The Etherio Board User's Manual

Will Ware <wware@alum.mit.edu>

$Id: etherio.tex,v 1.10 2002/12/12 04:40:13 wware Exp $

Contents

The Etherio board is a 3-inch-by-4-inch printed circuit board that offers an inexpensive and convenient way to put TTL-compatible I/O lines on your Ethernet subnet. These lines can be sensed and controlled from any computer on the network, with code written in any language that includes a networking API.

The networking protocols used by the Ethernet Digital I/O Board are all standard, and documented in this booklet. The board runs a small IP stack, and you talk to the I/O lines using UDP datagrams.

1 Hardware

Along the right edge of the Etherio board is a 34-pin ribbon cable which provides 32 I/O lines, with a ground connection and a five-volt supply. The 32 lines are grouped as four eight-bit ports (A, B, C, and D) as shown in Figure 2. There are an additional 24 I/O lines available in a prototyping area, on ports E, F, and G. The UDP command byte sequences for accessing these ports are given in Table 1.

In addition to the seven 8-bit ports, there are two 8-bit direction registers, DR0 and DR1. These set the directions of the I/O lines as inputs or outputs, in groups of four bits. The assignments of bits in DR0 and DR1 are shown in Table 2. Setting a bit high will cause the corresponding four lines to be outputs. For instance, setting DR0 to 0x01 would make ports B, C, and D all inputs, with port A bits 4-7 as inputs and port A bits 0-3 as outputs.

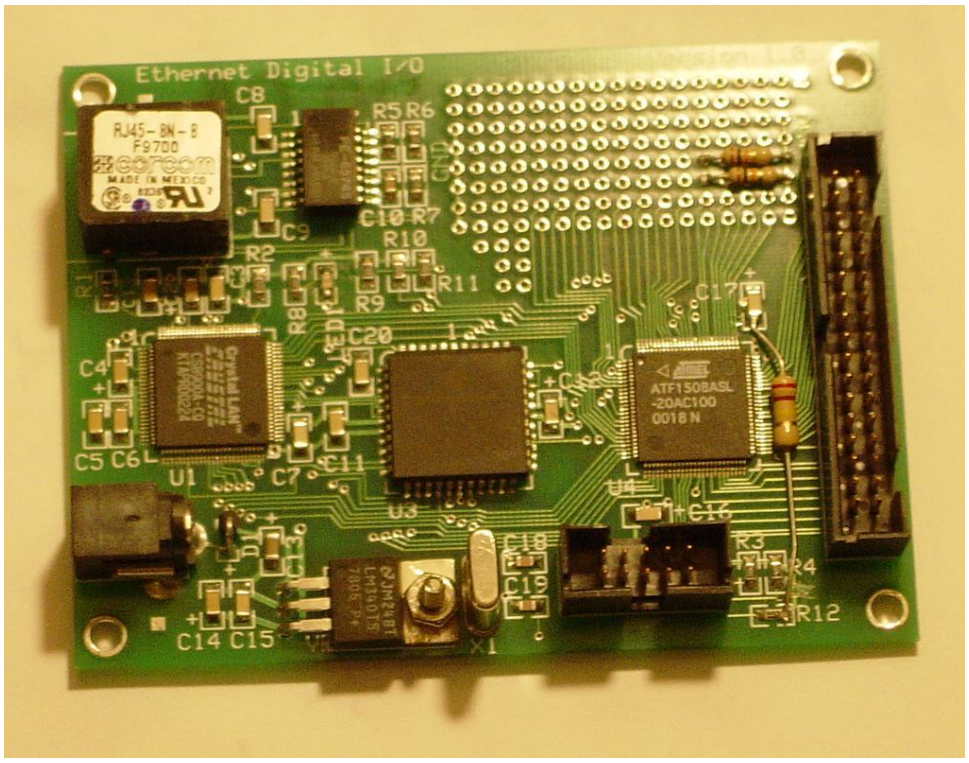

Figure 1: The Ethernet Digital I/O Board

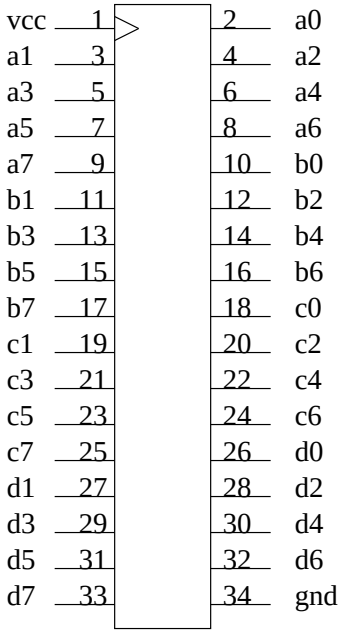

Figure 2: The ribbon cable

Operation	UDP command	UDP response
Write port A	0x00 x	y
Write port B	0x01 x	y
Write port C	0x02 x	y
Write port D	0x03 x	y
Write port E	0x04 x	y
Write port F	0x05 x	y
Write port G	0x06 x	y
Set DR0	0x08 d	d
Set DR1	0x09 d	d
Read port A	0x10	y
Read port B	0x11	y
Read port C	0x12	y
Read port D	0x13	y
Read port E	0x14	y
Read port F	0x15	y
Read port G	0x16	y
Turn LED on	0xFF 0x01	0x01
Turn LED off	0xFF 0x00	0x00

Table 1: UDP commands and responses

Direction Register	Bits							
	7	6	5	4	3	2	1	0
DR0	D4-7	D0-3	C4-7	C0-3	B4-7	B0-3	A4-7	A0-3
DR1	xx	xx	G4-7	G0-3	F4-7	F0-3	E4-7	E0-3

Table 2: Direction register bit assignments

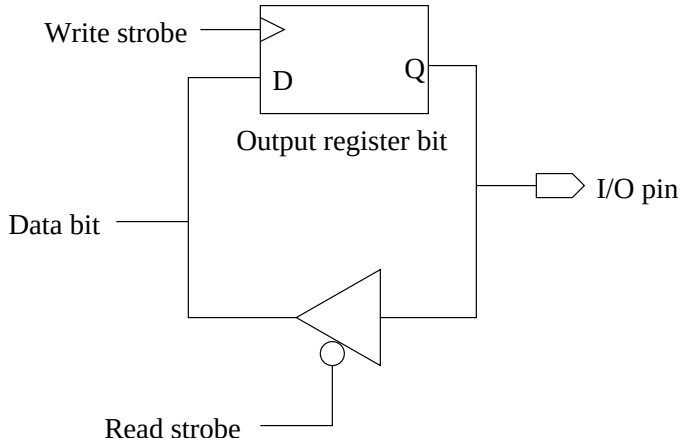

Figure 3: I/O bits are read back directly from the pin

When reading back an I/O port bit, the board reads back the voltage present at the pin, not the corresponding bit in the output register. These two can differ even for an output pin, if the register bit is set high but the pin is externally tied to ground. Please see Figure 3.

Include a diagram of pin assignments for the prototyping area. Describe the "special" I/O lines, which are useful to users who are interested in reprogramming the programmable logic chip (the CPLD). These lines are things like global resets, global clocks that reach every point on the chip with minimal time deltas, global output enables, and things like that.

A future board might have optically isolated inputs and relay outputs, or I might sell an add-on board that provides these. This would take care of the most egregious causes of hardware faults, like connecting an I/O line directly to house current.

2 Host code

The Etherio board is controlled from a host computer on the same network. The communication protocols are all standard, and are available on Windows machines, Linux and Unix machines, and Macs. The idea is to stick with standards so that you, the user, can write your own host code to do whatever you want with the board.

It might be good to write an Etherio driver in Perl or Python or C that listens for network requests in XML-RPC or HTML and grants Etherio access to other processes running on the subnet, or running elsewhere.

2.1 Basics

The Python code in Subsection 2.4 defines a class that communicates with the board, and can be used as a software representation of the board in larger programs.

I've been trying to identify a situation where the board gets into a "stuck" state and stops communicating. This can be prevented by the judicious use of a one-millisecond delay. I think the problem occurs when the board has sent out a response, and the host turns around too quickly and sends another command.

The Ethernet chip is the same one used in many desktop NICs, so it must be able to keep up. The problem must be with the PIC microcontroller. Maybe in some future version I'll go with a faster better controller. But for now this is inexpensive. I sure wouldn't mind better diagnostics than I can get out of a PIC.

When it's actually ready to sell, the board will be a DHCP client, which means that it will automatically obtain an IP address from the network's DHCP server (frequently the subnet's router) so that the host code won't need to use a hard-wired IP address. I guess it would be good to be able to tell the thing to accept a static IP address; maybe that flexibility is a feature for a future incarnation.

The other feature that will be in place when I sell the boards is a discovery protocol, which allows a host to say, "All you Ethernet I/O Boards on this subnet, identify yourselves!" And then all the boards will report in, supplying their IP addresses. The host code might then offer a user the chance to flash LEDs to identify which physical board goes with which IP address.

2.2 Fun with CGI scripts

If the subnet with the Etherio board also has a publicly accessible web server, then you can write a CGI script to read and write I/O lines in response to HTTP requests. In response to a request, the CGI script would talk to the Etherio board, get a response back, and send some dynamically generated HTML to the requestor.

The Perl function in Subsection 2.6 sends a command to the Etherio board and gets back a response. The function returns zero if the board is out of communication.

Oops! That's not right! It doesn't do the retry.

Using this function, it's easy to write a CGI script that can read inputs and set outputs. What better way to put your refrigerator or burglar alarm on the web for all to see and tinker with. Best of luck with that.

There are a couple of ways to usefully configure an Etherio board on a busy subnet. Because of the board's timing restriction, where it gets confused if packets arrive too quickly in succession, it's best to make one process on one computer act as a proxy for anybody else who wants to talk to the board. So you really want to write something like an Etherio driver, but it doesn't need to be written in C and it doesn't need to

2.3 Fun with Network Protocols

Let me make a quick plug for the Ethereal Network Analyzer, an incredibly useful tool that I used to debug network transactions between my Linux box and the Etherio board. It's easy to use, it's free, it's available for Windows and Linux and other UNIX flavors, and it will bring you long life and inner peace. Download it, use it, love it. Using it taught me an enormous amount about what's flowing over my network cables.

Network protocols come in layers. The lowest layer is Ethernet, which is the hardware you bought at the computer store. Ethernet packets have hardware (or "MAC") addresses for the source and destination computers. MAC addresses are six bytes. Ethernet headers specify a type number for the type of packet they are carrying. If it's an IP packet, the number is 0x800. The Ethernet packet acts as a wrapper for the IP packet.

The next layer is IP, or "Internet protocol". IP assigns addresses to every computer on your network. IP addresses are four-byte numbers like "192.168.1.150". IP allows packets to be routed from one subnet to another. (A subnet is a local collection of computers that occupy a shared contiguous IP address space.) To move packets around, and to connect your subnet to your Internet service provider, a router must translate IP addresses as packets cross subnet boundaries.

IP packets are wrappers for the next protocol layer. Frequently this layer is TCP, or "transfer control protocol". IP doesn't guarantee that a packet will arrive for certain. TCP adds message IDs, sequence numbers, and checksums to make sure that all the packets in a message are received, and are reconstructed into the correct order at the receiver. This means that TCP can reliably carry large messages over unreliable networks. That's why we use TCP for shipping HTML pages and other large files.

UDP, or "user datagram protocol", is a thin layer that occupies the same level in the protocol hierarchy as TCP. It doesn't provide TCP's reliability mechanisms, but it fits much more easily into a very small microcontroller like the one on the Etherio board.

TCP and UDP both add the concept of "ports" which represent distinct communication channels used simultaneously by different processes.

- `http://www.networksorcery.com/enp/protocol/ip.htm`
- `http://www.networksorcery.com/enp/protocol/tcp.htm`
- `http://www.networksorcery.com/enp/protocol/udp.htm`
- `http://www.networksorcery.com/enp/protocol/icmp.htm`
- `http://www.wikipedia.com/wiki/Ethernet`
- `http://www.wikipedia.com/wiki/ICMP`
- `http://www.wikipedia.com/wiki/Internet_protocol`
- `http://www.wikipedia.com/wiki/Transmission_Control_Protocol`

- http://www.wikipedia.com/wiki/User_Datagram_Protocol

The whole idea here is to talk about how UDP is unreliable, and so the user has to write code that does the timeout-and-retransmit trick that appears in my Python code. Try to get to that point a bit more succinctly.

Also point out that because the board responds to every command packet, and because UDP uses IP addresses and ports to keep each host process interaction distinct, it's possible for each host process to guarantee its own integrity. Another thing is that if the board receives the same command twice in a row (which can result from UDP losing the first response), the board's outputs don't change state.

2.4 Python class for talking to board

```
class Board:

    def __init__(self, ipaddr='192.168.1.150', port=5000):
        self.ipaddr = ipaddr
        self.port = port
        self.socket = sock = \
            socket.socket(socket.AF_INET,
                          socket.SOCK_DGRAM)
        sock.connect((ipaddr, port))
        sock.setblocking(0)
        self.itersPerMsec = initial = 200
        tries = 20
        tt, ms = time.time, self.msec
        t0 = tt()
        for i in range(tries):
            ms()
        msTime = (tt() - t0) / tries
        self.itersPerMsec = int((0.001 / msTime) *
                                initial)

    def __getattr__(self, key):
        for (keyval, cmd) in (("A", 0x10),
                              ("B", 0x11),
                              ("C", 0x12),
                              ("D", 0x13),
                              ("E", 0x14),
                              ("F", 0x15),
                              ("G", 0x16)):
            if key == keyval:
                return self.talk((cmd,))
        return self.__dict__[key]

    def __setattr__(self, key, value):
        for (keyval, cmd) in (("A", 0x00),
                              ("B", 0x01),
                              ("C", 0x02),
                              ("D", 0x03),
                              ("E", 0x04),
```

```
                            ("F", 0x05),
                            ("G", 0x06),
                            ("DDR0", 0x08),
                            ("DDR1", 0x09),
                            ("LED", 0xff)):
            if key == keyval:
                return self.talk((cmd, value))
        self.__dict__[key] = value

    def msec(self):
        for i in range(self.itersPerMsec):
            dummy = 12345L * 98765L

    # Flash the LED, so we can tell the board is alive,
    # or distinguish a particular board if we're using
    # many of them at once.
    def flash(self):
        self.LED = 1
        time.sleep(0.01)
        self.LED = 0
        time.sleep(0.5)

    # Send a command and expect a response. If
    # there's no response for 0.25 seconds,
    # retransmit. If this fails ten times, throw
    # an exception.
    def talk(self, cmd):
        tx, rx = self.socket.send, self.socket.recv
        tt = time.time
        self.msec()
        def tochar(x):
            return chr(x & 0xFF)
        if type(cmd) != types.StringType:
            cmd = string.join(map(tochar, cmd), "")
        for i in range(10):
            tx(cmd)
            tryUntil = tt() + 0.25
            while tt() < tryUntil:
                try:
                    return ord(rx(20))
                except socket.error:
                    pass
        raise IOError, "Etherio board not communicating"
```

2.5 C code to communicate with the Etherio board

```
#define ETHERIO_ADDR  "192.168.1.150"
#define ETHERIO_PORT  5000

typedef enum { A=0, B=1, C=2, D=3, E=4, F=5, G=6, DDR0=8, DDR1=9 } ioport;
typedef enum { READ, WRITE, LED_ON, LED_OFF } operation;
```

```
int value;

unsigned char interact_with_board(void)
{
        int i, j;
        int sockfd, cmdlen;
        struct sockaddr_in dest;
#define MAXBUF 5
        char txbuf[MAXBUF], rxbuf[MAXBUF];

        /* Open socket */
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
                perror("Socket");
                exit(errno);
        }

        /* Initialize server address/port struct */
        bzero(&dest, sizeof(dest));
        dest.sin_family = AF_INET;
        dest.sin_port = htons(ETHERIO_PORT);
        if (inet_aton(ETHERIO_ADDR, &dest.sin_addr.s_addr) == 0) {
                perror(ETHERIO_ADDR);
                exit(errno);
        }

        /* Connect to server */
        if (connect(sockfd, (struct sockaddr*)&dest,
                    sizeof(dest)) != 0) {
                perror("Connect ");
                exit(errno);
        }

        /* Make the socket non-blocking */
        fcntl(sockfd, F_SETFL, O_NONBLOCK);

        switch (operation) {
        case READ:
                txbuf[0] = 0x10 + ioport;
                cmdlen = 1;
                break;
        case WRITE:
                txbuf[0] = ioport;
                txbuf[1] = value;
                cmdlen = 2;
                break;
        case LED_ON:
                txbuf[0] = 0xFF;
                txbuf[1] = 1;
                cmdlen = 2;
                break;
```

```
        case LED_OFF:
                txbuf[0] = 0xFF;
                txbuf[1] = 0;
                cmdlen = 2;
                break;
        }

        for (i = 0; i < 5; i++) {
                /* Send the command to the board */
                send(sockfd, txbuf, cmdlen, MSG_DONTWAIT);
                /* Get a response */
                for (j = 0; j < 1000; j++) {
                        bzero(rxbuf, MAXBUF);
                        if (recv(sockfd, rxbuf, 1, 0) >= 0) {
                                /* Clean up the socket */
                                close(sockfd);
                                return rxbuf[0];
                        }
                }

        }

        /* Board is out of communication */
        perror("Board not communicating");
        exit(1);
}


void continuous_flashing_led(void)
{
        /* flash the LED */
        while (1) {
                printf("Turning LED on\n");
                operation = LED_ON;
                interact_with_board();
                short_delay();
                printf("Turning LED off\n");
                operation = LED_OFF;
                interact_with_board();
                sleep(1);
        }
}

void another_typical_usage(void)
{
        operation = WRITE;
        ioport = C;
        value = 0x3B;
        interact_with_board();
        operation = READ;
```

```
        ioport = A;
        printf("Port A says 0x%02X\n", interact_with_board());
}
```

2.6 Perl function to communicate with the Etherio board

```
sub talkToBoard
{
    use Socket;
    use Fcntl;
    my ($MSG, $MAXLEN, $ipaddr, $portaddr, $i, $j, $k);
    socket(SOCKET, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
        or die "socket: $!";
    fcntl(SOCKET, F_SETFL, O_NONBLOCK)
        or die "can't set non blocking: $!";
    $MAXLEN = 3;
    $ipaddr = inet_aton("192.168.1.150");
    $portaddr = sockaddr_in(5000, $ipaddr);
    for ($i = 0; $i < 5; $i++) {
        send(SOCKET, $_[0], 0, $portaddr) == length($_[0])
            or die "cannot send";
        for ($j = 0; $j < 1000; $j++) {
            $MSG = "\000\000\000";
            if (recv(SOCKET, $MSG, $MAXLEN, 0)) {
                return ord($MSG);
            }
        }
    }
    die "Board not communicating";
}

# Tell what's happening on port A
printf "0x%02X\n", &talkToBoard("\x10");

# Blink the LED too fast to be seen
while (1) {
    &talkToBoard("\xFF\x01");
    &talkToBoard("\xFF\x00");
}
```

3 Reprogramming the PIC and the CPLD

The PIC is reprogrammed using gputils on a Linux box, and then using Fred Eady's Windows program to ship it over the 10-pin ribbon cable.

The CPLD is reprogrammed by using Atmel's version of WinCUPL to design some logic, and then using ATMISP with the Atmel programmer to ship the JEDEC file down the 10-pin ribbon cable.

It would be schweet to put the Atmel programming hardware on my little programmer board. It would be nice to write versions of all this stuff that work under Linux entirely.

If I really intend for people to be able to do this, I should offer a few more examples, and throw them on the CDROM.

3.1 I/O connections for the CPLD

The pin-out for the CPLD will be needed for any reprogrammming. The programming tools are WinCUPL and ATMISP. The ATF15xx-DK CPLD Development Kit is available at Digikey, on page 233 of their current catalog.

```
/* PIC Data Bus */
Pin [29,30,28,27,21,20,19,17] = [SD0..SD7];

/* PIC Address Bus */
Pin [22,23,24,25] = [SA0..SA3];

Pin 16 = PicRD;  /* active-low read, write strobes */
Pin 14 = PicWR;

/* Four eight-bit I/O ports on JP1 connector */
Pin [75,72,71,70,69,68,67,65] = [PortA0..PortA7];
Pin [64,63,61,60,58,57,56,55] = [PortB0..PortB7];
Pin [54,53,52,50,48,49,47,46] = [PortC0..PortC7];
Pin [45,44,42,40,37,36,35,33] = [PortD0..PortD7];

/* Three eight-bit I/O ports in the prototyping area */
Pin [7,6,5,2,8,9,10,12] =        [PortE0..PortE7];
Pin [100,99,98,97,96,94,93,92] = [PortF0..PortF7];
Pin [84,83,81,80,79,78,77,76] =  [PortG0..PortG7];
```

4 Future projects based on the Etherio board

4.1 Logic analyzer

The near-term configuration of the logic analyzer is as a 8Kx16 FIFO connecting the GPIOs of two Etherio boards. The board (Board B) at the FIFO inputs is the device under test, and the board (Board A) at the outputs is drawing info back into the laptop.

This configuration is intended to help me debug DHCP software. A subroutine writes two numbers into the FIFO, th program counter value when it was called (14 bits) and the contents of the W register (8 bits).

The board under test needs to be easily reprogrammable. The circuitry for doing this is described on page 136 of the datasheet, and the 39025e.pdf datasheet.

Dedicate four lines of Board B to reprogramming Board A's micro.

I can now do line numbers. I'll be able to program the board from a Linux box. I only need the Windows laptop to program the CPLD, and to do PCB layout.

I've got the thing built, using FIFO chips. I frequently, however, get fifo-full exceptions when they shouldn't be occurring. This could be because of my cheesy diode logic, which I should replace with real OR gates.

4.2 Logic analyzer revisited

I used these fancy expensive FIFO chips and they're more grief than they're worth. I could have reprogrammed one CPLD to be a RAM controller and accomplished the same thing a lot cheaper

and more reliably.

Here's how it can work. The RAM is a vanilla static RAM, maybe a 62256 for the current application and a CY7C1041B (15 nsec, 256Kx16) in the longer term. The expensive part of the FIFOs is the dual-port-ness of the RAM, but I don't need that because my data collection time and my readback time don't coincide. The analyzer's CPLD has an address counter and whatever control circuitry is needed to perform the acquisition. The same CPLD controls all the control lines to the RAM, and controls the buffer that feeds data into the RAM.

The write buffers and read buffers are all conceptually 74244s. The read buffers allow the PIC to read the RAM one byte at a time, no matter how wide it is. Presumably the entire width of the RAM is written simultaneously.

Index