

OrcBoard Manual **v5**

Edwin Olson
eolson@mit.edu
(C) 2005 by Edwin Olson

January 6, 2009

Contents

1	Introduction	5
1.1	Feature Summary	5
2	Getting Started	9
2.1	Finishing Assembly	9
2.2	Power Supply	10
2.3	Starting and using Orcd	10
2.4	A simple Java program	11
3	Using Sensors	15
4	User's Guide	17
4.1	Digital Input/Output	17
4.2	Analog/Digital Converters and Filters	17
4.3	Gyros	18
4.4	Motor drivers and control	18
4.4.1	MD-22 Mode	18
4.5	Programmable PWM generators	19
4.6	Configuring and using Orcd	19
4.7	Interactive Mode	19
4.8	Using the built-in firmware diagnostics	19
4.9	Updating your firmware	20
5	Developer Information	21
5.1	Firmware overview	21
5.1.1	Contributing Code	21
5.1.2	Runtime Architecture	21
5.1.3	Packet Queues	21
5.1.4	Data Flow	21
5.1.5	Important Implementation Notes	22
5.2	Packet protocol	22
5.2.1	Transaction IDs	22
5.2.2	Command IDs	23
5.2.3	Status Code	23
5.2.4	Checksum	23
5.3	FPGA	23
6	OrcBoard Schematics	25
7	OrcPad Schematics	33
8	Electrical Characteristics	37

Chapter 1

Introduction

The OrcBoard is a flexible I/O board designed to allow a standard computer to interface with the “real world.” The OrcBoard uses a full-speed (11 Mb/s) USB port. In particular, the OrcBoard was designed with robots in mind, and for many robots, the OrcBoard is the only I/O device needed.

The OrcPad complements the OrcBoard by providing an LCD and push buttons. These human-interface devices can be controlled by the PC.

In addition, the OrcBoard and OrcPad contain a large number of built-in features which allow sensors to be tested without the need for an external PC. For example, built-in firmware allows most robots to be driven around by using the OrcPad’s joystick.

1.1 Feature Summary

The OrcBoard incorporates a great deal of I/O, including Analog-to-Digital converters, Digital In/Out, Futaba-style servo controllers, SRF04-style ultrasound rangefinders, and more. An I2C bus and SPI bus are available for expansion.

The analog input pins use a 12-bit analog-to-digital converter, and the OrcBoard offers user-programmable time averaging to generate higher-precision estimates at the expense of update rate and latency.

In addition, the OrcBoard has a pulse-width measuring capability useful for sonar range finders. The board has only one of these integrators, however the input pin can be selected dynamically from any of the pins 0-23.

The OrcBoard can drive four bi-directional motors, supporting continuous current draws of over 1.0 amp (at 12V). The current actually being drawn by the motor is measured by the OrcBoard and can be returned on command. There are also four high-speed quadrature phase decoders capable of counting rates over 10MHz.

The OrcBoard has a fairly fast embedded CPU running at 16MIPS (ATMega 128 at 16MHz), with an FPGA (Altera Flex 6000) implementing much of the digital functionality. Since the firmware is open source and thoroughly commented, users can easily modify the firmware to implement new features. For example, custom control loops can be run on the OrcBoard without intervention from the host. This is especially useful when there are hard real-time constraints which must be satisfied.

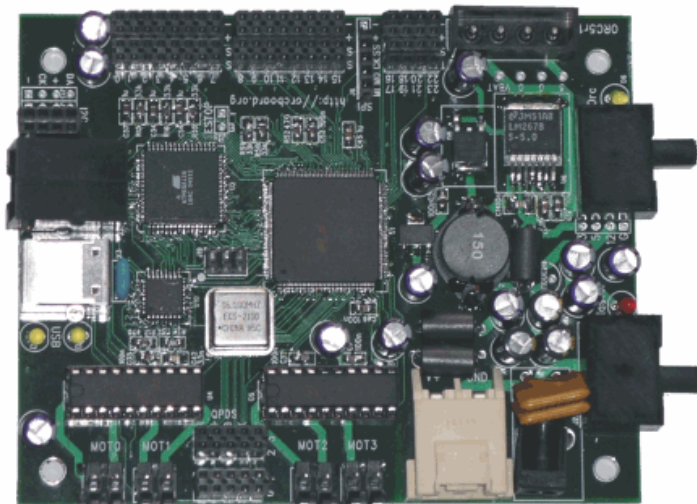


Figure 1.1: OrcBoard and OrcPad photos. The OrcBoard measures 3.6" by 4.5". The OrcPad measures 2.25" by 5.5".

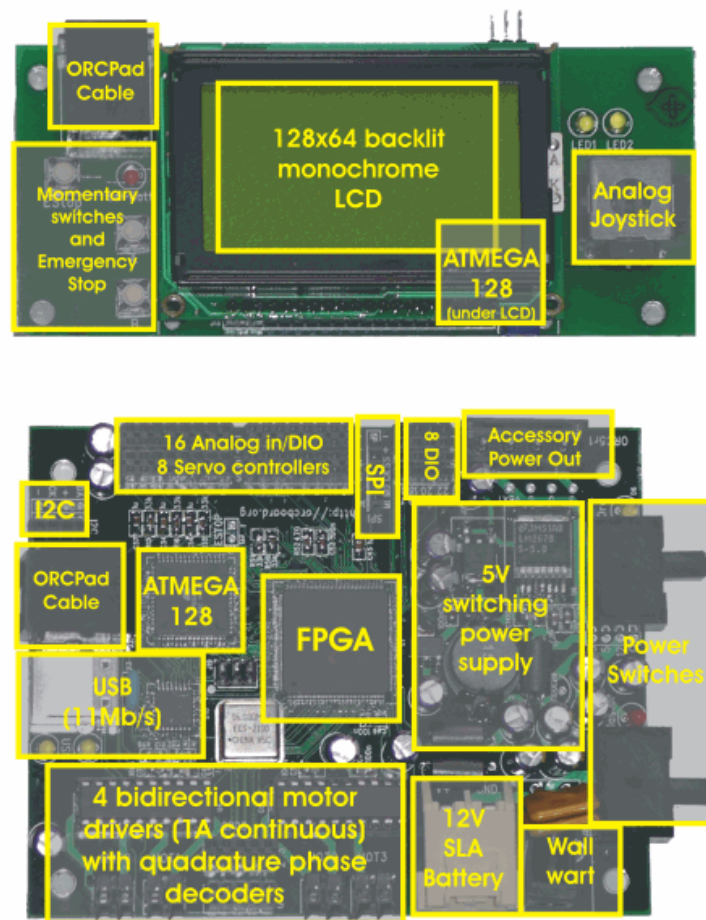


Figure 1.2: OrcBoard and OrcPad photos with functional overlay.

Chapter 2

Getting Started

Before doing anything with your new OrcBoard, please carefully read the following warnings!

Warning! The RJ-45 connector on the OrcBoard and OrcPad are *NOT* ethernet connectors; use them only to connect the OrcBoard to the OrcPad, and never to an ethernet port.

Warning! Do not attempt to power the OrcBoard from the accessory power plug with, for example, a computer's power supply. This connector is a power *output*, not a power input.

Warning! When attaching a battery, *always* use an inline fuse. Even a modest lead-acid battery can discharge at a rate of hundreds of watts (in the case of a short), which can cause burns or fires.

General electronics precautions also apply: do not set the board down on a conductive surface (you might short the back of the board), observe static-sensitive part precautions (i.e., put the OrcBoard into an enclosure so you don't zap it), and always check the polarity of connectors before inserting.

Now, let's get your OrcBoard up and running! All of the software you'll need is freely available at <http://www.orcboard.org>.

2.1 Finishing Assembly

Chances are that your OrcBoard and OrcPad are completely assembled, except for a few minor things.

A small plastic "thumb knob" should be attached to the OrcPad joystick.

1. Remove the protective paper from both sides of the plastic nub.
2. A small rectangular hole has been cut all the way through the knob, but it is slightly tapered: one opening is slightly bigger than the other. Insert the wider side down onto the joystick shaft. Push with some force until the metal shaft is about half-way inserted into the plastic nub. It should fit rather snugly.
3. To prevent the nub from being pushed too far down the joystick shaft, add a drop of super glue (not included) to the top of the hole, making sure that the glue fills the cavity.

You'll need to make motor and battery cables which are compatible with the connectors on the OrcBoard. The battery cable connector is somewhat unusual (though readily available from digikey), so one has been provided.

2.2 Power Supply

Power supply issues are extremely important when designing a robot. The OrcBoard is designed to run from *single* voltage source, rather than requiring isolated supplies for the electronics and motors. In addition, the OrcBoard supports in-system recharging simply by simultaneously connecting the battery and a suitable wall adapter.

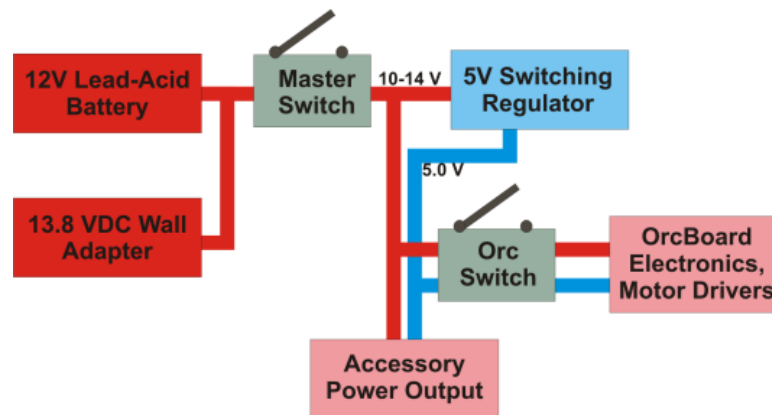


Figure 2.1: OrcBoard power system. The OrcBoard receives power from either a battery or wall adapter. This voltage can range from 10-14V depending on battery state, and is used to power the motors. A 5V supply is generated to power the electronics.

The OrcBoard has two switches. In typical operation, both will be switched into the “on” position. When the robot is not in use, the battery’s charge level can be preserved by switching the master switch “off”, which will eliminate the small but noticeable current draw caused by the 5V switching regulator and a power indicator LED.

The accessory power output connector is a convenient source from which to power other accessories. It provides both 5V and ~ 12 V, and while the 5V supply is well-regulated, the ~ 12 V supply is *not*. It is directly connected to the battery, and can fluctuate between 10-14V depending on the state of the battery. In addition, motor transients can cause both brown-outs and surges. For this reason, the ~ 12 V rail should only be used after careful consideration of the accessory’s immunity to such noise.

2.3 Starting and using Orcd

Under most versions of Linux (kernel $\geq 2.6.9$, and many older versions), the drivers for the OrcBoard are included by default (the OrcBoard uses the generic `ftdi_sio` module.) When the OrcBoard is powered and plugged into the USB port, it should be automatically recognized and appear as a device like `/dev/ttyUSB0`. A udev rule can help (`/etc/udev/rules/orc.rules`) ensure it is consistently mapped to the same device, for example `/dev/orc`:

```
DRIVER="ftdi_sio", NAME="orc", MODE="0777"
```

Orcd is a persistent daemon which allows client applications to access the OrcBoard. Applications could open `/dev/ttyUSB0` directly, but then only one application could run at a time. It is often advantageous to allow multiple programs to communicate with the OrcBoard at once: one process could control an arm, while another controls the wheels, for example. This is also convenient for debugging: if the robot is behaving oddly, a second diagnostic program can examine or even log the OrcBoard’s sensor data.

Running `orcd` is very simple; at the shell, type:

```
% orcd -d /dev/ttyUSB0
```

Note that if the OrcBoard is reset, orcd does *not* need to be restarted; orcd will automatically reopen the port when the OrcBoard is initialized again (a udev rule will help ensure that the OrcBoard is always assigned the same device name.) Consequently, orcd can be safely run in your system’s startup scripts. Orcd has a thorough help screen accessible via the command-line option “--help”.

On a related note: whenever the OrcBoard is reset (due to a power failure, or a switch being flipped “off” and “on” again), it reinitializes all of its internal state to default values. For example, motors are stopped, and all expansion ports are reconfigured to be inputs. This reinitialization will cause many user programs to stop working properly; they must be restarted as well.

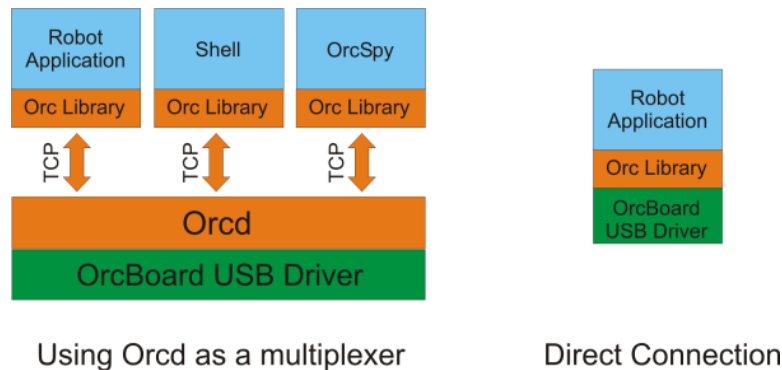


Figure 2.2: In a typical Linux environment, only one process can connect to a USB device like the OrcBoard at the same time. This makes it difficult to run multiple applications at once, including debugging utilities like OrcSpy. Orcd allows multiple applications to share access to the OrcBoard by multiplexing their transactions.

Orcd also provides a simple interface to the OrcBoard’s interactive command prompt. Connect the OrcBoard, run orcd, and simply telnet to localhost, port 7321. (Normal command traffic is carried on port 7320.) A wide variety of diagnostic commands are implemented; type “help” to get started.

2.4 A simple Java program

A simple robot program is given below. It assumes a differentially-driven robot with a collision-detection switch mounted on the front. Whenever the collision-detection switch is pressed, the robot stops. Otherwise, it drives forward.

Note the basic structure of the program: the Orc object is created first, then the individual devices (like DigitalInputs and Motors.) These wrapper classes provide a convenient way of holding all of the configuration information (like which port the device is connected to) for a device.

Consider the motors of a differentially-driven robot. One is mounted 180 degrees backwards from the other. That means that when we apply +12V to both motors, the robot doesn’t go straight, it turns because one wheel is going backwards! One of the features of the Motor wrapper is that it allows you to specify a “backwards” flag at instantiation time, so that you flip one of the motors around. Once flipped, you don’t have to worry about which motor is backwards: you can simply tell both motors to go forward.

The DigitalInput wrapper has a similar “invert” flag, which is useful when a switch is wired up such that it is logically the complement of what you want. Consider a collision-detection switch; in the event of a collision, it is natural to expect `collisionButton.read()` to return true. But most switches are normally-open, and with the standard pull-up sensor cable (see Chapter 3), `collisionButton.read()` will return the opposite of the expected value. If the DigitalInput object is instantiated with the invert flag set to true, this problem is corrected.

More complicated sensors have even more parameters; infrared analog range finders like the Sharp GP2D12 produce an output voltage that is a complicated function of distance. The wrapper class for these sensors accepts the parameters for a function which converts voltage to distance so that calling `rangeFinder.read()` simply returns distance in meters.

In short, you should write code that makes sense when you read it; the details of *where* and *how* the devices are connected should be abstracted away by instantiating wrappers with the appropriate options set.

```
import org.orcboard.orc.*;

public class SimpleRobot
{
    public static void main(String args[])
    {
        // makeOrc tries to automatically locate the OrcBoard and connect to it.
        // It's better than calling new Orc() yourself.
        Orc orc = Orc.makeOrc();

        orc.lcdConsoleWrite("Hello, World");

        // Create a new DigitalInput wrapper on port 0, with the "invert"
        // flag set to true, so that it reads "true" when there's a collision.

        DigitalInput collisionButton = new DigitalInput(orc, 0, true);

        // Our left motor is on port 0, right motor is on port 2.
        // The left motor is rotated 180 degrees from the right motor,
        // thus it is running "backwards". So we invert the left motor,
        // but do not invert the right motor.

        Motor leftMotor = new Motor(orc, 0, true);
        Motor rightMotor = new Motor(orc, 2, false);

        // Loop forever...

        while (true)
        {
            if (collisionButton.read())
            {
                // We've hit something! Stop!

                leftMotor.set(0);
                rightMotor.set(0);
            }
            else
            {
                // We haven't hit anything. Full speed ahead.

                leftMotor.set(1);
                rightMotor.set(1);
            }

            // Wait 10 milliseconds before looping.
            try {
                Thread.sleep(10);
            } catch (InterruptedException ex) {
```

```
        // (Unfortunately, Thread.sleep can throw an exception,  
        // but there's nothing we can do about it.)  
    }  
}  
}  
}
```

This program must be compiled and run against the orc.jar library, e.g.:

```
% javac -cp orc.jar:. SimpleRobot.java  
% java -cp orc.jar:. SimpleRobot
```

You can terminate this program by hitting Control-C. Upon program exit, the orc library will stop all motors automatically.

Chapter 3

Using Sensors

Warning! Plugging a sensor (which outputs a value) into a pin configured as a digital output will damage the OrcBoard. Use of a current-limiting resistor of 1k will prevent such damage.

Warning! Inadvertently connecting power to ground will damage the OrcBoard. Insert sensors with the OrcBoard powered off, and check the polarity of the connector before powering.

The standard OrcBoard sensor connector is a female receptacle. Sensors connect using male header.

There are two types of sensor ports on the OrcBoard: 3 pin, and 4 pin. Both connectors provide +5V and ground (though the amount of current available is limited); the other pin (or pins) are signal pins.

On all 3 pin connectors (ports 0-15), the signal pin may be used as a digital input, digital output, analog input. A servo-compatible PWM output is available on ports 0-7. Additionally ports 0-2 support “current sense”: they measure how much current the sensor (or servo) is consuming. Since a motor’s current is proportional to torque, this allows one to measure how much force a servo is exerting.

The 4 pin connectors (ports 16-23) have two digital I/O pins. They are useful for bump sensors (two buttons can be wired up per port), and for ultrasound range finders (which require both a “pulse” and an “echo” signal pin.) Analog input is not supported on these ports.

It is worth the time and effort to create reliable, rugged connectors. Debugging problems caused by an intermittent sensor connection can be very frustrating.

Notice that virtually every connector on the OrcBoard has “ground” on the *outside* of the board. The convention of OrcBoard cables is to call the ground wire “pin 1”, and to color that wire to designate it as such. The color coding thus provides a polarization cue: the colored wire should be on the outside of the board.

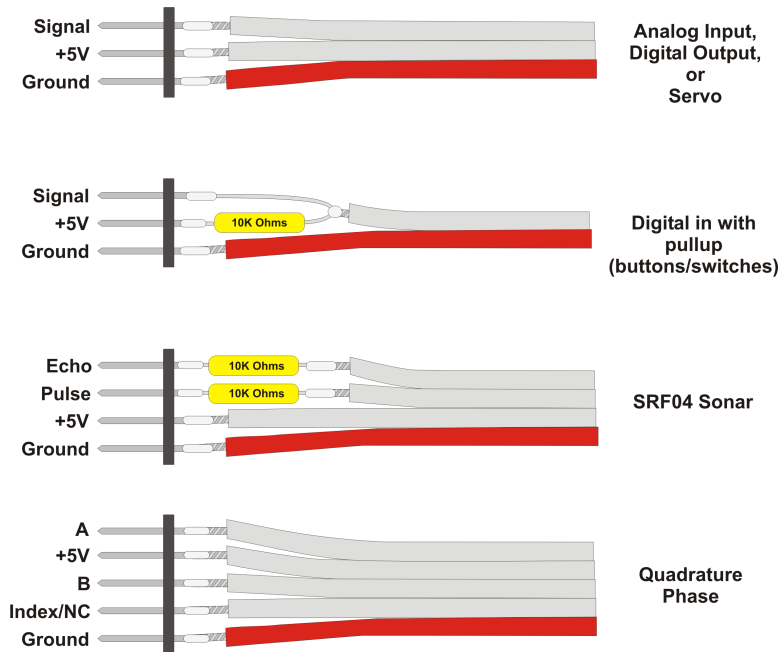


Figure 3.1: Common sensor cable configurations.

Chapter 4

User's Guide

4.1 Digital Input/Output

Each pin on the OrcBoard can be used as either a digital input or output. These pins do not have integrated pull-up resistors; these can be easily provided externally either as part of the cable assembly (recommended), or by populating the corresponding pull-up resistor 0603 resistor on the back of the OrcBoard.

Digital outputs are TTL and CMOS compatible. A digital output “1” is represented by a voltage around 3.7 V, somewhat less than the VCC power rail. If rail-to-rail outputs are required, the FPGA can be reconfigured to use open-collector outputs— an external pull-up resistor (10k nominal) will then pull a logic “1” up to 5.0V.

4.2 Analog/Digital Converters and Filters

Each analog input channel samples at a fixed rate, whether or not the host asks for the analog value. By default, this is 400 Hz. When the host asks for a value, the most recently sampled value is returned immediately.

Most real-world signals (in the context of robotic sensors, at least) have two useful properties:

1. They change very slowly compared to the 400 Hz sampling rate. For example, an analog range finder changes only as fast as the robot moves, and a robot does not move appreciably in 1/400th of a second.
2. They tend to be somewhat noisy. (This is a good thing!)

Even though the analog-to-digital converters on ports 0-15 are “only” 12 bit, these two properties allow us to achieve higher resolution by time-averaging the signal. It is important that the signal changes slowly, since the we will only get a meaningful average if the signal is essentially constant over our averaging period. It is also important that the signal is noisier than the quantization noise (1.2mV), otherwise the quantization noise (which is deterministic and non-linear) will prevent the average from producing a higher-resolution estimate. (Suppose you're trying to estimate a signal of 0.5, but your converter can only read 0 and 1; for an average of many 0s and 1s to approach 0.5, the noise in the signal must be at least 0.5 so that both 0 and 1 occur with equal probability.)

This operation is known a low-pass filter, and the extra resolution comes for “free”, because the firmware maintains a full 16 bits of resolution, even though the converter produces only 12 bits.

The low-pass filter is implemented using an infinite impulse response (IIR) filter. Given a raw ADC value $x[n]$ (the n indicates the time step), the filtered value $y[n]$ is computed:

$$y[n] = \alpha y[n - 1] + (1 - \alpha)x[n] \quad (4.1)$$

Large values of α increases the “strength” of the filter. Setting $\alpha = 0$ disables the filter completely. The firmware implements α with a sixteen bit number in the range $[0, 65535]$. Note that if this parameter exceeds about 65000, the filter will start producing erroneous results due to excessive truncation error when it evaluates Eqn. 4.1.

4.3 Gyros

Analog Device gyros are simply analog inputs. The gyros produce an analog output voltage that is proportional to *angular rate*. If integrated over time, the heading of the robot can be computed.

Readings from the gyro can be processed by user software using the built-in API, or you can use the firmware’s built-in gyro facility to perform the integration for you. The firmware supports only one gyro, but it samples the gyro at the full ADC rate, producing better estimates than would be easily possible by sending commands over USB.

The proper connections to the gyro are:

4.4 Motor drivers and control

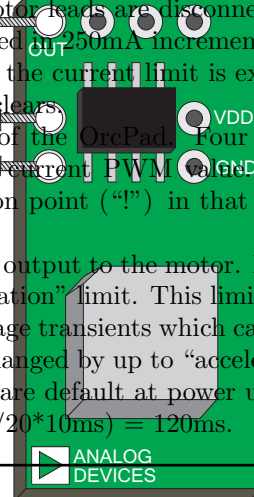
The OrcBoard supports four bi-directional motor driver ports, each capable of sourcing about 1A of current continuously. Each motor is controlled by an 8 bit command comprised of a 1 bit direction and 7 bits of PWM (i.e., sign-magnitude.) Consequently, there are *two* zeros, 0x00 and 0x80. The former is mapped to “brake” (the motor leads are internally shorted together), and the latter is mapped to “idle” (the motor leads are disconnected.)

The firmware enforces a current limit which can be adjusted in 250mA increments using the firmware diagnostics (press A+B on the OrcPad.) When the current limit is exceeded, the PWM value sent to the motor is reduced until the fault clears.

The state of the motors is summarized on the top line of the OrcPad. Four vertical “magnitude” indicators show the magnitude of the motor’s current PWM value. In the event of an over-current, the display will show an exclamation point (“!”) in that motor’s display.

The PWM values commanded by the host are not directly output to the motor. Instead, the PWM value output to the motor is subject to an “acceleration” limit. This limit causes PWM values to vary more slowly, helping to reduce large voltage transients which can result from rapid changes in PWM. At 100 Hz, the PWM may be changed by up to “acceleration” amount. Larger values imply faster accelerations. The firmware default at power up is 20. Consequently, to go from PWM=0 to PWM=255 takes $(255/20*10\text{ms}) = 120\text{ms}$.

4.4.1 MD-22 Mode



(To be written)

4.5 Programmable PWM generators

The PWM generators, available on ports 0-7, have programmable clock rates (in addition to programmable duty cycle). The default value is suitable for controlling most Futaba-style servos.

(To be written)

4.6 Configuring and using Orcd

Orcd allows multiple applications to use the same OrcBoard and OrcPad by multiplexing their communications. This can come in handy for debugging: you can separate your main robot code from the code which dumps out all of the sensor data.

Orcd also has a handy “shell” built in. Many OrcBoard systems do not have a display, aside from the OrcPad. The “OrcShell” is a very simple application, built into orcd, that allows you to run applications on the host computer. (Currently only Linux is supported.)

When starting Orcd, the OrcShell starts automatically if its configuration file can be located (by default in `/etc/orcd.conf`). The source code distribution of Orcd contains a heavily-documented example, but the short story is that you’re describing a menu: each line is composed of a label and a command. The label is what’s shown in the menu, and when you select that item, the corresponding command is executed. For example, you could put:

```
My Robot Program = /home/eolson/robot/robot_program
HelloPad = /usr/bin/java -cp /root/orc.jar org.orcboard.examples.HelloPad
```

Note that it’s important to specify the full paths to executables and jar files; OrcShell does not read your `.bashrc` or other shell configuration files. It simply invokes `/bin/sh`.

A couple built-in commands provide special features (such as `!reload`, `!standby`, `!ipaddresses`); these are described in the example configuration file.

The OrcShell will automatically hide itself whenever another program is connected, and reappear when that program ends. If you launch a program from the OrcShell, you can kill it (forcibly) by holding down B+Stick for 2 seconds.

4.7 Interactive Mode

The firmware supports an interactive debugging interface using the STDIN command. This interface allows fairly low-level access to firmware features. The easiest way to access the interactive mode is to run `orcd`, which implements a TCP server which allows telnet-like access to the interface. Connect to port 7321 to use this interface.

This interface should not be used for primary control of the robot; it is very slow and relatively untested. However, when implementing new firmware features, it is a very easy way to interact with the board without writing several layers of API glue.

4.8 Using the built-in firmware diagnostics

The firmware on the OrcBoard includes a number of handy features for viewing and manipulating the OrcBoard without writing any code.

To enter the built-in menu, tap A+B.

Particularly useful is the “Drive Mode”, which allows you to drive a differentially-driven robot using the joystick. It assumes that the motors are connected to ports 0 and 2. If

the joystick doesn't seem to control the direction correctly, you can adjust the drive mode configuration (in the Config menu). You can both swap which motor is left/right (the letters "l" and "r"), and invert the polarity of each motor (upper and lower case). For example "lr" means the left motor is in port 0 and the right motor is in port 2, and both neither are inverted. "rL" means the right motor is in port 0 (uninverted) while the left motor is in port 2 and is inverted.

The OrcPad has a calibration mode which can be used to center the joystick. To Enter it, hold A+B+Stick for about 4 seconds. Follow the on-screen prompts to calibrate your joystick. This setting information is saved across both power cycles and firmware updates, so you will seldom have to repeat the process.

4.9 Updating your firmware

From time to time, new features are added to the firmware while bugs are removed. To get the benefit of this work, you must reflash your firmware.

There are two "kinds" of configuration information on an Atmel CPU. The "fuses" specify very simple (but fundamental) configuration options, and rarely need to be reprogrammed. The FLASH contains the firmware itself.

It is very important when updating your firmware *not* to change the fuse configuration. Incorrect fuse settings can damage the device! In the case of the OrcBoard, this is often manifested by an FPGA that gets quite warm. If this happens, power off immediately, and power on again only when ready to reflash the fuses.

Chapter 5

Developer Information

5.1 Firmware overview

5.1.1 Contributing Code

The firmware for the OrcBoard and OrcPad are open source. Bug fixes and feature improvements can be contributed by users.

Coding conventions for Java code follow Sun's Java Coding Guidelines. Coding conventions for C follow the Linux kernel programming conventions. Patches which do not follow these guidelines will generally not be accepted.

5.1.2 Runtime Architecture

The OrcBoard and OrcPad use a heavily interrupt-driven architecture, with a flexible timer architecture that allows periodic events to run. Aside from interrupts and timers, there is only one thread; this thread is responsible for reading commands, processing them, and queuing a response to be sent.

5.1.3 Packet Queues

All communications is performed using packets. On each Atmel, a pool of packets is maintained; these packets are recycled. The alternative, using `malloc()` and `free()` to allocate packets, is inferior because they are slower, they can cause heap fragmentation and unpredictable failure when the heap is exhausted, and because they are non-reentrant (which makes their use difficult from an ISR.)

Queues have a fixed capacity, which must be a power of two.

The UART and USB subsystems both maintain a send queue, which is serviced by their respective ISRs (see below).

The queues satisfy a simple invariant: the packet free list will never be empty, and puts to the packet free list will never block. The first invariant is satisfied by making the number of packets in the system greater than the number of packets which can be in send queues or otherwise "in flight". The second invariant is satisfied by making the capacity of the free list equal to the total number of packet buffers allocated. These invariants simplify several concurrency issues that could otherwise lead to deadlocks or leaked memory.

5.1.4 Data Flow

The OrcBoard serves as a communication hub between the host (USB) and the OrcPad (UART).

USB RX: USB receive is done synchronously from the main loop, rather than from an

interrupt. There is no fear of dropping characters since the FT245BM device handles flow control itself.

USB TX: USB transmission is done via an ISR. The USB device can typically accept many characters at once (since USB bandwidth is so large), so we could potentially spend a large amount of time in this ISR. Because of this, care has been taken to re-enable interrupts from within this ISR.

UART RX: Receiving data from the UART is the single most critical timing constraint in the system since the Atmel's UART FIFO is only a couple bytes deep. For speed reasons, the RX ISR dumps characters into a simple circular FIFO.

UART TX: Sending to the UART is done via an ISR which services a packet send queue.

The UART link between the OrcBoard and OrcPad uses XON/XOFF flow control, implemented at the ISR level. Once the receiver's RX ringbuffer is full, and the sender's sendQueue is full, future calls to `uart_send()` will block. Consequently, we achieve the desired throttling behavior: no more requests will be processed until the receiver catches up.

There is a deadlock scenario that can be triggered when both OrcBoard and OrcPad assert XOFF that is difficult to trigger, but does currently exist in the code. As an "escape" valve, `uart_send` will drop a packet if the deadlock has been detected.

5.1.5 Important Implementation Notes

Because a UART is used for communications between the OrcBoard and OrcPad, it is important to limit the worst-case latency of every timer and ISR. (Timers run in an interrupt context.)

In timers and ISRs, there are a number of forbidden activities:

- Calling libc functions. They are generally not reentrant. This includes, in particular, `malloc()`, `free()`, `printf()`, `snprintf()`, etc.
- Blocking. In particular, only non-blocking queue commands should be used. If the timer/ISR needs to send a packet, but a packet is not available (or the send queue is full), the ISR should reschedule itself to run later. For example, do not use `usb_send()` or `uart_send`; use `usb_send_try()` or `uart_send_try()`. Likewise, do not

5.2 Packet protocol

All communication with the OrcBoard and OrcPad is accomplished using a packet protocol. All packets begin with the value `0xED`; this aids synchronization after a communication failure.

Offset	Description
0	Always the value 237 (0xED)
1	Transaction ID
2	<i>datalen</i> : length of the data section
3	First byte of data
...	(more data)
<i>datalen</i> + 3	Modified checksum byte

5.2.1 Transaction IDs

Most request packets result in a response packet. The user does not need to wait for a response to a request before issuing another request, however response packets might arrive in an order different from the requests. To disambiguate this, each request is tagged with a "transaction id"; this transaction id will be copied into the response packet. The client is responsible for generating unique transaction ids.

There are a few exceptions. If a response is not needed for a command, transaction ID `0xF0` can be used. If a message is being sent asynchronously from the OrcBoard or

OrcPad, and doesn't correspond (in a one-to-one manner) with a request, it will use a special transaction ID. Updates of the joystick position are examples.

Transaction ID range	Used by
00-EF	Available for use by user
F0	Used to signify that no response is required
F7	Used by Orcd when broadcasting data
FD	Do not use: used in private OrcBoard/OrcPad communication
FE	Used by OrcPad when broadcasting data
FF	Used by OrcBoard when broadcasting data

Note that if Orcd is being used, the transaction id space (00-EF) must be shared by all clients. Orcd will send clients a packet indicating what subset of that space they are permitted to use. Applications not using Orcd are free to use the entire space.

5.2.2 Command IDs

Every request packet begins with a command id. Command IDs are segregated into three groups:

Command range	Used by
00-7F	OrcBoard commands
80-EF	OrcPad commands
F0-FE	Reserved for use by Orcd

When the OrcBoard receives a packet in the range 00-7F, it processes the command itself. For commands 80-EF, the packet is retransmitted to the OrcPad. When the OrcPad processes a command, it sends the response to the OrcBoard, which relays the command to the host.

The OrcBoard does not send NACKs; if ill-formatted data is received, it is silently dropped.

If the OrcBoard receives a packet destined for the OrcPad, but the OrcPad is not connected, the packet will be silently dropped.

Particular commands are documented in the source code and API documentation.

5.2.3 Status Code

Every packet sent by the OrcBoard or OrcPad begins with a one-byte status code.

5.2.4 Checksum

The checksum is computed by summing every byte in the packet, rotating to the left by one bit after each byte. As a special case, checksum values of 0xDE are *always* considered valid.

5.3 FPGA

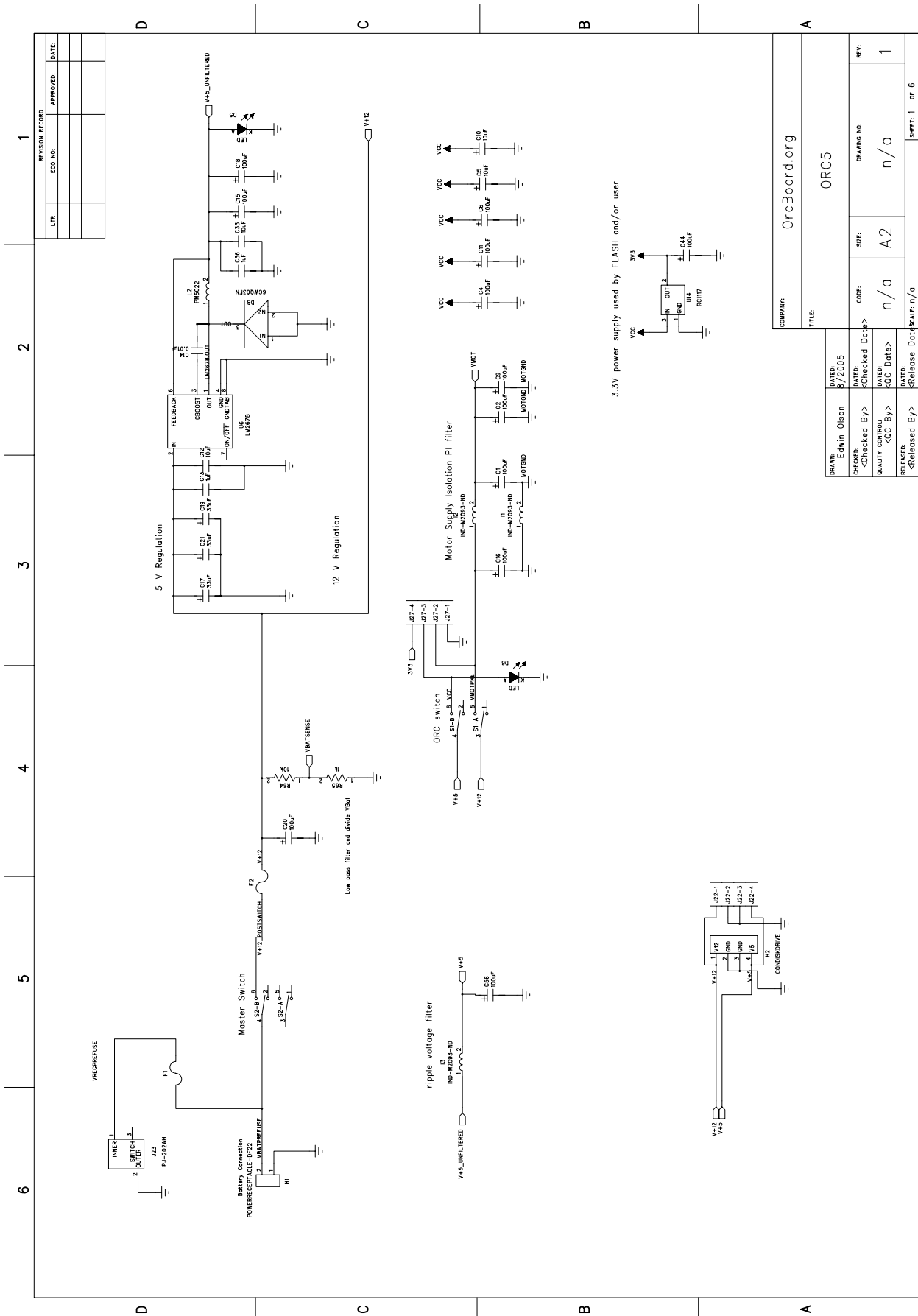
The digital I/O on the OrcBoard, including quadrature phase decoders, PWM generators (both servo and motor), and various timer channels, are implemented on an Altera 6000 FPGA. This FPGA is quite old by modern standards, but it is a good choice for the OrcBoard because it is compatible with 5V signals.

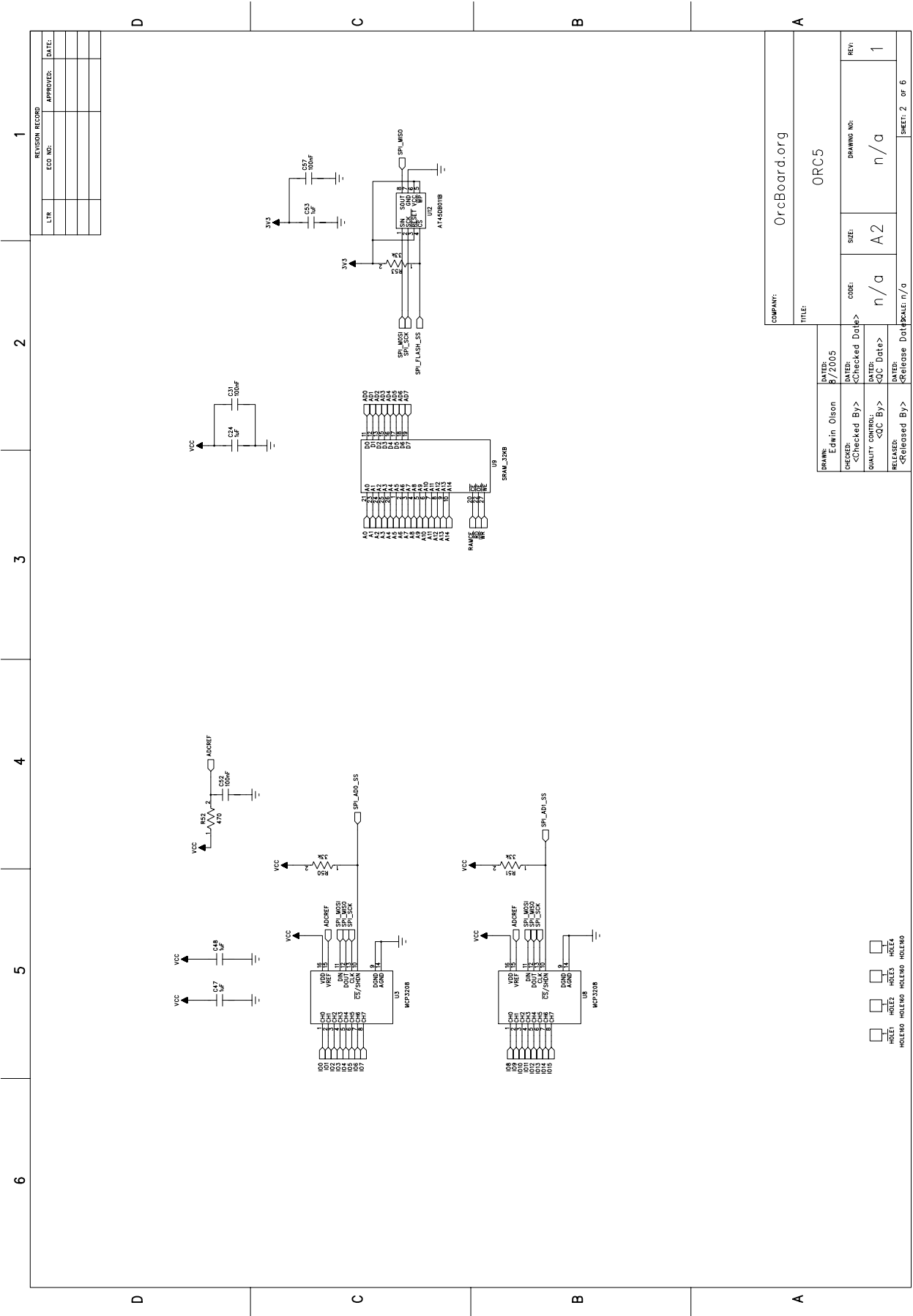
Chapter 6

OrcBoard Schematics

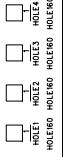
The OrcBoard was designed and laid-out using Mentor Graphics PADs, a commercial product. The schematics are organized into six pages:

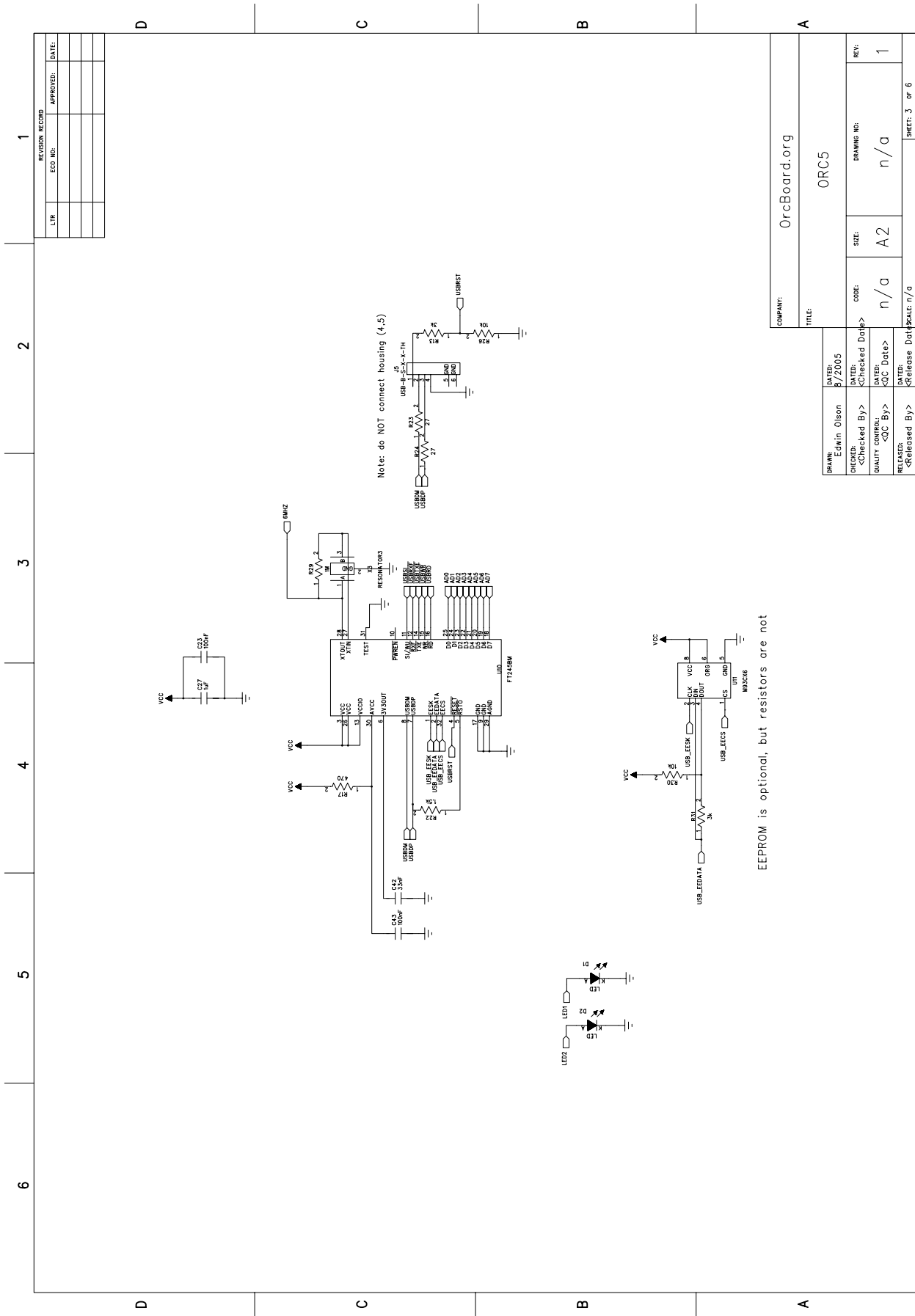
1. Power supply : connections, regulators, filtering
2. ADC converters, external SRAM and FLASH
3. USB interface
4. Atmel CPU and FPGA
5. Connectors, and more connectors
6. Motor drivers

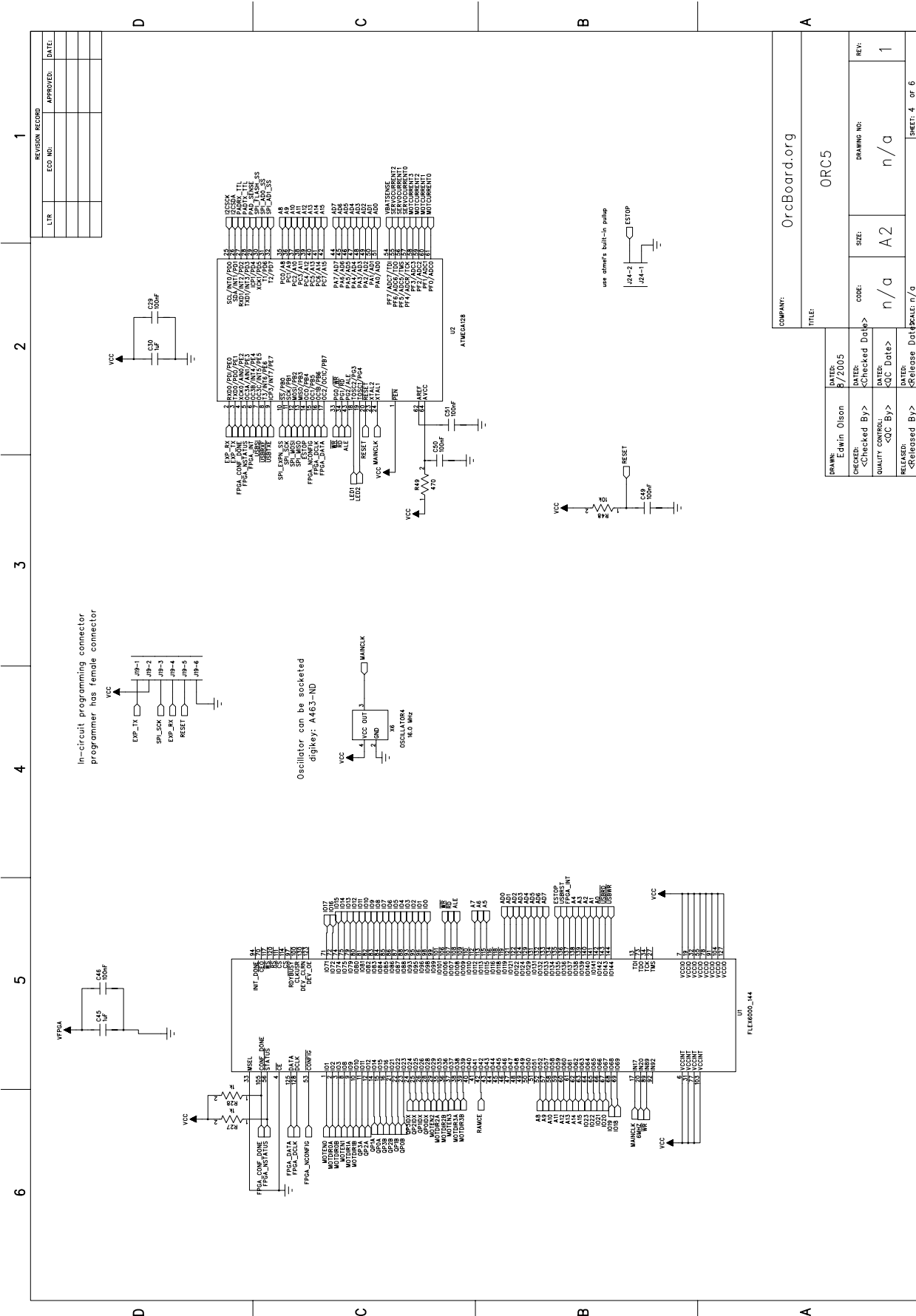


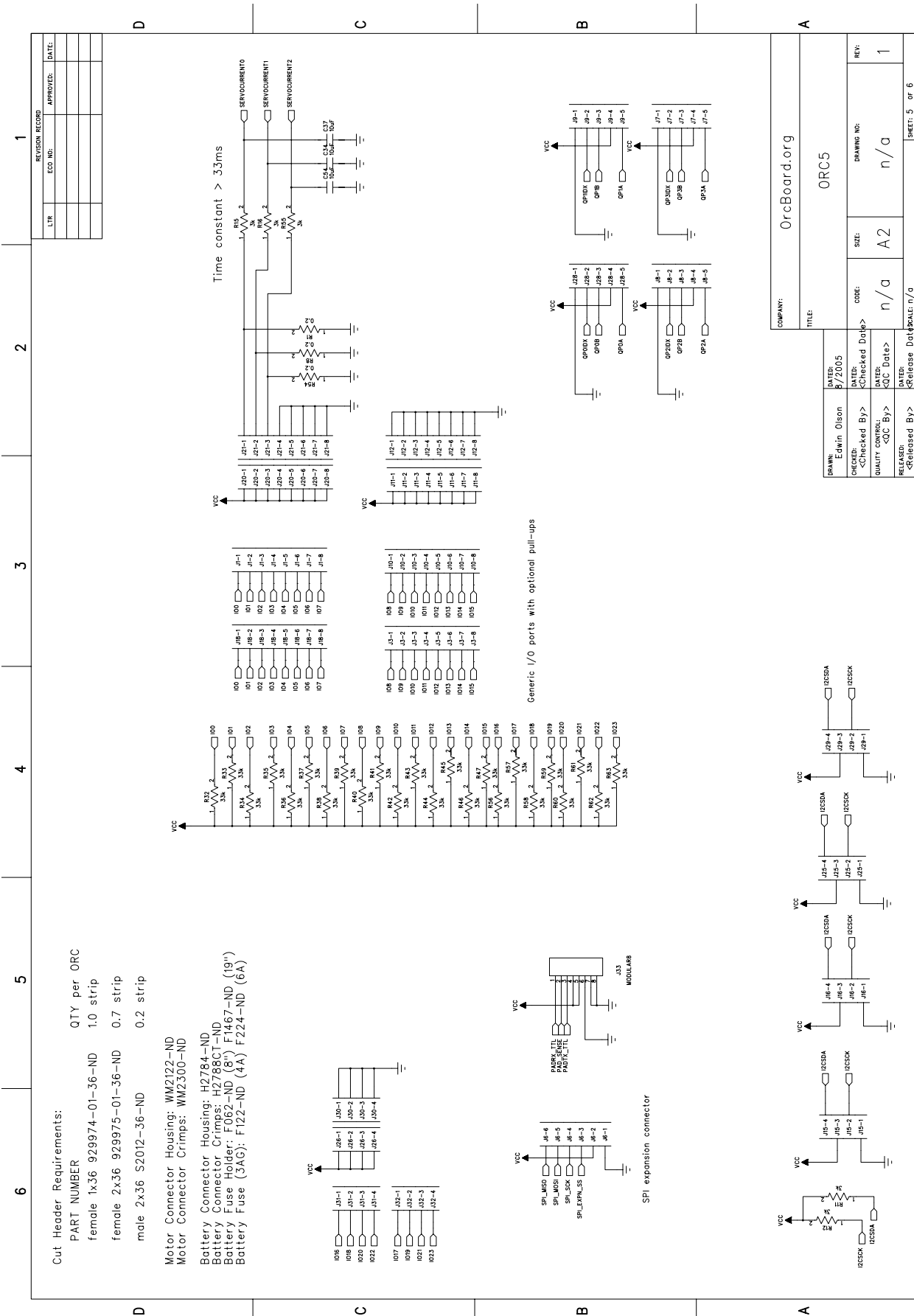


COMPANY: OrcBoard.org		TITLE: ORC5	
DRAWN: Edwin Olson	DATED: 8/20/05	CHECKED: <Checked By>	DRAWING NO: n/a
QUALITY CONTROL: <QC By>	DATED: <QC Date>	SIZE: A2	REV: 1
RELEASED: <Released By>	DATED: <Release Date>	SCALE: n/a	SHEET: 2 of 6









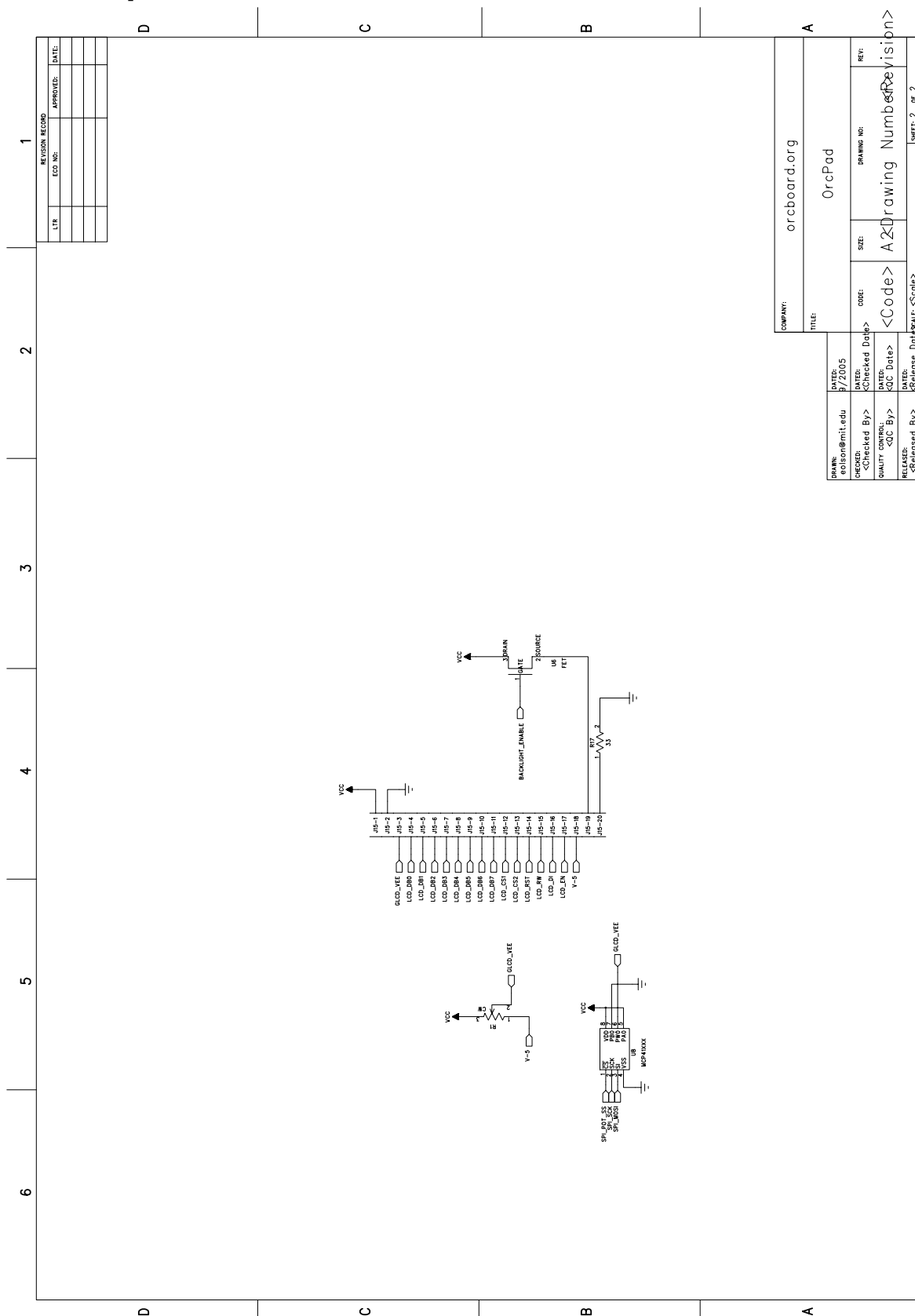
Chapter 7

OrcPad Schematics

The OrcPad was designed and laid-out using Mentor Graphics PADS, a commercial product. The schematics are organized into two pages:

1. Atmel CPU and most everything else
2. LCD connector

The OrcPad is designed to be a general-purpose hobbyist board, useful for a wide variety of applications. In the usual OrcPad configuration, most of these features are unnecessary.



Chapter 8

Electrical Characteristics

Parameter	Description	Min	Typical	Max	Units
Power Supplies					
V_{mot}	Motor voltage rail	9	12	28	V
V_{logic}	5 volt rail		4.97		V
I_{logic}	5 volt rail current		0.2	3.0	A
Motor Drivers					
I_{mot}	Continuous motor current (at 12V)			1.5	A
I_{peak}	Peak motor current (at 12V)			2.5	A
I_{supply}	OrcBoard supply current (at 12V)		15	50	mA
Digital I/O					
V_{oh}	Digital output “1” voltage	2.5	3.7	5.0	V
V_{ol}	Digital output “0” voltage	0	0	1	V
I_{io}	Output current (per pin)			5	mA
I_{iotot}	Output current (total)			40	mA
Digital Performance					
f_{CPU}	Atmel and FPGA clock		16.00		MHz
$f_{quadphase}$	QuadPhase counting rate			8	MHz
Communications Performance					
t_{lat}	Achievable Latency (Host/Board)		2.5		ms
f_{cmd}	Achievable Transaction rate (Host/Board)		5100		Hz
B/W	Achievable Bandwidth (Host/Board)		100		KB/s
t_{lat}	Achievable Latency (Host/Pad)		3.5		ms
B/W	Achievable Bandwidth (Host/Pad)		50		KB/s
f_{cmd}	Achievable Transaction rate (Host/Pad)		2600		Hz
Analog Performance					
f_{samp}	Sampling frequency		400		Hz
$E_{ADC_{res}}$	External ADC resolution		1.2		mV
$I_{ADC_{res}}$	Internal ADC resolution		2.4		mV
$C_{ursense_{res}}$	Current sense resolution		13.9		mA