cases more than 10 years ago. It seemed clear that a better platform could be built with today's technology.

In 1998, we began work on a new general-purpose platform, with the goal of providing superior capabilities by leveraging modern electronics components. We began with a blank slate, unconcerned with maintaining backwards compatibility with existing platforms, but with an intense desire to mimic the simplicity and user-friendliness of the most popular existing platforms. Our goal was to provide a massive improvement in computational horsepower, memory capacity, and an enhanced ability to interface with motors, servos, and sensors.

This thesis documents the design, implementation, and ongoing evolution of our new robotics platform. Our platform includes both the controller board itself and the software that runs on it. Section 1.1 summarizes many of the popular platforms used in robotics work today. The heart of this thesis, the design and implementation of the Otto controller board, is discussed in Section 2, with subsections discussing the board's various functional modules. Section 3 is devoted to the software development environment and

runtime utilities, which accelerate application development. In Section 4, we describe the robotics engineering course we taught during January 2001, which used the Otto board.

1.1 Existing Robotics Platforms

Several robotics platforms are currently available with widely varying capabilities. At the low-end are microcontroller chips that can be used without any additional hardware at a cost of less than $5. High-end platforms include single board computers running Linux costing thousands. Existing platforms can give a good indication of what features users need and want, so we examined several that were roughly the same price range we were targeting: several hundred dollars or less, an amount easily afforded by individuals.

A few dollars can buy a simple microcontroller that can be used as a single-chip solution for robots. A good example is the Microchip PIC line, which integrates a fairly fast CPU (<5MIPS) with various combinations of timers, A/D converters, and D/A converters. Timers are immensely useful for both controlling motors and processing sensor data from devices like rotary optical encoders. A/D converters allow users to process analog voltages, which are output by many types of sensors. For these simple microcontrollers, however, memory is often a limiting factor. Robots building a map of their environment or performing other complex tasks can very easily exhaust the memory capacity of most PICs (typically <<1KB), and most models do not support adding additional external memory.

Other platforms add external components to a microcontroller to increase their functionality. However, because every additional component added to the board increases the size, complexity, and cost of the platform, the features offered by the microcontroller itself usually comprise the bulk of the board's features. Additional components usually only add minor features, or make the board easier to work with (e.g., adding user-friendly connectors.) Therefore, the microcontroller at the heart of a controller board is one of the most important considerations.

A minimalist robotics platform called The Cricket [1], developed at the MIT Media Lab, is barely larger than a 9V battery. Its primary component is a small PIC microcontroller. The board adds only IR communications, a pair of buttons, and header to ease interfacing to the PIC. However, its small size and extreme low cost make it an appealing platform to those developing simple devices. Programming is performed in assembly or a simplified Logo or BASIC dialect.

A popular robotics platform is the Lego Mindstorms RCX Kit [2]. It has a relatively fast CPU at 16MHz, and a modest suite of on-chip peripherals including A/D converters and a few timer channels. However, the RCX module has a limiting 512 bytes of RAM, and is packaged in a way that artificially limits its total functionality to three sensor inputs and three actuator outputs. Several development environments are available, including a child-friendly graphical environment and various simplified versions of C, Logo, and other languages.

Widely used around MIT is the Handyboard [3] and its relative, the 6.270 Controller Board. Based on a design from around 1989 [4], the Handyboard's feature set was limited by the technology available at the time. The CPU is a rather sluggish Motorola 68HC11 at 2MHz. The Handyboard provides an impressive set of on-board peripherals compared to the Cricket and Mindstorms RCX, and makes all of them easily accessible to the user via header connections. An easy-to-use development environment called "Interactive C" [5] is almost universally used, but since Interactive C is an interpreted language, code runs more slowly than would compiled code.

The Compaq Skiff[6] is a robotics platform designed around the Compaq Personal Server platform. It has an Intel StrongArm SA110 running at up to 206MHz and 16MB of DRAM which gives it a large advantage in computational power over the platforms described above. However, the SA110 is a general-purpose microprocessor, not a microcontroller; it does not provide any features useful for robotics, requiring many components to be added externally. The resulting Skiff robotics board is extremely complex, large, and expensive. It occupies twice the circuit board area of the Handyboard, even though integrated circuits are mounted on *both* sides of each circuit board. The board runs Linux or NetBSD, and development can be done in virtually any standard development environment, including the GNU toolchain (C, C++) or Java. The Skiff is not commercially available.

Another possibility for robot developers is to use a single board computer. Many single board computers are available commercially, many with 486 or Pentium processors. Few offer built-in peripherals useful for robotics, but these can be added with expansion cards. Solutions based around general-purpose microprocessors are typically considerably more expensive than those based around microcontrollers since they require additional components to add robotics-specific features.

The features of the above platforms are summarized in Table 1. While some feature categories are easily quantified (e.g., RAM capacity), other feature categories are virtually impossible to quantify (e.g., extensibility). We have therefore subjectively and qualitatively summarized several categories. While cost is usually quantifiable, two of the boards are not yet commercially available—consequently, we have loosely estimated the cost of the platforms and provided a qualitative comparison.

	Cricket	**Lego RCX**	**HandyBoard**	**Skiff**
CPU	Microchip PIC16F84 (1 MHz)	Hitachi H8/3292 (16 MHz)	Motorola 68HC11E9 (2 MHz)	Intel SA110 (200MHz)
RAM	68 bytes	512 bytes	32 KB	Up to 16MB
Sensor Capability	Poor	Okay	Excellent	Excellent
Motor Driving	Okay	Okay	Excellent	Excellent
Servo Driving	Poor	Poor	Good	Excellent
Extensibility	Okay	None	Excellent	Good
Cost	Very Low	Moderate	Moderate	High

Table 1. Comparison of existing low-cost robotics platforms.

2. Design of a New Platform

We designed and implemented a new robotics platform to provide superior computational performance while maintaining the low-cost and easy-to-use nature of existing robotics platforms. This board would later be named the "Otto", the name of the bus driver on the television series "The Simpsons", but also a homophone of "*auto*nomous".

Before examining each part of the controller board in detail, it is useful to briefly identify the major subsystems (see Figure 1). The Otto board combines a Hitachi SH-2 microcontroller that directly interfaces to 2MB of DRAM (expandable to 16MB). The board supports 23 analog inputs, seven directly into the SH-2 plus sixteen more through a 16-to-1 analog mux. A Complex Programmable Logic Device (CPLD) provides the interface to digital peripherals such as buttons and the LCD. The SH-2 also provides two serial channels.

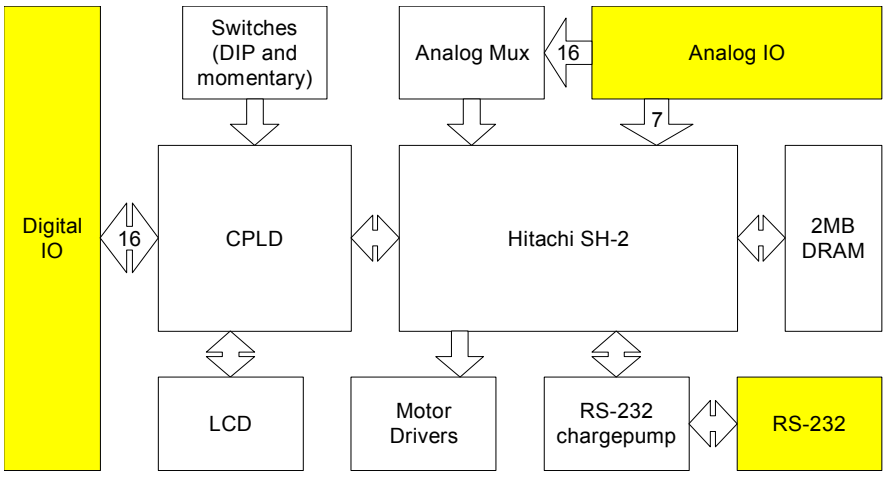


Figure 1. Block diagram of Otto controller board, revision 2.

While other configurations are possible, the most typical configuration of the Otto controller board has capabilities below in Table 2. An asterisk denotes that the feature is planned for the next board revision.

CPU	Hitachi SH-2 704x @ 30MHz
Memory	2-16MB DRAM 512KB of FLASH*
LCD	Standard character-based LCDs supported.
Reconfigurable Hardware	Altera 7128 or Xilinx FPGA*
Analog Inputs	23
Digital I/O	16
High Current DC Motors (PWM)	4 channels, 2A each
Servo controllers	6
Quadrature Phase Decoders	2
Serial Ports	2

Table 2. Otto board capabilities.

2.1 Computational Core

We expect that many users will be drawn to the Otto because of its impressive computational capabilities. The computational power of the SH-2 is complemented by both volatile and persistent memory.

2.1.1 Microcontroller

The most important design decision for a low-cost robotics platform is the selection of the microcontroller. In a low-cost platform, it is essential to keep the number of components on the circuit board at a minimum; thus, the features provided by the microcontroller itself form the bulk of the features of the whole board.

We considered a large number of microcontrollers, including the Motorola 68332, Motorola 6812 (a modified version of the processor used in the HandyBoard), Hitachi H8 (used in the RCX), and the StrongArm SA110 (used in the Skiff), but ultimately settled on the Hitachi SH-2. The SH-2 is a fast RISC CPU with a rich set of on-board peripherals well suited to robotics applications. The on-board peripherals allow very simple board design; a minimalist implementation could consist of just a power supply and the SH-2. While the SH-2 represents an excellent combination of performance and built-in peripherals, our feature goals for the microcontroller board require adding external components. The capabilities of the SH-2 are summarized in Table 3.

Speed	29.49 MHz
Architecture	32 bit pipelined RISC, in-order single issue
Instruction word	16 bit, 2 register format
On-Chip RAM	4 KB
DRAM controller	Up to 16MB
On-Chip FLASH	256 KB
A/D Converter	10 bits x 8 channels
Package	FP-144 or FP-112
Serial Ports	2
Timers	5 Channels (each can do several PWM outputs or a single more complex operation)

Table 3. Capabilities of the SH-2 7045 Microcontroller

The SH-2 7045 is available in two different packages, both of which are fine pitch surface mount variants. The chips are almost identical in functionality, except the FP-144 exports a full 32 bit data bus rather than the 16 bit data bus used by the FP-112. We elected to use the FP-112 on our first two board revisions rather than the much smaller pitch FP-144 in order to simplify assembly in the hopes that users might be able to assemble their own boards. Since both parts require professional assembly anyway, the ease of assembly is somewhat irrelevant—future versions will use the FP-144 since it allows for a higher-speed memory subsystem and therefore faster overall performance.

2.1.2 Memory

The SH-2 supports both SRAM and DRAM off-chip memory. SRAMs provide significantly faster access times than DRAMs; SRAMs can provide data in a single cycle

whereas DRAMs typically require two or three cycles to access a random location. However, SRAM is considerably more costly than DRAM. We wished to provide megabytes of memory, which would be very costly if implemented in SRAM, so we chose to use DRAM.

We included a 2MB DRAM chip, organized as 1Mx16. The SH-2 supports memory widths of 8b, 16b, and also 32b on the FP-144 part. An 8b wide memory would provide wholly inadequate performance: fetching a single instruction (16 bits) would require two memory accesses, and processing performance could not possibly reach the chip's capacity of roughly one instruction per cycle. The 16 bit wide part we chose allows good performance on straight-line code; the SH-2 supports burst DRAM accesses, so consecutive memory locations can often be accessed in one cycle each. However, load/store operations cause significant performance hits since a total of 48 bits (16 bits of instruction plus 32 bits of data) must be transferred in a single cycle. We will combat this in the next revision of the board by using a 32b wide DRAM. Every cycle, the next *two* instructions can be fetched, freeing the SH-2 to begin a data memory access the next cycle if necessary. Loads and stores will still result in a processor stall, but the performance impact is minimized by a larger bus width.

The SH-2 has a very small instruction cache on-die. It has a capacity of 1KB and is direct-mapped (non-associative). The instruction cache can be accessed in a single cycle, drastically improving performance compared to fetching instructions from the slow

DRAM. While cache hit rate statistics for the SH-2 are not available, we can estimate the hit rate based on a similar architecture's hit rate on a similar cache. For a MIPS processor (similar to the SH-2 but with 32bit rather than 16bit instructions), a 1KB direct-mapped cache achieves a hit rate of 96.94% on SpecInt92 code [7]. We reasonably conclude that even the SH-2's small 1KB cache can result in a significant performance improvement. However, our existing software does not enable the SH-2's integrated cache since the cache would interfere with the operation of the debugging software (discussed in Section 3.). Future revisions of the board and debugging software will solve this problem by invalidating the cache when changes to instruction memory occur (i.e., modification of breakpoints).

2.1.3 Persistent Memory

It is essential that a robotics controller board provide some form of persistent storage, so that when it is powered on it can run a basic debugging monitor or a user program. The SH-2 7045 includes 256KB of FLASH memory on-chip, which is more than enough capacity for any reasonable program (we rarely used more than 40KB for our programs). Unfortunately, the FLASH on the SH-2 is rated for "hundreds" of rewrite cycles. This means that after several hundred erase/rewrite cycles that the FLASH may not operate properly, making the board useless. It is necessary, therefore, to minimize the number of rewrite cycles performed on the SH-2. If every iteration of a user's edit-compile-debug cycle caused an erase/rewrite cycle, a user could thus quickly "use up" the FLASH.

Future board revisions will add a high-endurance FLASH chip. This will allow us to write as often as we desire to the FLASH. However, our first two board revisions did not have this additional FLASH chip. We implemented a method for providing persistent storage of user programs in the SH-2's FLASH while minimizing the number of erase/rewrite cycles. This technique, which imposes some inconvenient burdens on the user, is described in Section 3.2. An additional FLASH chip will avoid these inconveniences.

2.2 Interfaces and Extensibility

Computational performance is only one aspect of the Otto controller board. The ability to interface with motors, sensors, and personal computers for programming and debugging purposes is critically important. The board and its peripherals must also be provided with electrical power, which presents certain problems and design trade-offs. In addition, we have provided several ways of extending the capability of the Otto board through expansion connectors and reconfigurable hardware.

2.2.1 Motor Control

The SH-2 has extensive on-chip timer resources that can be used to generate Pulse-Width-Modulation (PWM) signals. PWM is useful for controlling motors with finer resolution than simply "on" or "off". Imagine a motor controlled by the following signal, where a high value indicates "on" and a low value indicates "off":

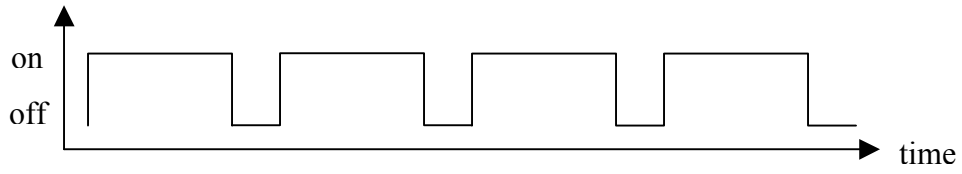


Figure 2. PWM waveform with large duty cycle.

A motor driven with the above waveform is likely to rotate more slowly than if the control signal were left "on", but it will rotate more quickly than a motor driven by:

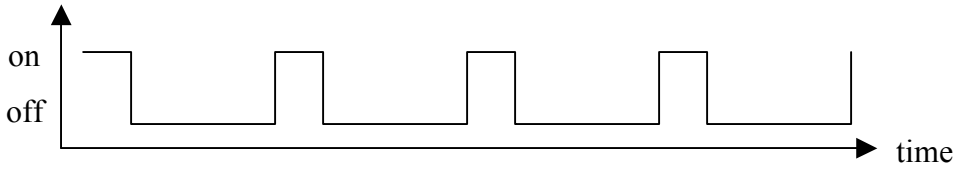


Figure 3. PWM waveform with small duty cycle.

The operation of a PWM generator is specified in terms of two parameters: T_{pulse} and T_{period}. T_{pulse} is the length of time that the PWM signal is high during every interval of T_{period}. The duty cycle of the PWM signal is equal to T_{pulse}/T_{period}.

Due to the non-linear nature of an inductive motor load, the duty cycle is not linearly proportional to rotational speed. However, PWM provides an invaluable method of achieving intermediate output power levels. With PWM, users can regulate the acceleration of their robot, preventing the robot's wheels from slipping due to a large impulse. If motor feedback is available (either through a tachometer or optical encoder), PWM can be used to implement speed control.

The SH-2's PWM generators cannot directly drive high-power motors. We therefore used two dual-channel 2A full H-bridge motor driver chips, providing a total of four independent, high-current, bi-directional motor drivers. Since the motor drivers contain full H-bridges, motors can be driven both forward and backward. However, a PWM waveform only controls speed; to control direction, an additional signal is required. This "direction" signal is implemented with simple digital I/O on the Otto board.

We also provide current-sense feedback circuitry for each channel, so that the amount of current being consumed by the motors can be monitored in software. For motors with quadrature-phase encoders, this is not terribly useful, but for less expensive motors without encoders, current consumption can be a helpful source of feedback. For example, if current consumption suddenly increased for a motor, one might deduce that the wheel had become stuck on an obstacle, causing the motor to stall. The robot could then back up and try to find a route around the obstacle. The feedback circuit is implemented with a precision low-ohm resistor connected between the motor's load and ground; the voltage drop across the resistor is proportional to the current drawn by the motor. After being low-pass filtered to reduce noise, this data is sent to the board's A/D converters.

2.2.2 Servo Control

Servos are motors fitted with integrated control circuitry that allow them to accurately rotate to a specified angle within a certain range of motion (typically 180 degrees). They are often used to build robotic arms or to pan a sensor.

Servos are controlled by periodically sending a pulse whose width, T_{angle}, corresponds to the desired angle. The neutral position, corresponding to the middle of the rotational range, is represented by a pulse of $T_{neutral}$. Pulses of different widths signal different positions, according to the following relation:

$$T_{angle}=T_{neutral}+(angle)*T_p \qquad -90<angle<90$$

Equation 1

$T_{neutral}$ and T_p are parameters provided by the servo manufacturer.

Servos must be repeatedly sent pulses, to "remind" the control circuitry in the servo what position to seek. The refresh frequency required is specified by the manufacturer and can be represented as $1/T_{update}$.

Conveniently, a PWM signal can be used to control a servo motor by setting the PWM T_{period} to the servo motor's T_{update} and the PWM T_{pulse} to T_{angle}. Changing the angle of the servo is just a simple matter of setting the T_{pulse} value. The T_{period} used in servo control is generally about 30ms—much longer than what would be used to control a conventional DC (non-servo) motor (usually less than 1ms), but the SH-2's PWM circuitry can accommodate both.

No motor driver chips are necessary for servo motors; the PWM signal sent to servos to only drives control logic; the power used by servos to change position is drawn directly from the 5V power supply, which already has substantial current sourcing capability (see Section 2.2.3).

In the Otto board's default configuration, up to six servo motors can be controlled in addition to the high-current DC motors. The PWM is implemented using the SH-2's hardware timer facilities. If more servos are required, the PWM capabilities used to control the high-current motors can be reconfigured to control servos instead, for a total of ten servos.

2.2.3 Power Supply

Powering the board's components is a surprisingly difficult problem. Our basic assumption is that users will power the motors and other high-current, high-voltage devices directly from the batteries (i.e., without any voltage regulation). While using a regulated motor voltage can simplify motor control, the regulation circuitry invariably limits the maximum amount of power that can be applied to the motors, reducing motor performance. In addition, it is desirable to have only one battery pack to maintain in a robot, so rather than require a second battery pack to power the board, we provide the option of generating a 5V supply from the same batteries that power the motors. Since motors can cause significant electrical noise, this requires substantial power filtering.

We considered two basic types of power regulators: linear regulators and switching regulators. Each type has significant advantages and disadvantages.

Linear Regulators require an input source of higher voltage than that required on the output. The "excess" voltage is bled away as heat through (conceptually) a variable resistor. The resistance of the variable resistor is dynamically adjusted so that it and the load itself form a resistor divider and the output voltage is of the desired magnitude. The efficiency of a linear regulator drops inversely with increasing input voltage and is equal to:

$$P_{eff} = \frac{V_{out}}{V_{in}}$$

Equation 2

Some robotics users prefer relatively low voltage (typically 6V) motors and batteries. A linear regulator will perform very well with such a low voltage input— an efficiency of 5V/6V=83% is realizable. However a user with 12V motors will likely have a 12V battery, and will therefore suffer 5V/12V=42% efficiency when powering the logic components. This inefficiency not only causes the batteries to drain more quickly than necessary, but also causes a significant heat problem. Our board consumes about 250mA on average which is 5V*250mA=1.25W. The amount of heat being generated by the regulator is (12V-5V)*250mA=1.75W. This amount of power, when radiated by a small TO-220 package, results in a significant amount of heat; junction temperatures of 119°

Celsius, just below the absolute maximum of 125° C, can be expected according to the regulator's datasheet [8].

Switching Regulators generally achieve greater efficiency than linear regulators. A typical step-down switching power supply works by repeatedly energizing a storage element and then discharging it. The rate and duty cycle during which the storage element is energized is adjusted so that the average voltage is the desired value. This output is then filtered so that, rather than a saw-tooth wave output, a relatively continuous value is produced. Greater efficiency is achievable since there is no inherent loss involved; unlike a linear regulator, 100% efficiency is possible with idealized components.

A major problem with switching regulators is the transient response time. A significant current surge can cause a large change in output voltage, and it can take a long period of time for the switching regulator to restore the voltage to the desired level. When the output voltage dips too low, it can cause logic components to erroneously reset. Switchers can also be quite noisy in terms of Electro-Magnetic Interference (EMI). Care must be taken to ensure that the noise does not cause problems with nearby systems.

We have used a number of different power supply systems in our boards, but have not yet found a perfect solution. In our first board revision, to accelerate development time, we used a single linear regulator. However, we had significant heat problems. Our second

revision used a switching power supply, but its transient response time was too slow; when multiple servos (which are powered by the regulated 5V supply) moved at once, a large current spike would occur, causing the supply voltage to drop. When the supply voltage drops below a threshold (4.375V), circuitry on the Otto board intended to ensure proper power-up behavior is triggered, causing the board to reset. We ultimately added a linear power supply in addition to the switching supply to provide power to the servos. This was a stopgap solution—we needed to stop the board from resetting erroneously, and the low-efficiency linear regulator was the easiest way to do so.

Our next board revision's power supply design is still being debated. While linear regulators work well when the battery voltage is low (6V, for example), we want the board to be useful for as large a design-space as possible—we would ideally like to support battery supplies from 6V to 24V. Supporting the higher range of voltages mandates a switching supply since at 24V, efficiency would drop to 21% and heat dissipation would exceed the regulator's thermal maximum even with a reasonable (10° C/W) heat sink.

Three designs are being considered: an improved single switching supply, separate switchers for logic and other high-current devices, and separate battery sources for logic and high-current devices. The last option, allowing for separate battery sources, would completely eliminate the problem of regulating a high voltage source (e.g. 24V) down to 5V; the user would provide a 24V source and a more reasonable source (6V) for logic.

This would require only a linear regulator and would yield good performance, but it requires multiple battery packs that are cumbersome, heavy, and inconvenient.

A single switching supply, if it could be modified to provide adequate transient responses, would be optimal; it would require the least amount of hardware and would still allow a single battery source. If we cannot improve the performance of the switcher, we could provide two independent switchers—one for the logic (which exhibits relatively small current transients) and one for high-current, more volatile devices (which can tolerate brown-outs.) Multiple switchers, however, increase the cost and size of the board considerably.

2.2.4 User Configurable Hardware

A major problem in any general-purpose hardware system is that it cannot possibly provide every feature that every user needs. For example, a user may require four quadrature phase decoders—a reasonable requirement, but one that the SH-2 cannot meet by itself. However, it would require a very large amount of hardware to meet *everyone's* minimum requirements. User configurable hardware allows users to include their own custom-designed hardware. For example, a user requiring a few extra quadrature phase decoders can write a Verilog or VHDL description of one (or download one from the Internet) and install it in the configurable device. Many possible features can be added in this way: additional timers, clock generators, quadrature phase decoders, PWM generators, interfaces to other chips (like an LCD module), simple FIR filters, etc. The

configurable hardware device communicates to the SH-2 via memory-mapped I/O. The CPLD or FPGA appears as an SRAM to the SH-2.

Our first two board revisions used an Altera MAX 7000S series CPLD. This choice was prompted by availability and the eagerness of Altera to provide us with samples. The 7000S maintains its configuration internally, saving board space and making design easier. However, the 7000S series has several problems making it imperfect for our application. The 7000S uses a large NOR plane which consumes power even when idle, reducing battery life. Further, in our experience, our VHDL and Verilog designs were usually flip-flop limited; our logic was usually quite simple, but large numbers of macrocells would be required just to form the registers. An FPGA would most likely provide a better match of logic and storage capacity, especially since at 30MHz, extremely fast pin-to-pin performance (one area where CPLDs perform better than FPGAs) is not a significant issue.

We used the configurable hardware device to implement an interface to an LCD module. Character mode LCD screens in sizes such as 16x2, 20x2, and 20x4 are readily available with a simple controller built in. The most common controller, an HD44780, operates at a maximum of 2 MHz [9]. The SH-2's 30MHz bus is far too fast to interface to the LCD's controller directly. We implemented a simple controller in the Altera 7000S that allowed us to perform bus transactions to the LCD in two phases: a setup and a completion. The setup phase would configure all of the appropriate pins to begin a read or write request,

and the completion would end the transaction. By splitting the phases, they could be separated by many CPU cycles thus giving the LCD controller enough time to respond. In addition, the SH-2 could perform unrelated operations between the two phases, enhancing performance.

In our next revision, we wish the configurable hardware chip to be optional since it can be quite costly. An LCD module is a popular feature among users, so making the LCD functionality dependent on the configurable hardware chip is less than ideal. If possible, our future revisions will control the LCD module directly from the SH-2 by attaching it to digital I/O pins. The interface to the LCD module will be implemented in software by bit-banging the digital I/O port.

2.2.5 Analog I/O

The SH-2 provides a mid-speed analog-to-digital converter able to perform conversions every 9.3uS and an integrated 8-to-1 analog mux. Based on our experience with other controller boards, we determined that eight analog sensors would be insufficient for a relatively large number of robots. Therefore, we fitted the controller board with an additional, external 16-to-1 analog mux. As a result, there are a total of 23 analog inputs on our controller board, as depicted in Figure 4.

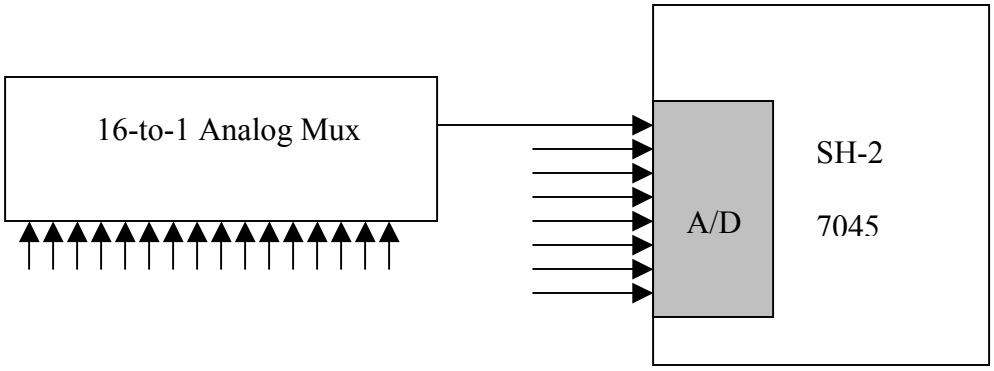


Figure 4. Analog input module block diagram.

The muxing scheme shown above provides two different classes of analog ports. Those connected directly to the SH-2 are sampled more quickly, while those connected to the 16-to-1 analog mux are sampled considerably less frequently. For most sensors, like the popular Sharp GP2D12 infrared range finder, update frequency is a minimal concern. The GP2D12, and most sensors like it, can provide a new value only every 30ms or so. At 9.3uS per sample, the SH-2 could conceivably sample all 23 analog ports every 0.21ms. Clearly, the SH-2's sample rate is not a limiting factor for these types of sensors. We have provided software that performs continuous updates of all 23 sensor ports; when a user needs the value of a given analog input, the most recently measured value is instantly returned—the user does not have to wait for the port to be sampled again. With our current scheme, we continuously sample each of the 8 "fast" ports (one of which corresponds to the output of the 16-to-1 mux), and each time we make a loop, we increment the select input on the 16-to-1 mux. The sampling performance is summarized in Table 4.

Analog Input Class	Number of Ports	Update Period	Update Frequency
Fast	7	74.4uS	13.4kHz
Slow	16	1190.4uS	840 Hz

Table 4. Analog port sampling performance.

However, there are sensors for which the SH-2's sampling performance is not adequate. An example is the Mitsubishi M64282FP CMOS imaging sensor. The imager is capable of producing 128x128 grayscale images at a maximum rate of 30 frames per second. This would require a sampling rate of nearly 500kHz, or one sample every 2.0uS. Even with the SH-2's A/D converter devoted to sampling only one input, it can achieve only 107.5kHz, or one sample every 9.3uS. This corresponds to a rate of 6.5 frames per second. While the M64282FP can provide data at this slower rate, it would clearly be desirable to increase the data rate. However, to keep the board layout as simple as possible, we do not intend to add an additional high-speed A/D converter; this function could always be provided through the Otto's expansion connector (see Section 2.2.8).

The SH-2 does not have a digital-to-analog converter built-in, and we decided not to include one externally due to their limited usefulness. One possible application for a D/A converter is in driving a speaker; much higher audio quality is possible by using a D/A converter. However, simple audio feedback can be implemented with simple digital I/O, and this is adequate for most purposes.

2.2.6 Digital I/O

Digital I/O can be used to interface to switches and other logic devices. The SH-2 has very few pins available for use as digital I/O. The configurable hardware device, however, has a very large number of available pins that can be used for digital I/O. These pins can be accessed from the SH-2 via memory-mapped I/O—a read or write to particular memory addresses is actually intercepted and handled by the configurable hardware. Whether a particular pin corresponds to an input or an output can also be configured by performing the appropriate memory access.

In practice, digital I/O is far less useful than analog I/O for interfacing with sensors. A typical robot uses only a handful of digital switches. Digital I/O, however, can be extremely useful for interfacing to external logic devices unable to interface directly to the SH-2's bus. The I/O pins can implement a simple proprietary interface instead, simplifying the design of the external devices.

2.2.7 Serial Port

The SH-2 supports two independent serial channels. Each can be run at up to 115,200 bits per second. However, the SH-2 cannot produce the required high-voltage signals required to meet the RS-232 specification. A Maxim 203 RS-232 transceiver charge-pump is therefore provided on the board. The Maxim 203 is a highly integrated charge-pump that requires no external charge-pump capacitors, which helps reduce the size of the board.

2.2.8 Expansion Connector

As previously stated, it is impossible to make a board that provides every feature for every user, so we provided an expansion connector that allows the addition of a complex component to the board. The expansion connector includes all of the system bus pins, including address pins, data pins, control pins, as well as IRQ lines. Almost any type of device could be added through this connector, from an Ethernet interface to a coprocessor.

On our next revision board, we plan to add an I^2C two-wire interface. I^2C can achieve 3.4Mbps data transfer rates, and up to 128 devices can be added through daisy-chaining. More and more components are becoming available which support I^2C, including LCD displays, motor controllers, and even CPUs. In addition to allowing new devices to be easily attached to the Otto controller board, the I^2C interface conserves board space since it requires only a single two-wire connector on the board.

2.3 Other Design Issues

The Otto board's design was affected by a number of additional concerns: safety, manufacturability, and feature parity with existing platforms. Safety involves protecting the user from the board, as well as protecting the board from the user. We also had to choose between several different manufacturing options—for example, whether to use a

two-layer or four-layer printed circuit board. Lastly, ensuring that the Otto provided compelling advantages over other platforms influenced our design.

2.3.1 Safety

The Otto board's safety features were a consideration throughout its design. The risk of electrical shock to humans is very low. Only voltages greater than 35V DC are considered to be hazardous [10], and all of the board's operating levels are significantly lower.

The larger concern for safety was protecting the circuitry from human error. Hobbyists and amateurs are likely to make errors in wiring sensors, motors, and other devices that connect to the Otto board. We attempted to minimize the risk of damage to the board caused by simple user error in several ways.

The power supply is the first line of defense against user error. Almost any fault condition that could damage the board involves the power supply. The most obvious example, as well as the most likely fault condition, is a short circuit. The regulators we use provide two protection mechanisms: thermal and current limiting. If the current sourced by the regulator exceeds a preset threshold, the regulator drastically reduces the amount of current provided until the fault clears. The regulator will also shut down if it becomes too hot, which would most likely be due to an unusually large load (but not a short circuit) being applied for a long time. We have also considered adding a

conventional fuse, but have so far not seen a need. The same issues exist for the motor drivers; should the motor connectors be inadvertently shorted together, the resulting large current draw will cause the motor driver chips to safely shut down. We have not extensively tested either mechanisms due to the limited number of boards we have produced thus far, but we intend to do so for future revisions of the board.

Electrostatic discharge (ESD) is an additional threat to the board. While simply being soldered to a larger circuit board affords some protection to electrical components, those parts most likely to encounter ESD, like the RS-232 charge pumps, have manufacturer-added ESD protection. However, unless the Otto is put into a protective case, it is impossible to eliminate the risk and a reasonable amount of care must always be exercised.

In order to reduce the risk of shorts throughout the board, we have attempted to minimize the amount of exposed conductor. For example, much of the header on the board is connected directly to the power or ground planes. Using female header is an easy way of making these pins better protected against stray wires. We have chosen to use HandyBoard-style connectors (pseudo-polarized one-signal with power and ground—see Figure 5) to ease users' transitions to our board. With these connectors, no damage will occur if the connector is inserted backwards. We also added mounting holes for feet; should the board be set down on a conductive surface, the traces on the bottom side of the board will not be shorted.

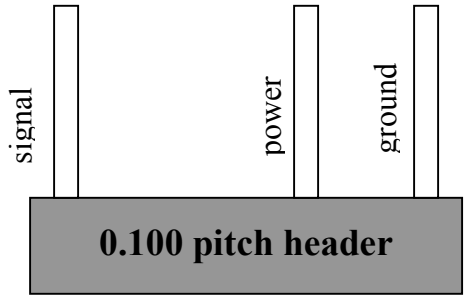


Figure 5. Sensor connector providing power and ground.

2.3.2 Schematic Capture, Layout, Manufacturing and Assembly

We used the PADS schematic and layout tools PowerLayout and PowerPCB. Schematic entry was relatively straightforward. The complete schematics are included in Appendix A. Layout, however, was a time-consuming task. We did not have experience laying out relatively complex boards like the Otto, and we underestimated the amount of time required to do it.

Our board measures roughly 4.5” by 4” and has four layers. The outside layers are used for routing (we made extensive use of the BlazeRouter autorouter) while the inner layers were reserved for power and ground. In addition to simplifying routing, dedicated signal planes for power and ground improve signal characteristics by providing an optimal return path for current. We manually increased the thickness of traces that would be carrying large amounts of current.

It is virtually impossible for an amateur to manufacture his own four layer board, which was initially disappointing to us; we had hoped that ambitious amateurs would be able to

make their own Otto boards. However, we realized that even if we had produced a two layer layout, that most amateurs would not be able to solder the high-density components. Having already committed to a path requiring professional manufacture, we chose surface mount components for many of the parts, even when through-hole parts were available, since it allowed the overall board to be smaller.

Our initial design began in 1998 with selection of a microcontroller. The pace increased in the spring of 1999 when we began doing actual schematic capture and layout. We started our first manufacturing run in the fall. After finding and fixing several bugs, we used the board in a small robot the following spring. Our second revision's primary goal was to fix the bugs in the first revision, not to add additional features. This board was manufactured in the fall of 2000 and was used in a robotics class we taught at MIT during the Independent Activities Period (IAP) the following January (see Section 4).

It is difficult to accurately estimate the cost of the board, but our estimate is around $200-$300 each in runs of around 10-20 boards. It is difficult to estimate more precisely since corporate sponsors donated many of our components and the manufacturing and assembly were partially sponsored. In addition, buying and manufacturing in larger quantities has a dramatic impact on the cost. Our estimate is based on a per-unit cost of about $160 per board for manufacturing and assembly, with the remainder going towards purchase of board components. We expect dramatically lower costs when Otto is produced in greater quantities. The second revision board is shown in Figure 6.

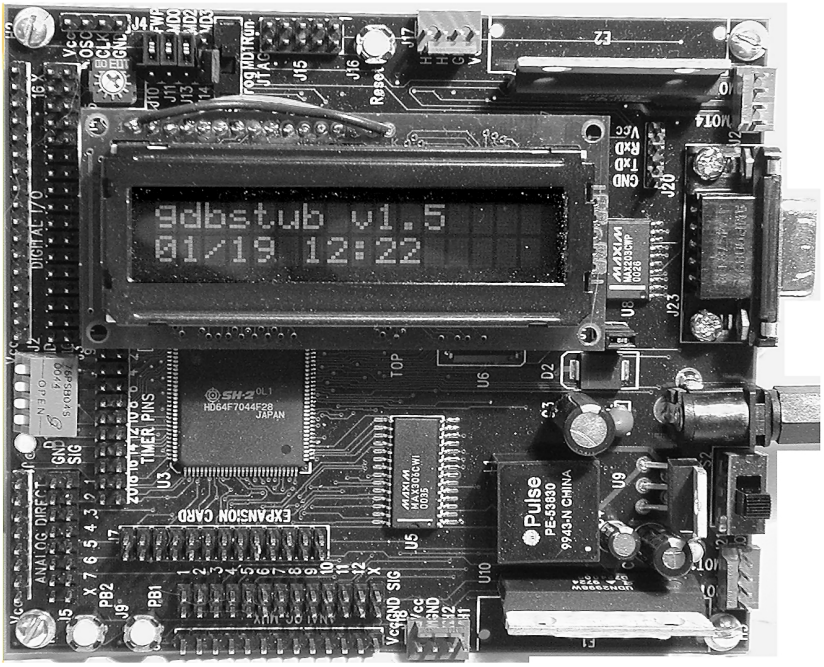


Figure 6. Photograph of the Otto controller board.

2.3.3 Hardware Comparison

Our board represents a significant improvement over other currently available boards. Table 5 summarizes the features of the Otto board and several other commonly available boards (previously described in section 1.1).

	Cricket	Lego RCX	HandyBoard	Skiff	Otto
CPU	Microchip PIC16F84 (1 MHz)	Hitachi H8/3292 (16 MHz)	Motorola 68HC11E9 (2 MHz)	Intel SA110 (200MHz)	Hitachi 7045 (28.8 MHz)
RAM	68 bytes	512 bytes	32 KB	Up to 16MB	Up to 16MB
Sensor Capability	Poor	Okay	Excellent	Excellent	Excellent
Motor Driving	Okay	Okay	Excellent	Excellent	Excellent
Servo Driving	Poor	Poor	Good	Excellent	Excellent
Extensibility	Okay	None	Excellent	Good	Excellent
Cost	Very Low	Moderate	Moderate	High	Moderate

Table 5. Comparison of existing low-cost robotics platforms and the Otto board.

3. Software Development Environment

The Otto board can be programmed with the GNU toolchain. During selection of a microcontroller, we discovered that there is an existing port of the GNU toolchain to the Hitachi SH-2, including a linker, assembler, compiler, and debugger. Using the GNU toolchain allows users to employ a mature and free development environment, which is a powerful advantage for the Otto platform. We also used the Cygnus newlib library, which provides implementations of libc functions (such as printf and malloc). We were able to use most of the toolchain components without modification.

3.1 Development of the gdbstub

The first real application we developed was a gdbstub, which allows code to be downloaded into the SH-2 and interactively debugged using the GNU gdb debugger. When we began work on the Otto board, there was a project underway to develop a gdbstub for SH-2 based boards [11], but it was in an extremely primitive state. It took a considerable amount of time to get the gdbstub to work on our board due to differences between our board and the Hitachi SH-2 reference board for which the gdbstub was originally written. In addition, we found and corrected several critical bugs. These corrections have since been incorporated back into the generic SH-2 gdbstub source tree.

With the stub done, it was possible to connect to the board from a host PC and examine and modify the contents of memory, as well as configure, test, and characterize the on-

board peripherals. For example, it was possible to manually configure the timer channels for PWM operation and check the output pins to ensure correct operation. With a working gdbstub, developing user applications became relatively easy.

GDB is the standard way of interacting with the Otto board. Code can be downloaded, debugged, variables inspected and modified, and much more. While it is certainly possible to completely remove the gdbstub and install user programs, there is little reason to do so. The gdbstub can load and run user code with or without debugging information, and when no breakpoints are set, there is no performance impact at all.

The gdbstub implements breakpoints by actually modifying the instruction stream, overwriting some instructions with TRAPs. This requires code to be in RAM in order for it to be debugged—breakpoints set for code located in FLASH are never triggered since the instructions cannot be replaced. User code can also send debugging messages via the gdbstub for display on the host PC's terminal.

3.2. Memory Map and Code Location

Before any code could be assembled, compiled, and linked into a runnable form, a linker script had to be written. The linker script tells the linker where different data and code objects will be located in memory.

Our board differed from other SH-2 boards in its memory layout; for example, we used DRAM rather than SRAM. The memory map is shown in Table 6. A significant difference between typical linker scripts and those found in embedded system is that code and data are often stored in one location but run in another. Suppose that we wish to put a C program into FLASH (so it will be run when the board is powered on), and it declares an initialized global variable:

```
int g_initialCount=237;
```

This causes four bytes of data to be allocated and initialized to the value 237. This data must go in FLASH, because that is the only non-volatile memory available; if it were stored in DRAM (or other volatile memory), the initial value of 237 could be overwritten and lost. The variable cannot be stored in FLASH either, since the FLASH is read-only (except under very special "programming" conditions.) The solution is to assign g_initialCount an address in DRAM or SRAM, but to store the initial value in FLASH. Then, a special startup routine copies static initializer data from FLASH to its actual DRAM/SRAM location. This sort of initialization is performed by the C RunTime initialization code (crt0). We needed to modify the standard crt0 program so that it could also handle basic hardware initialization, such as initializing the DRAM controller.

Address Range	**Size**	**Memory**	**Purpose**
00000000-0003ffff	256K	FLASH	Persistent code storage; libraries
00200000-003fffff	2MB	CPLD/FPGA	Communicating with configurable hardware.
01000000-01ffffff	16MB	DRAM	Heap, Stack, User programs
fffff000-ffffffff	4KB	SRAM	Reserved for instruction cache and runtime software

Table 6. Memory map of the Otto board

We tried to balance two contradicting desires: to provide persistent storage of the user application (so that the user program will not be lost when the board is powered off) and to increase the lifetime of the SH-2 by minimizing the number of erase/write cycles to the SH-2's on-chip FLASH. The SH-2's on-chip FLASH is officially rated for only hundreds of erase/write cycles, after which the FLASH may no longer work properly. As users develop code, they are likely to rapidly make revisions to their code and wish to download the code to the board and run it right away. Asking users to try to reduce the number of attempts in order to reduce the FLASH erase/write cycles is simply not an option.

Our solution to this problem relies on the observation that when users are rapidly evolving their code, they do not need persistent storage of the program; most likely, the program will be run only once, and will be run right away. Therefore, we provided two linker scripts, one that targets the DRAM, and another that targets the FLASH. Only when users are satisfied with their code do they actually commit it to FLASH—otherwise, they simply download the code into DRAM. This dramatically reduces the number of erase/write cycles required of the SH-2's FLASH. Code is downloaded into DRAM through the gdbstub.

Alternately using FLASH and DRAM for code does introduce problems. For example, the DRAM has a slower access time than the FLASH; code running out of FLASH

therefore executes somewhat faster. When users practice good programming techniques (for example using the kernel's sleep() function rather than a "for" loop to delay a given length of time), this is not a significant problem.

Our next board revision will attempt to solve this problem by adding an additional high-endurance FLASH chip. Inexpensive chips (less than $10) with endurances of a million erase/write cycles and high capacities (512KB or more) are available. If a user makes two revisions per minute, he could work continuously for about a year before a failure could be expected. This additional FLASH chip will allow users to always benefit from persistent storage, without concern of wearing out the SH-2. In addition, with an external FLASH chip it is not necessary to delete the contents of the SH-2's FLASH to permanently store a program, allowing the gdbstub (and/or other utilities) to always remain accessible. The large capacity of an additional FLASH chip makes it possible to store both user programs and configuration data for a FPGA. In the future, we will implement a very simple file system to track different types of data stored in the FLASH.

Code cannot be run directly out of the off-chip FLASH since the SH-2 gdbstub implements breakpoints by modifying the instruction memory— overwriting user instructions with TRAP instructions. To overcome this, we plan on copying the user program from the external FLASH into DRAM when the board starts up, then run the code from DRAM. This provides a consistent runtime mode—program execution speed

will be constant, only one linker script is required, and the gdbstub (and other data in the on-chip FLASH) is always available.

In addition to program instructions, the DRAM is used for the stack and heap. The SH-2 does not have an MMU (Memory Management Unit, the governor of virtual memory), so a dynamically growing stack is a difficult feature to achieve. With an MMU, the stack for each process can grow until it hits that processes' heap; little memory is wasted since the empty regions in each processes' memory space simply aren't mapped to real DRAM. In an MMU-less processor like the SH-2, each stack must exist within the same address space, and every gap represents wasted DRAM. There is no way to locate a dynamically growing heap and multiple stacks in a way that doesn't waste real DRAM. Therefore, we accepted the standard solution in MMU-less machines that the stack for each process must be preallocated (usually *from* the heap) and therefore be fixed in size. In practice, fixed stack sizes do not cause any significant problems, but users must be aware of the limitation so that they do not try to allocate large variables on the stack or write programs with large amounts of recursion.

Ideally, the heap would begin where the user code and data ended. This is easy to achieve; malloc determines where the heap begins based on the value of the symbol _end, which is defined by the linker script to be the highest memory address used by the user's code or data. If the user program, with its appropriate definition of _end, is linked against

malloc, the heap will indeed begin immediately at the end of the user's program resulting in absolutely no memory waste.

Ensuring that malloc knows the correct value of _end is much more complicated when malloc and the user application are compiled at different times. In an attempt to reduce the amount of time required to download user programs, we put common functions (malloc, printf, cos, log, etc) into the SH-2's FLASH. User code is linked against the SH-2's FLASH, so that when these common functions are used, a separate copy does not need to be downloaded. However, malloc is compiled and written to FLASH long before the user program has been written; it cannot know the correct value of _end. Not only is the correct value of _end a problem, but the libraries in the SH-2 FLASH often require some static data storage which must be located somewhere in DRAM; this space must also be allocated *before* the user code has been written.

Our work-around has been to simply divide the DRAM space into two pieces; a segment for the user program, and a segment for the heap. We reserved the first 256KB of DRAM for user code (our typical programs and global variables never exceeded about 64KB). At the beginning of DRAM+256KB, we can put static variables for libraries residing in FLASH, and after that, _end denotes the beginning of the heap. Using a fixed size of 256KB results in memory waste for user code less than 256KB in size, and simply doesn't work when user code is larger than 256KB. One solution would be to resolve symbols (including _end) on the board at runtime, which requires a dynamic loader to be

ported to the Otto board. The SH-2 FLASH would then contain an actual ELF or COFF object file, rather than ready-to-run (already linked) code. Realistically, however, the small amount of wasted DRAM will not be missed.

3.3 Provided Software Libraries

To assist users in developing programs, we made several software libraries available. Standard functions are implemented by Cygnus' newlib, an open-source libc clone including implementations of 'printf', 'strcpy', 'malloc', as well as mathematical functions such as 'sin', and 'exp'. We have written libraries for dealing with common peripherals such as character-based LCD displays and IR transceivers. In addition, libraries for interfacing to the Otto board's built-in peripherals (such as PWM generators and A/D converters) are provided.

The LCD driver is particularly useful. Users can purchase any one of a number of pin-compatible LCD modules, including 2 line, 16 character displays, and 4 line, 20 character displays. The LCD module provides the most convenient way of displaying debugging information, since the LCD module is always available, even when the board is disconnected from the host PC.

3.4 Kernel

One feature that is essential for a robotics board is the ability to multitask—to run multiple threads. Multi-threaded programming greatly eases the design of programs that must be able to react quickly to external stimulus while carrying out computationally intensive tasks. We had originally considered using an off-the-shelf operating system to provide this feature, but eventually decided against it. Many of them require a very large amount of persistent storage; fitting a Linux port in the few hundred KB of available storage, for example, is virtually impossible. Furthermore, few of them specifically support the SH-2; porting them to the SH-2 would be a significant undertaking. Lastly, most provide far more features than are necessary or appropriate; for example, there is little to no need for file systems or pipes. On the other hand, by not using a standard operating system, we lose the ability to leverage a potentially large number of device drivers such as Ethernet devices. This is not a major issue for a small board such as the Otto; there is no PCI or ISA bus on the Otto, making most device drivers inapplicable.

We decided that it would be best to write a simple multithreading library, rather than port an entire operating system to the board. Our kernel is very simple and lightweight, making it potentially useful for many embedded operating systems. It provides multithreading as well as basic synchronization primitives.

There are several special considerations for an embedded system's thread scheduler. A very small scheduling quantum can be extremely useful; when communicating with polled peripherals, a small quantum reduces the communication latency by polling more

frequently. In many cases, the peripherals can be operated in an interrupt-driven mode, which provides the best possible performance (lowest latency, lowest overhead), but we expect that many users will be uncomfortable writing their own interrupt service routines (ISRs). Having a small quantum means that the scheduler will be invoked more frequently; it is important to make the scheduler relatively simple so that CPU time is not wasted needlessly. Multiple thread priorities are also quite helpful. For example, a thread designed to detect an imminent collision should run at a higher priority than one searching for a optimal path between two points.

The scheduling algorithm is a typical multilevel priority scheme designed to minimize the risk of starvation by ensuring that even low-priority threads get run periodically [12]. Each thread is associated with a "base priority" and a "current priority". The programmer specifies the base priority; the higher the number, the more frequently the thread will win the scheduling election. Thread election is actually performed on the current priority, which is initially set to the base priority. The current priority is incremented each time a thread is *not* selected for execution, but when a thread is elected, that thread's current priority is reset to the base priority. In this way, the longer a thread waits for a chance to run, the higher its current priority becomes. This scheduling algorithm ensures that every thread will periodically run (eliminating risks of scheduler-induced starvation), and that higher-priority threads will run more frequently than lower-priority threads.

4. Mobile Autonomous Systems Laboratory (MASLab)

The Otto board was used in a robotics course at MIT during the one-month long independent activities period (IAP) in 2001. This course was called "Mobile Autonomous Systems Laboratory", or MASLab; MIT subsequently assigned the course number 6.186. Students built robots using provided parts and programmed them to perform a series of tasks in a maze-like world. MASLab offered students six units of pass/fail credit and six engineering design points (EDPs), a modest but effective incentive to students.

4.1 Motivation

MASLab was conceived to give MIT undergraduate and graduate students an opportunity to solve challenging robotics problems by building and programming robots. The Otto platform was a key ingredient, providing ample computational power to implement complex behaviors.

Since IAP 2001 was MASLab's first year, we limited enrollment to improve our ability to deal with unexpected problems. A total of nine students enrolled, ranging from freshmen to seniors, most with little robotics experience. These students formed three teams, each team with their own robot.

4.2 The Challenge

The MASLab challenge was to build and program a robot to find and retrieve "target" objects, and drag or push the targets back to the position where the robot was activated. The playing field in which the robots operate was surrounded by walls and contained a number of obstacles (see Figure 7). The shape, size, and configuration of the walls and obstacles were unspecified; they changed dramatically from round to round. The surface of the playing field was a flat and smooth tile floor. Obstacles and walls had vertical walls with a fixed height and were made out of a highly infrared (IR) reflective surface.

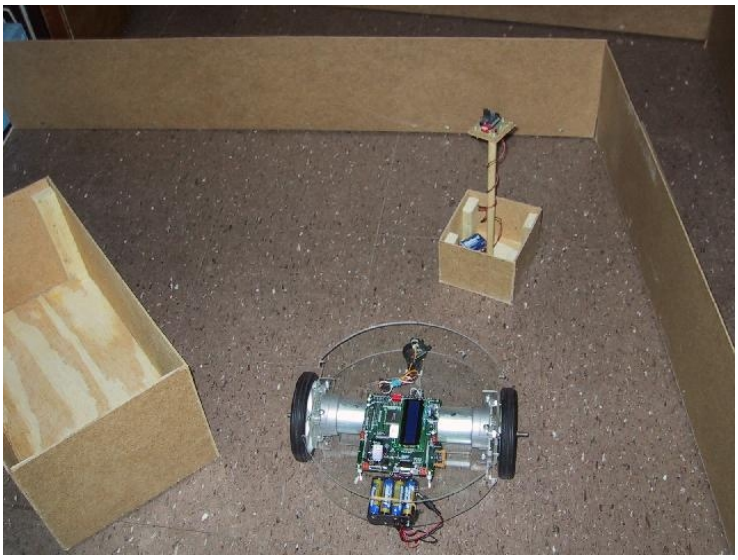


Figure 7. A robot in a playing field.

Target objects were small boxes of a specified and fixed size, with a vertical dowel designed to be easily grasped. Each target was fitted with an IR beacon, mounted higher than walls and obstacles, to make it easier for robots to find them. Stationary IR beacons

were also positioned around the periphery of the playing field; robots could use them as navigational markers. Each IR beacon transmitted a unique code, making it possible to identify each beacon.

The MASLab challenge is significantly more difficult than most other university robotics challenges such as the Trinity Fire-Fighting contest[13], or the contests used in MIT's 6.270 [14]. The most notable difference is that in the MASLab contest, robots do not know how the playing field is configured. In addition, the requirement that robots move a target back to the robot's starting position requires the robots to be able to navigate relatively precisely. We believe that the similarity of the MASLab challenge to the problem of navigating a real-world environment like an office makes the class more compelling for students. We also believe that the greater difficulty inspires more creative work.

We designed an IR beacon and receiver system, which made it possible to determine the direction to other beacons from up to three meters away. A single transceiver module contained both the beacon and the receiver plus a 4bit DIP switch that determined the identification code transmitted by the beacon (see Figure 8). The transmitters were formed by six IR LEDs, broadcasting in every direction. The module's receiver provided directional reception; when the receiver was panned horizontally across the playing field, the directions to other beacons could be determined. All of the teams constructed an optical baffle out of paper and matte black paint. The baffle made the receiver more

directionally selective. The beacon/receiver module was implemented on a 1.5”x1.5” two layer PCB and was controlled by a PIC16F84. The module cost under $15 each with a manufacturing run of 24 boards. It interfaced to the Otto board through digital I/O. We will provide additional information on the modules by request.

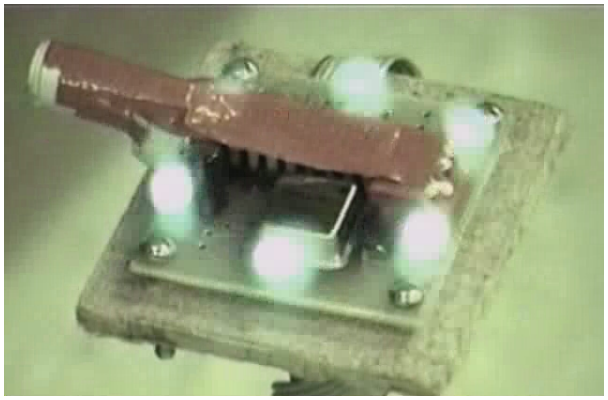


Figure 8. Infrared transceiver with a baffle, viewed in the infrared.

While we allowed students to use any material in the construction of their robots, most used the plexiglass we provided. Plexiglass proved to be a useful material; it is easily cut and shaped with hand tools and is quite strong. We purchased DC motors with integrated gearheads and optical encoders from a surplus warehouse for $25 each, with each team receiving two. With two independent motors available to them, all three teams opted for a differential drive train. IR rangefinder sensors, as well as an IR beacon/receiver, were also provided to the teams. The IR rangefinder sensors have poor resolution and limited range (8cm to 30cm) but were still useful for both building a map of the environment and

obstacle avoidance. In future years, we hope to provide ultrasound range finders that provide substantially better resolution and range (several meters).

4.3 Class Timeline

The first week of IAP was devoted to instruction and parts distribution. Only one student had done robotics works previously, so we presented several lectures on basic robotics and AI concepts. We also presented information on the Otto platform, including documentation for the controller board as well as the runtime utility APIs.

Students spent the second week physically constructing their robot and characterizing the motors and sensors we provided. We were surprised by the length of time taken by the teams to construct the robots; we had produced a simple but functional chassis in an afternoon. By the end of the second week, two teams were still working on their robot's bodies. Our best guess is that students underestimated how much time would be required to finish their robots, and therefore did not push themselves to make rapid progress early on. In future years, we hope to address this by adding more checkpoints during the class. An early checkpoint will likely include a functional chassis during the first week.

During the third week, students worked feverishly to implement basic robot capabilities, such as "turn *n* degrees", "go forward *n* units", or track the robot's current position using odometry. This was the first exposure to control systems for many students, and many struggled with it. The team composed of the most experienced students began

implementing behaviors (such as “stalking” an IR beacon) in addition to low-level control functions.

Students put in long hours during the fourth week, implementing the strategies they thought would work the best in the challenge. The most experienced team made steady but slow progress throughout the week. The team with the least experience confidently programmed an elaborate AI algorithm over the course of several days only to discover that the non-idealities of the real world made their work nearly useless. As instructors, we debated whether to alert teams when they we saw them working on something that we thought were impractical or whether to let them discover their errors on their own. While we generally saw the value of learning from mistakes, we certainly would have cautioned against coding an elaborate AI algorithm without incremental testing. Unfortunately, that team had largely disappeared from lab and we had difficulty getting status updates from the team members. While we can’t dictate that students spend all their time in lab, we hope that we can better express the importance of staying in communication with instructors in the future. We hope to assign teams to teaching assistants, whose job will be to prevent such occurrences.

4.4 Exhibition

An exhibition was held at the end of the fourth week, to an audience of MIT affiliates and friends of the students. We chose an exhibition environment, rather than a competition, to

foster an atmosphere of inter-team cooperation, but also because the small number of teams that participated would not allow a very interesting tournament.

All three teams presented fully constructed robots, but all wished they'd had more time for programming. One team's robot could successfully find, capture, and drag back a target while avoiding obstacles fairly consistently. Their robot iteratively stopped, scanned their surroundings, and by assigning "points" to various sensor results, used a hill climbing strategy to determine their course (their robot is shown in Figure 9). The second team was able to accurately triangulate the positions of targets by measuring the angle to the target from two different locations about a foot apart. They measured the length of the baseline using odometry and were able to use odometry to drive to the target, but did not have time to incorporate obstacle avoidance into their routines. The least experienced team suffered an initial setback by not incrementally testing their software, but by the end of the contest, they could iteratively stop, scan, and move in the direction of the target, using odometry to return to their starting point in a straight line.

We were pleased to see that all of the teams were able to build relatively sophisticated robots capable of using odometry and IR modules to navigate a very difficult playing field. Two of the teams even used velocity feedback control systems (PD) for their drive train.

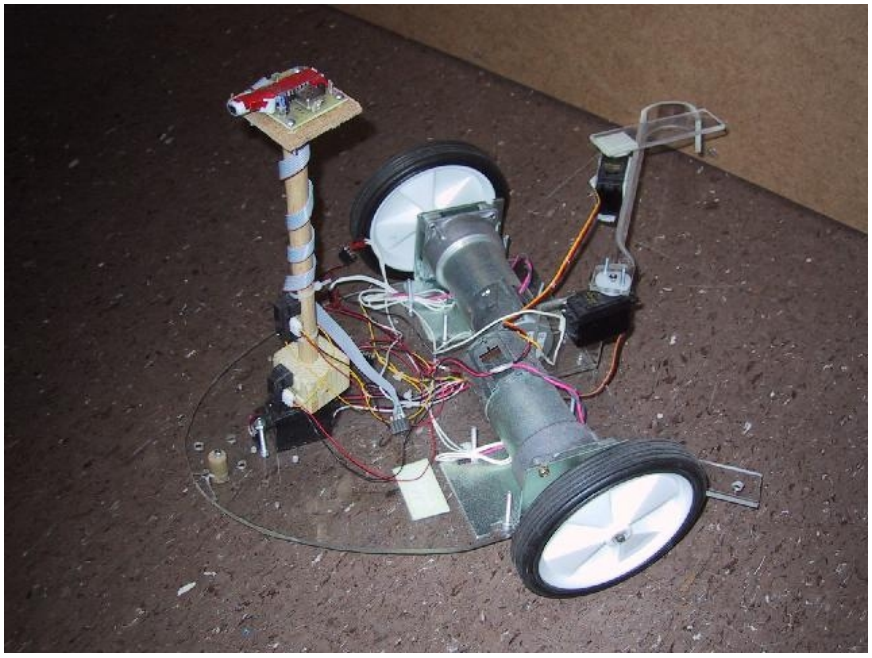


Figure 9. A student-made robot.

4.5 Results, Future Plans and Conclusions about MASLab

We plan on teaching MASLab in the future, and to continue using the Otto board. The Otto board's quadrature phase decoders and high-power motor drivers enabled sophisticated motor control, a capability used by all three teams. The students provided valuable feedback about the Otto board that has influenced plans for future revisions. In addition, the students helped improve the libraries we provided by offering suggestions and reporting counterintuitive or buggy behavior.

5. Conclusion

We have developed a general-purpose controller board that provides higher performance than similarly priced boards. A rich set of runtime utilities, including a preemptive, multitasking scheduler has also been provided. The GNU toolchain can be used to program and debug the board.

Through the course of two manufacturing runs, our design has been tested and improved. The software used on our board, an important element of any robotics platform, was continually improved during this time. After our second manufacturing run, we used our board in a robotics contest at MIT. During the Independent Activities Period (IAP) in January 2001, nine adventurous students with an interest in robotics built autonomous robots around our controller board. This litmus test helped validate our board's design and establish its usefulness in the context of robot building.

We have since presented our platform to several groups with positive feedback. Shortly after teaching the robotics course, we presented our design to 6.270—MIT's well-known robotics course—in the hopes that they would adopt our design in future years. They enthusiastically embraced our platform and have pledged to use it in future years, as well as assist in future revisions. We also presented to the AAAI Symposium on Robotics and Education in March 2001 [15], to an audience of researchers and educators, who expressed interest in our platform.

While the current incarnation of the Otto board is a capable platform, we will continue to improve it. Our work has been placed in the public domain to encourage others to contribute. Adding additional features, improving robustness, making the board more economical to produce and easier to use are high priorities. We believe the Otto platform will become a popular and powerful asset to educators and researchers working with robotics.

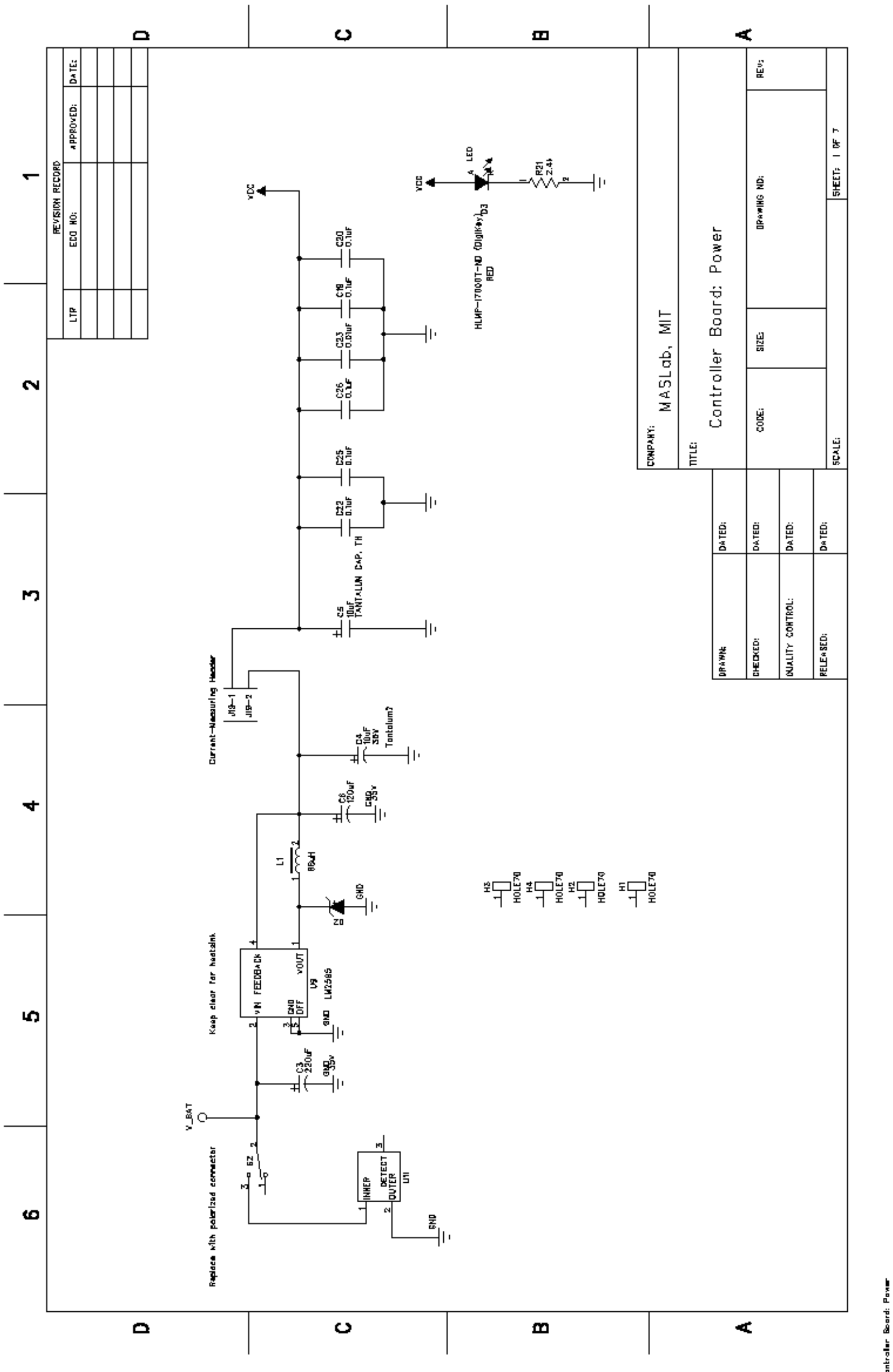
Appendix A. Schematics

We used the PADS PowerLogic and PowerPCB tools to develop the schematics and PCB layout. These tools are expensive commercial products, but were donated to us directly from PADS.

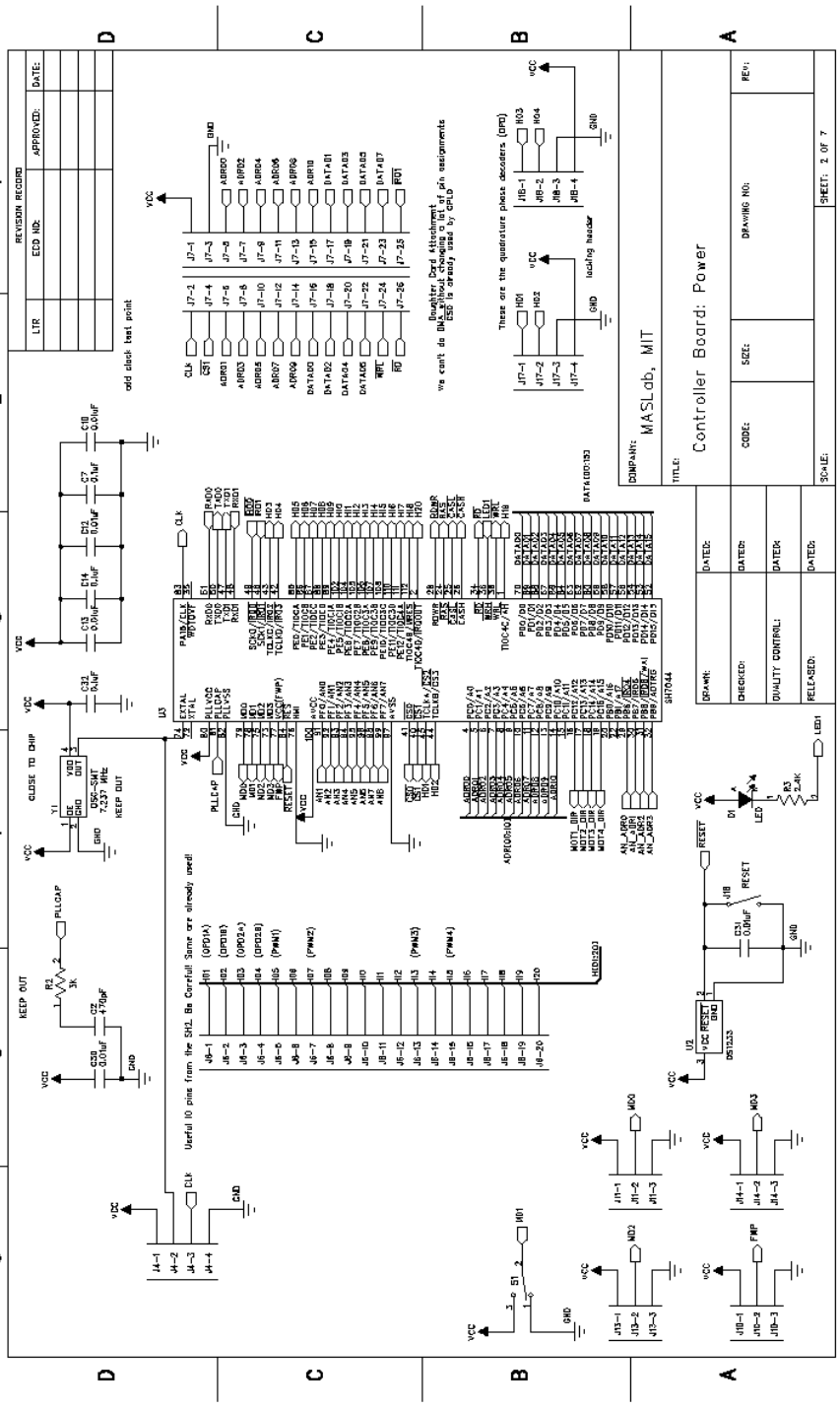
Schematics were divided into several pages for conceptual clarity. The schematics included here are those from our second manufacturing run. The second run had no connectivity errors but several design issues:

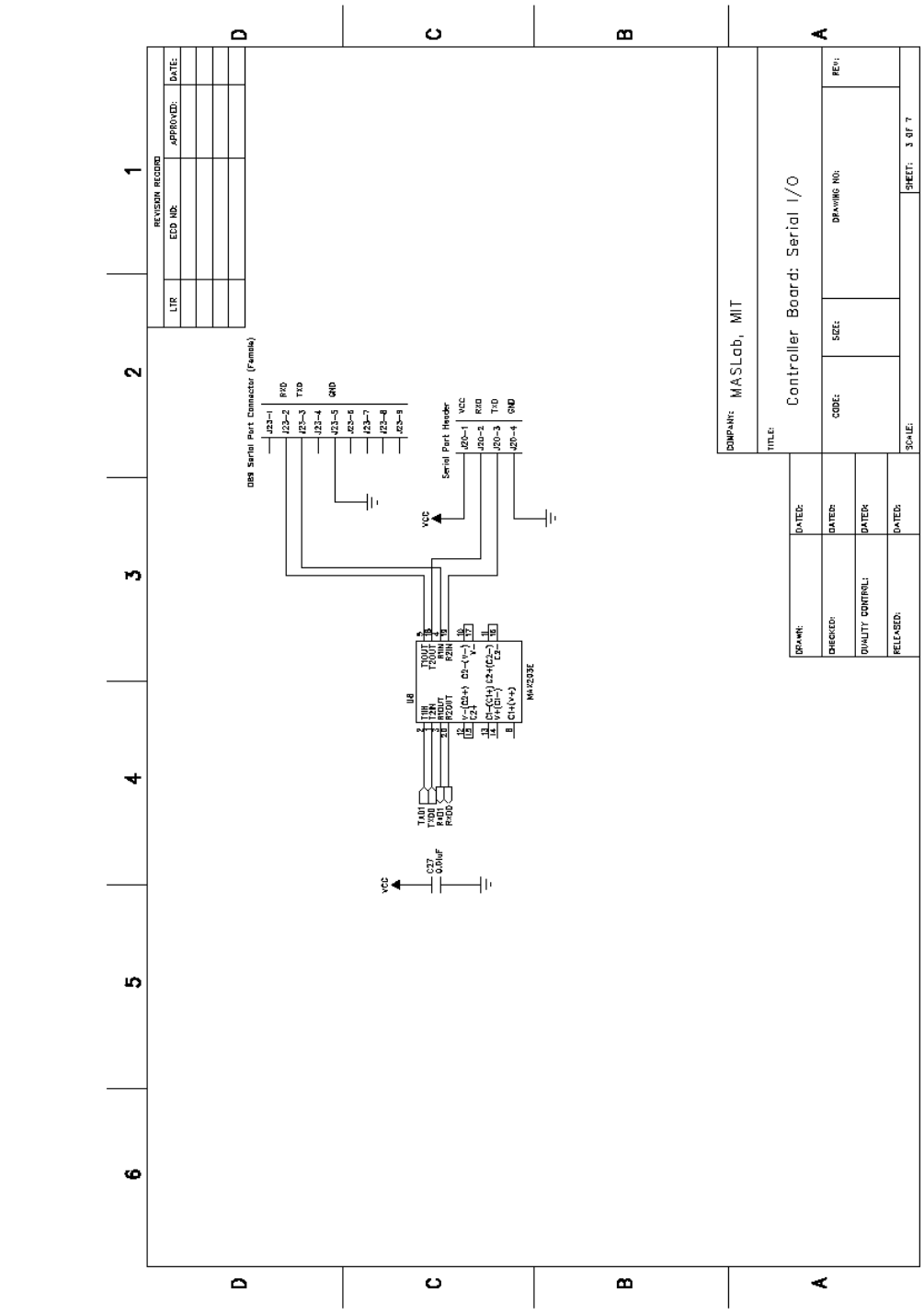
- Servo connectors were omitted, but servo control outputs are accessible from the timer pin test points.
- Switching Power Supply has excessively slow transient response, which can cause the board to reset when peripherals (commonly servos) cause a current surge.
- Sensor connectors are not HandyBoard compatible (pins are in a different order)

While these schematics are otherwise functional, we strongly the reader to obtain the most up-to-date schematics from the authors before beginning manufacturing, or working on derivative designs.

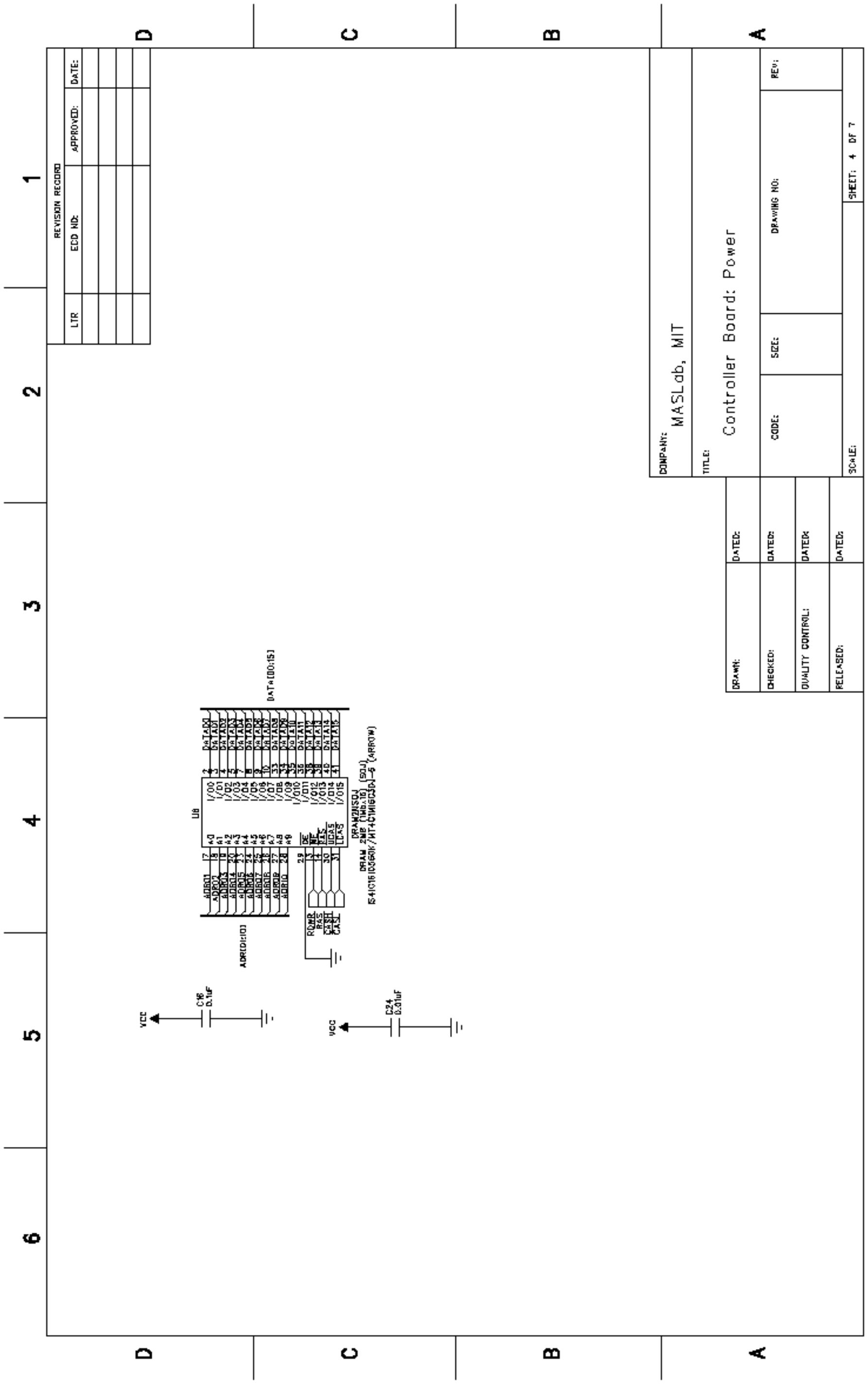

MASLab, MIT
Controller Board: Power
SHEET: 1 OF 7
V_BAT
VCC
GND
LM2595
Current-Measuring Header
Keep clear for heatsink
Replace with polarized connector
Controller Board: Power

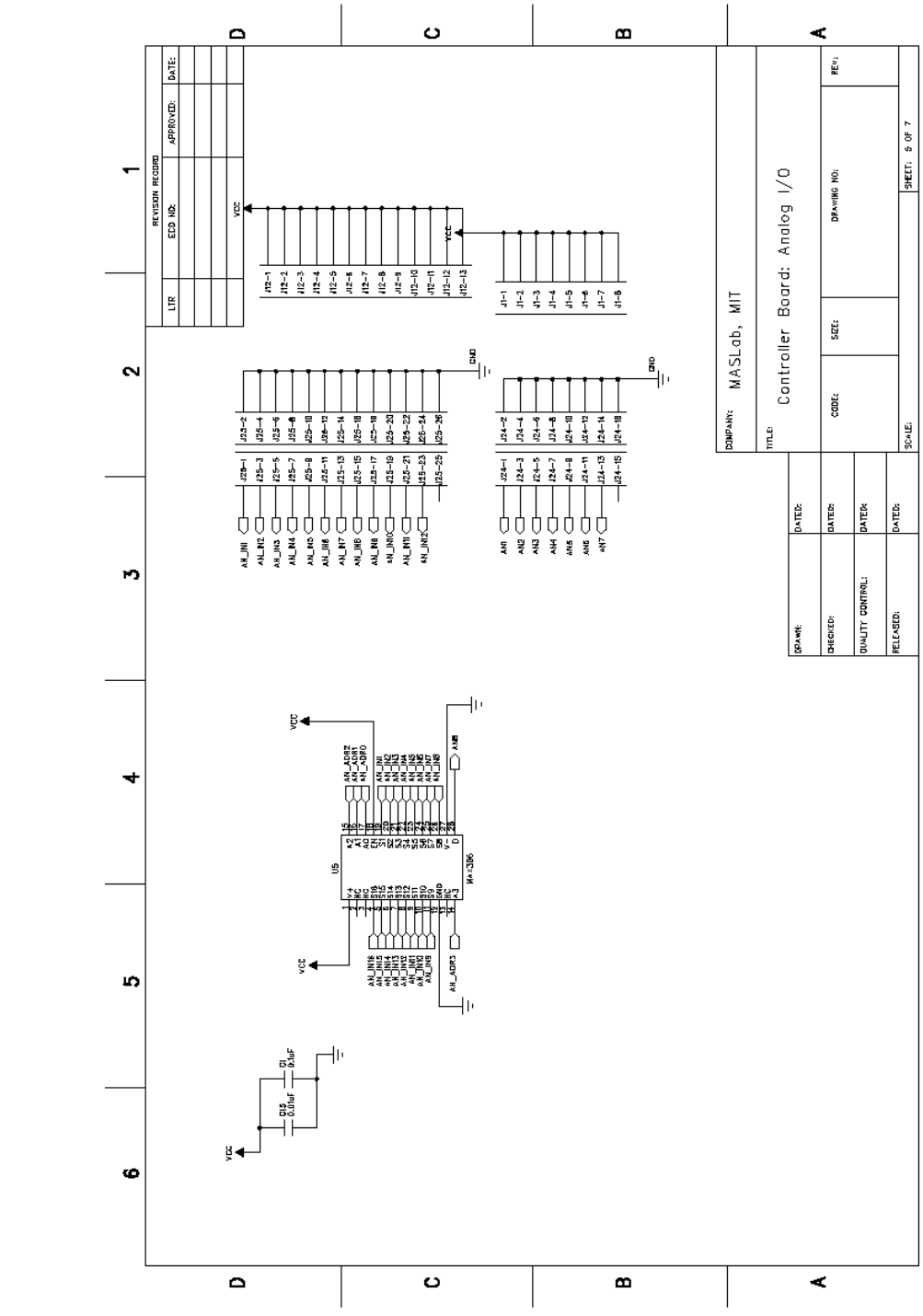
maslab15.sch-2 - Sun Apr 15 13:24:27 2001



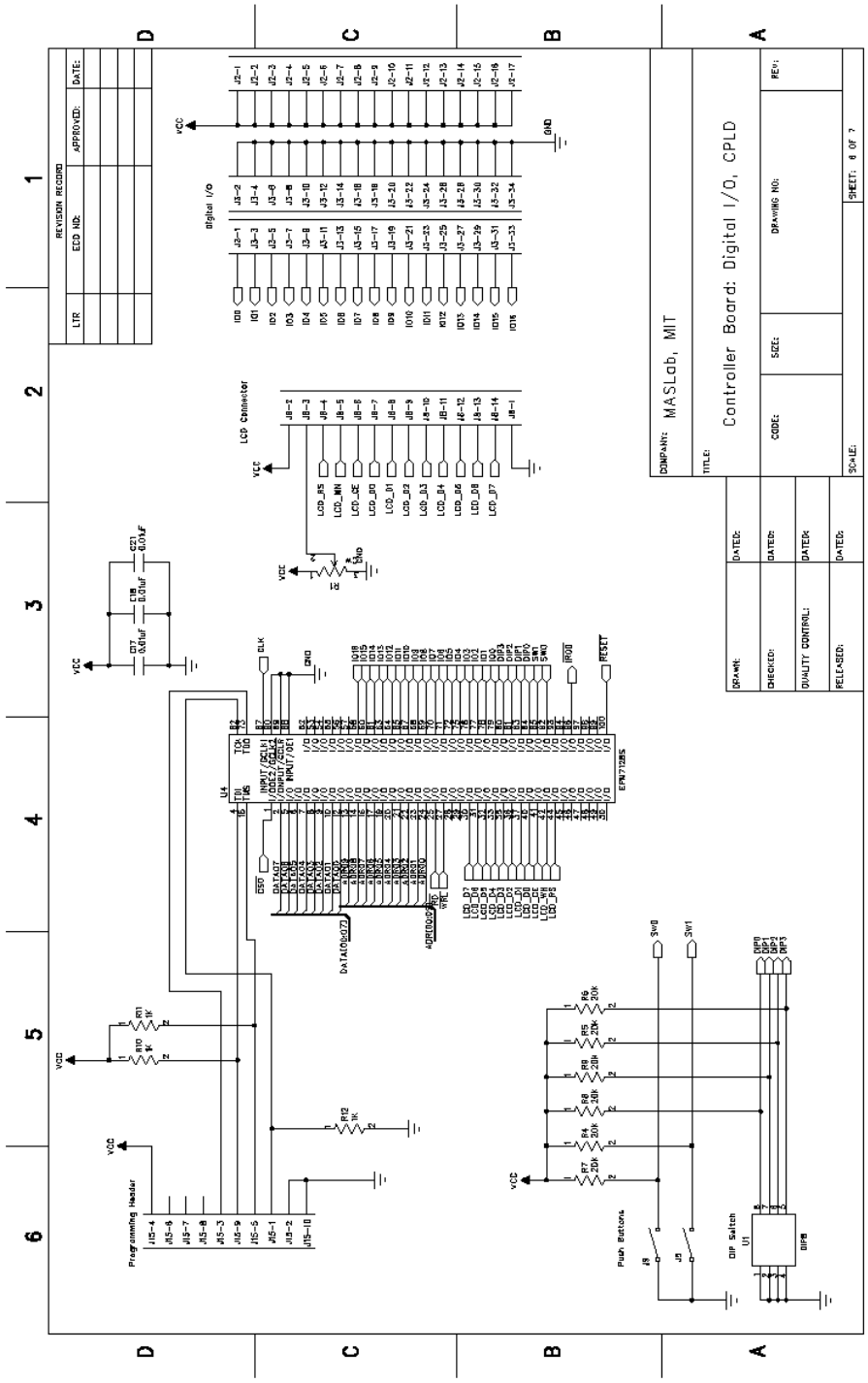

maslabl5.sch-3 - Sun Apr 15 13:24:24 2001
REVISION RECORD
LTR
ECO NO:
APPROVED:
DATE:
DB9 Serial Port Connector (Female)
J23-1
J23-2
J23-3
J23-4
J23-5
J23-6
J23-7
J23-8
J23-9
RXD
TXD
GND
Serial Port Header
J20-1
J20-2
J20-3
J20-4
VCC
RXD
TXD
GND
VCC
U8
T1IN
T2IN
R1OUT
R2OUT
T1OUT
T2OUT
R1IN
R2IN
C1+
C1-
C2+
C2-
V+
V-
MAX203E
C37
0.01uF
T1OUT
T2OUT
R1IN
R2IN
COMPANY: MASLab, MIT
TITLE:
Controller Board: Serial I/O
SIZE:
CODE:
DRAWING NO:
REV:
SCALE:
SHEET: 3 OF 7
DRAWN:
CHECKED:
QUALITY CONTROL:
RELEASED:
DATED:

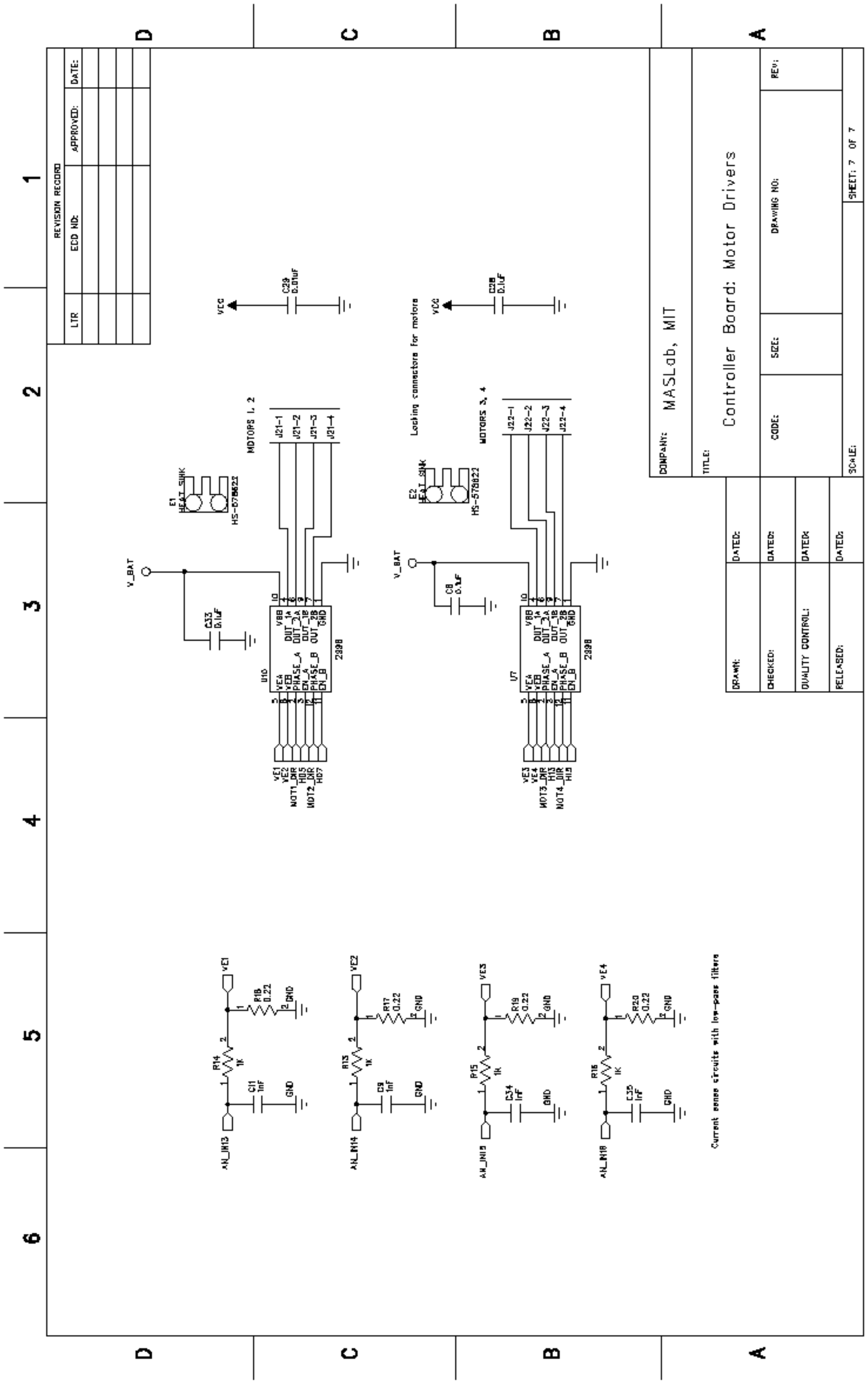
maslab15.sch-4 - Sun Apr 15 13:24:28 2001



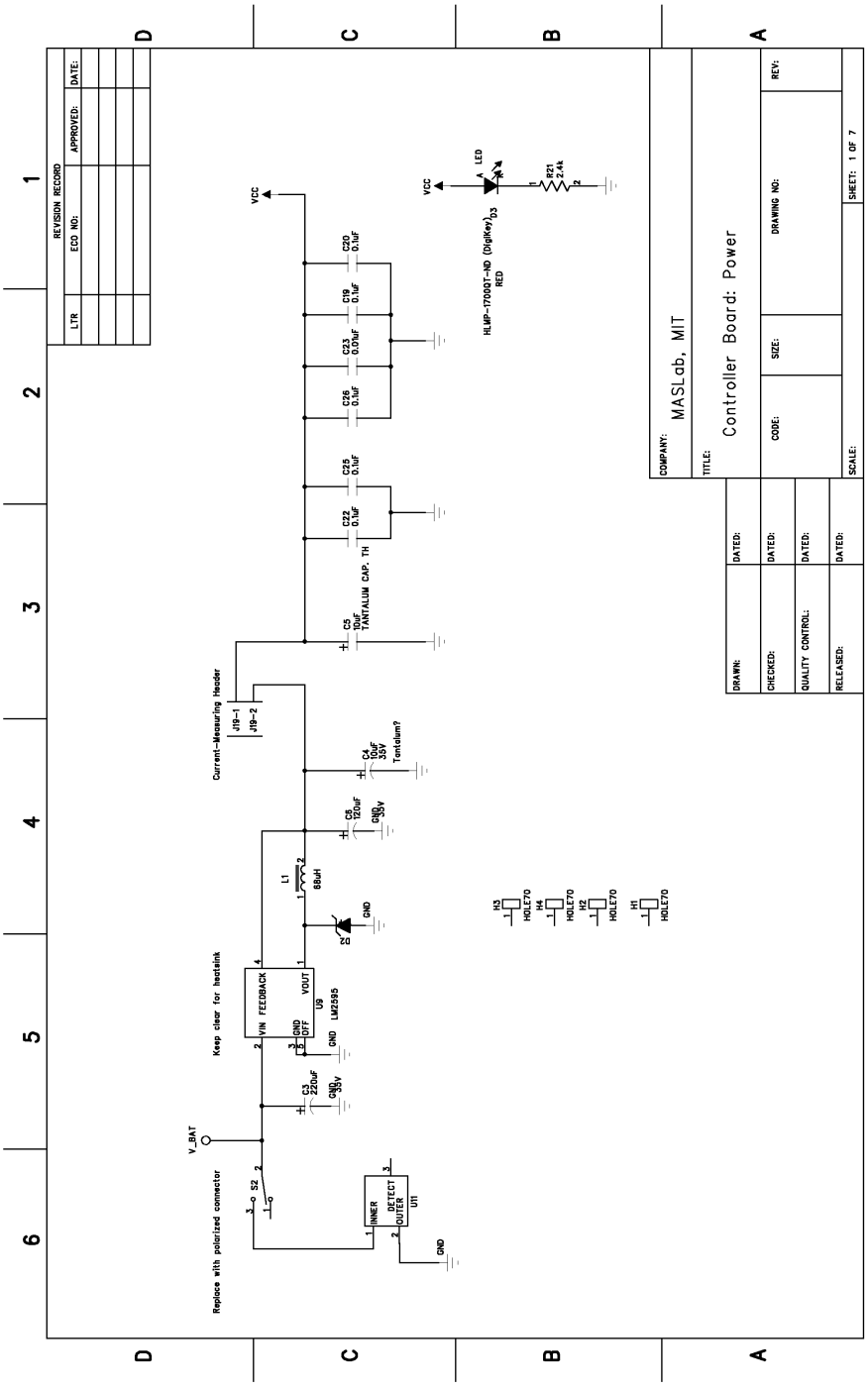

maslab15.sch-5 - Sun Apr 15 13:24:24 2001
REVISION RECORD
LTR
ECO NO:
APPROVED:
DATE:
COMPANY: MASLab, MIT
TITLE: Controller Board: Analog I/O
CODE:
SIZE:
DRAWING NO:
REV:
SHEET: 5 OF 7
SCALE:
DRAWN:
DATED:
ENGINEER:
QUALITY CONTROL:
RELEASED:
VCC
GND
J12-1
J12-2
J12-3
J12-4
J12-5
J12-6
J12-7
J12-8
J12-9
J12-10
J12-11
J12-12
J12-13
J1-1
J1-2
J1-3
J1-4
J1-5
J1-6
J1-7
J1-8
J25-2
J25-4
J25-6
J25-8
J25-10
J25-12
J25-14
J25-16
J25-18
J25-20
J25-22
J25-24
J25-26
J25-1
J25-3
J25-5
J25-7
J25-9
J25-11
J25-13
J25-15
J25-17
J25-19
J25-21
J25-23
J25-25
AN_JN1
AN_JN2
AN_JN3
AN_JN4
AN_JN5
AN_JN6
AN_JN7
AN_JN8
AN_JN9
AN_JN10
AN_JN11
AN_JN12
J24-2
J24-4
J24-6
J24-8
J24-10
J24-12
J24-14
J24-16
J24-1
J24-3
J24-5
J24-7
J24-9
J24-11
J24-13
J24-15
AN1
AN2
AN3
AN4
AN5
AN6
AN7
AN8
U5
MAX386
AN_ADDR2
AN_ADDR1
AN_ADDR0
AN_ADDR3
C1
0.1uF
C12
0.01uF

maslab15.sch-6 - Sun Apr 15 13:24:25 2001




MASLab, MIT
Controller Board: Motor Drivers
SHEET: 7 OF 7

maslab15.sch-1 - Sun Apr 15 13:24:23 2001



Controller Board: Power

References

1 Fred Martin and Brian Silverman. *Cricket System Technical Overvew.* 1997.

2 *Lego MindStorms Robotics Invention System Technical Notes*. Mondo-tronics' Robot Store.

3 Fred Martin. *The Handy Board Technical Reference*. 2000.

4 Personal communication with Fred Martin

5 Interactive C. http://www.newtonlabs.com/ic/

6 *Compaq Personal Server Specification*. Compaq Cambridge Research Laboratory.

7 Gee, et al. *Cache performance of the SPEC92 benchmark suite.* IEEE Micro 13:4, 17-27.

8 LM340/LM78MXX Series 3-Terminal Positive Regulators. National Semiconductor.

9 Optrex Corporation. *LCD Liquid Crystal DisplayDatabook*. 1998.

10 MIT Electronic Laboratory Safety Guidelines

11 gdbstubs OpenSource project, led by Bill Gatliff, Hosted on SourceForge

12 Tannenbaum. *Modern Operating Systems*.

13 Trinity Fire Fighting web page. http://www.trincoll.edu/events/robot/index.html

14 MIT's 6.270. http://www.mit.edu/~6.270/

15 Bajracharya and Olson. *A Low-Cost, High-Performance Robotics Platform for Education and Research*. AAAI Symposium on Robotics and Education, 2001.