Design of a Microcontroller-Based Ethernet Messaging Device

David Clausen, Stanford University
December 13, 2000

Abstract: I describe the design of a microcontroller-based hardware device which has a standard RJ45 Ethernet port and an LCD display. The device will receive and decode UDP/IP datagrams, and display the contents of those datagrams as text on the LCD display.

1. Introduction
1.1 Description
This report is offered as documentation of my final project in EE281, "Embedded System Design Laboratory", in the Fall quarter of 2000 at Stanford University.

For my final project, I wanted to use an AVR AT90S8515 microcontroller to communicate over a standard Ethernet network. As a demonstration application, I decided to make a device which could receive UDP datagrams on an Ethernet interface, and display the contents on an LCD display. I also wrote a small perl script which prompts the user for text input, and sends that text to the device as a properly-formatted UDP datagram. This script can be run from a Windows or Unix computer.

For my in-class demonstration, I had a trivial network consisting of a laptop Windows 98 computer, the microcontroller device, and a 10-base-T crossover cable connecting the two. However, since the device communicates using standard IP, it should have no problems communicating over the Internet at large.

1.2 Photographs

Figure 1: Device POST shows NIC's MAC address and pre-programmed IP address

Figure 2: Compose a message in Win98, and send it via UDP over the Ethernet...
2. High Level Design
The project uses a generic NE2000-compatible ISA Ethernet card which I bought at a local electronics store. I have connected the card in such a way that it can be controlled by the AVR using 16 logic lines (8 bi-directional data, 5 address, 2 strobes, and 1 reset). After careful reading of numerous spec sheets, and studying the sourcecode for NE2000 drivers on other systems, I was able to write my own set of driver functions for this card in AVR assembly language.

The other main peripheral component in the system is a 4 line x 20 character LCD display. Since these displays are commonly used in microcontroller systems, it was easier to find documentation on how to control them. As with the NE2000, I ended up writing my own driver functions for this device in AVR assembly. This is done using 7 logic lines (4 data, 2 address, 1 strobe).

Sitting in between these two components is the "glue" software which decodes UDP datagrams, decodes ARP requests and sends ARP replies, and manages the state of the system.

3. Hardware Design
3.1. Ethernet Subsystem
The ISA card connector is connected to PORTA and PORTC of the AVR microcontroller. PORTA is used as an 8-bit bi-directional data bus between the Ethernet card and the AVR (D0 - D7 of the ISA spec). The low 5 lines of PORTC are used as address lines (always controlled by the AVR) (ISA A0 - A5). The next two lines are Read and Write strobes (active low, always controlled by the AVR) (ISA -IOR and -IOW). The last line signals a hardware reset to the ethernet card (active high, always controlled by the AVR) (ISA RESET).
The ISA connector has several pins which I have permanently connected either to ground or Vcc. The obvious ones are the power and ground feeds for the card. Also, -SMEMR and -SMEMW are permanently set high (these are active-low strobes for ISA access modes which I don't use). A5-A7, A10-A19, and AEN are permanently tied to ground. A8 and A9 are permanently set high. This combination effectively causes the low 5 address bits (A0 - A4) coming from the AVR to be added to a permanent offset of 0x300, permitting the AVR to address memory locations 0x300 - 0x320 using only 5 address lines. Conveniently, this is exactly the address range used by the NE2000 Ethernet card.

Many of the unused pins on the ISA connector are left unconnected. Among these are several other power lines (+12v, -5v, -12v), many lines for DMA access, IRQ lines, high-order data lines (for 16-bit data transfers), and a few others. One helpful fact was that the designers of the SMC Ethernet card were kind enough to delete the connection pads for the ISA lines which the card ignored. This was a wonderfully helpful indicator of which parts of the ISA spec I needed to worry about in my design, and which I could ignore. The only lines which are used by the NIC, but are unconnected in my design are several of the IRQ lines. Since the card works fine in polling mode, there seems to be no danger in ignoring IRQs set by the card.

The NE2000 NIC itself is used without modification. In fact, I have tried several cards from different manufacturers, and they all seem to work fine. The SMC EZNET-ISA drew the least amount of power, so that was the one I usually used in the system.

3.2. LCD Subsystem
The LCD display subsystem was built on its own semi-autonomous daughterboard. It has a 10-pin header which can be connected directly to one of the PORTx connectors on the STK-200 development board using a ribbon cable. I used it this way when I was writing the software to control the LCD. However, since this project does not use the STK-200, I hardwired the LCD daughterboard to PORTD of the embedded AVR. It also requires power and ground, which are connected to the 7805 power circuit. The board has a variable resistor which is used to adjust the contrast of the display, and also a LED power indicator.

3.3 Power Subsystem
The power circuit is based on a 7805 5-volt regulator. There is a 200 microfarad capacitor across the inputs, and smaller noise-absorbing capacitors near each component.

3.4 AVR Oscillator
The clock for the AVR is provided by a canned 4MHz oscillator from Pletronics.

4. Software Design
4.1. Ethernet Driver
The most challenging part of this project was designing the software to communicate with the Ethernet card. Unfortunately I was only able to find partial specifications for the
card on the internet, and so I had to infer some information about the card's operation by reading sourcecode of drivers for other systems. In addition, I had to learn about and properly control the basic mechanics of the ISA bus - timing issues, access modes, etc.

In summary, the AVR communicates with the card through a set of 32 memory locations. The AVR sets the address lines A0 - A5, then lowers and raises either the read or write strobe, causing a read or write of 8 bits on the data bus. If it was a write, the AVR drives D0 - D7 prior to the write strobe being lowered. If it was a read, the AVR reads values of D0 - D7 just before the read strobe is raised.

The lower half of the addressable memory locations (0x300 - 0x30f) are referred to as "registers" in the Ethernet card's controlling chip. These are documented in the spec sheet for the "National Semiconductor DP8390D Network Interface Controller", which apparently was the chip used in the original NE2000 series of Ethernet cards (this interface has since been reproduced by many card and chipset manufacturers, hence the long line of "NE2000 compatible" or "NE2000 clone" cards like the one I used for this project). Reading from and writing to these registers permits the AVR to configure the card's options, allocate space in the card's onboard RAM, initiate transfers of data between the card and the AVR, etc.

The upper half of the addressable memory locations (0x310 - 0x31f) are used for other purposes. After looking through the sourcecode for a couple of open-source drivers, I was able to infer that 0x310 is the general-purpose data-transfer location.

The card has an internal RAM buffer of 16KB. In order to transfer data to or from the card's onboard RAM, the AVR writes a pointer into one of the card's registers, then writes a memory transfer command to a command register, and then begins repeatedly reading from or writing to location 0x310. After each read or write, the card increments its internal pointer, so subsequent read or write operations to 0x310 actually interact with the next sequential location in the card's internal RAM. Most of this memory is used as a ring buffer to hold received packets, and since both the AVR and the NIC are accessing this memory, you must be careful to keep the set of pointers describing the state of the buffer up to date. Otherwise contention between the two could cause corruption of the data, or the buffer could overflow causing corruption or a card crash.

Another segment of memory is used to build and store packets for transmitting. Again, care is needed to make sure that the AVR and the NIC don't interact badly when accessing this memory.

The only other location which seems to have a significance is 0x31f, which can be written to initiate a software reset (although I do not make use of this in my project -- I use a hardware reset line to force a reset).

There are many other details surrounding the use of the Ethernet card. Refer to my sourcecode and the relevant spec sheets for more information. One important general point is that before using the Ethernet card with a project like this, you have to install it in
a real x86 computer, boot to DOS, and configure the card's EEPROM using the manufacturer's configuration program (which comes on a floppy disk with the card). The card must be put into non-plug-n-play mode, and you need to fix the I/O base address to 0x300 (or whatever value matches your hard-wired address offset).

4.2. Ethernet/802.3, UDP/IP stack
In addition to dealing with the NIC, the software must also transmit and receive properly-formatted data packets in order to participate on the network. Since the NE2000 NIC only provides raw bytes from the packets, it is up to my software to decode the Ethernet frame headers, as well as ARP and IP packet formats.

The device has both an Ethernet MAC address, and an IPv4 address. The MAC address is read out of the NIC's EEPROM during initialization of the card. Oddly, the device driver must read the MAC address from the card's EEPROM, and then write it back to several of the card's registers as part of the bootup sequence. The IP address is simply hardcoded into my software. For the testing and demonstration, I used an address in a private block: 192.168.0.14.

The first protocol the device must understand in order to participate on the network is ARP. The ARP protocol is used to discover the MAC address of a device on the network, when only the IP address is known to the sender. Since the AVR has a tiny amount of onboard RAM, the software decodes packets in a pipeline fashion -- reading one byte at a time, and jumping to the appropriate code as soon as a decision can be made about the type of packet, where the code continues to read out bytes from the NIC. The ARP-request handling code immediately sends a properly formatted ARP reply packet.

Additionally, the device receives and decodes UDP messages arriving on port 987. The data portion of these packets are sent byte-by-byte to the LCD device, as described below.

All other incoming packets are simply discarded.

4.3. LCD driver
The LCD display has an onboard controller which is compatible with the industry-standard Hitachi interface. From my research, it seems that most LCD displays use this interface. The controller uses either 4 or 8 lines as a bi-directional communication bus. I use the 4-bit mode to reduce the number of pins on the AVR which are dedicated to driving the LCD. There are 3 additional lines: "RS: register select", "RW: read/write", and "E: enable". In short, RS indicates whether the bus is being used to transmit a command or data. RW indicates whether the AVR is reading from or writing to the LCD controller. E is a strobe which is used to indicate when the values on the other pins are known to be stable. All 3 control lines are always set by the AVR. The 4 or 8 bus lines can be set by either the AVR or the LCD, depending on the operation. In "4-bit" mode, data is always sent in two 4-byte nybbles, high byte first.
When the system powers up, I follow the initial startup sequence from the part's spec sheet. When a message arrives from the Ethernet, I issue a command to position the LCD’s cursor at the top-left position, print 20 characters, reposition the cursor to the beginning of the next line, print 20 characters, and so on. Each message is presumed to be exactly 80 characters in length. Longer messages are truncated. Shorter messages would cause random values to be written to the display.

The spec sheet states that the Hitachi controller is rather slow, and you must be careful not send a new command or data byte until it has finished processing the last message. This can be done either by waiting for a certain number of microseconds between each transfer, or by reading a busy-flag value from the controller to find out when it is ready. I took the easy way out, and simply delay for the prescribed amount of time after each transfer.

5. Results
I am very pleased with the results of my efforts. The device does what it is supposed to do, and I am not aware of any bugs.

That being said, this is clearly a "first cut" of both the hardware and software. It is apparently possible to memory-map the ISA bus into some portion of the AVR's external memory locations. This way, you can use the AVR's built in address logic, strobes, etc. to communicate with the ISA device, which would simplify the software and speed up access to the NIC. Also, this would permit you to share the memory bus with other devices or an external SRAM chip. In the next revision, I would like to modify my hardware to support this mode of operation.

My software is also sub-optimal. I was very generous with my timing delays, and not terribly concerned with memory use or code beautification. My only concern in this revision was getting everything to work before the deadline. Now that I know how to do it, I think it would be best to simply start over and write new software from scratch for the next revision.

It would also be nice to have more network functionality available. A full TCP/IP stack is probably out of the question, but a minimal implementation, along with a basic socket library is probably quite reasonable.

The last thing I would really like to do is to use this Ethernet subsystem as a component in a more interesting device. Displaying text messages on an LCD is a good demonstration, but in practical terms it is a fairly useless device. One can imagine many other far more interesting Internet-Enabled devices which could be built with these components.
6. Parts List
* AVR 8515 microcontroller (Atmel AT90S8515-8PC-0033)
* NE2000-compatible ISA bus Ethernet card (SMC EZNET ISA)
* 4 line x 20 column LCD display (AND 721GST)
* 4MHz oscillator (Pletronics P1100-HC from Jameco)
* 7805 5v linear voltage regulator (Radio Shack 276-1770A)
* 220 microfarad electrolytic capacitor (Radio Shack 272-1029)
* 0.1 microfarad capacitors (Radio Shack 272-109A)
* ISA card edge connector (Jameco 42091)
* Perfboard (Radio Shack 276-1396A)
* Wire-wrap Sockets
* 2.5K-ohm variable resistor for LCD contrast
* LED and 1K-ohm resistor to indicate power is active
* Power switch
* Power connector
* 12v DC "brick" power transformer
### 7. Schematic

#### 7.1 ISA connector

<table>
<thead>
<tr>
<th>Gnd</th>
<th>Gnd</th>
<th>-IOCHCK</th>
<th>+5v</th>
<th>AVR-PA7</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVR-PC7</td>
<td>Gnd</td>
<td>RESET</td>
<td>D7</td>
<td>AVR-PA7</td>
</tr>
<tr>
<td>Vcc</td>
<td>+5v</td>
<td>D6</td>
<td>AVR-PA6</td>
<td></td>
</tr>
<tr>
<td>Vcc</td>
<td>IRQ9</td>
<td>D5</td>
<td>AVR-PA5</td>
<td></td>
</tr>
<tr>
<td>Vcc</td>
<td>-5v</td>
<td>D4</td>
<td>AVR-PA4</td>
<td></td>
</tr>
<tr>
<td>Vcc</td>
<td>DREQ2</td>
<td>D3</td>
<td>AVR-PA3</td>
<td></td>
</tr>
<tr>
<td>Vcc</td>
<td>-12v</td>
<td>D2</td>
<td>AVR-PA2</td>
<td></td>
</tr>
<tr>
<td>Vcc</td>
<td>-0WS</td>
<td>D1</td>
<td>AVR-PA2</td>
<td></td>
</tr>
<tr>
<td>Gnd</td>
<td>Gnd</td>
<td>D0</td>
<td>AVR-PA1</td>
<td></td>
</tr>
<tr>
<td>+12v</td>
<td>IOCHRDY</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Vcc | -SMEMW | AEN | Gnd |
| Vcc | -SMEMR | A19 | Gnd |
| AVRP-PC6 | -IOW | A18 | Gnd |
| AVRP-PC5 | -IOR | A17 | Gnd |
| AVRP-PC5 | -DACK3 | A16 | Gnd |
| AVRP-PC5 | DRQ3 | A15 | Gnd |
| AVRP-PC5 | -DACK1 | A14 | Gnd |
| AVRP-PC5 | DRQ1 | A13 | Gnd |
| AVRP-PC5 | -REFSH | A12 | Gnd |
| AVRP-PC5 | CLK | A11 | Gnd |
| AVRP-PC5 | IRQ7 | A10 | Gnd |
| AVRP-PC5 | IRQ6 | A9 | Gnd |
| AVRP-PC5 | IRQ5 | A8 | Vcc |
| AVRP-PC5 | IRQ4 | A7 | Vcc |
| AVRP-PC5 | IRQ3 | A6 | Gnd |
| AVRP-PC5 | -DACK2 | A5 | Gnd |
| AVRP-PC5 | TC | A4 | AVR-PC4 |
| AVRP-PC5 | BALE | A3 | AVR-PC3 |
| AVRP-PC5 | +5v | A2 | AVR-PC2 |
| AVRP-PC5 | Osc | A1 | AVR-PC1 |
| Gnd | Gnd | A0 | AVR-PC0 |

---

No Connection to this part of the ISA connector

---
### 7.2 LCD

**Contrast-control potentiometer:**

<table>
<thead>
<tr>
<th></th>
<th>Vcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Gnd</td>
<td>Gnd</td>
</tr>
<tr>
<td>2: Vdd</td>
<td>Vcc</td>
</tr>
<tr>
<td>3: Vo</td>
<td>-----------</td>
</tr>
<tr>
<td>4: RS</td>
<td>AVR-PD2</td>
</tr>
<tr>
<td>5: RW</td>
<td>AVR-PD3</td>
</tr>
<tr>
<td>6: E</td>
<td>AVR-PD1</td>
</tr>
<tr>
<td>7: DB0</td>
<td></td>
</tr>
<tr>
<td>8: DB1</td>
<td></td>
</tr>
<tr>
<td>9: DB2</td>
<td></td>
</tr>
<tr>
<td>10: DB3</td>
<td></td>
</tr>
<tr>
<td>11: DB4</td>
<td>AVR-PD4</td>
</tr>
<tr>
<td>12: DB5</td>
<td>AVR-PD5</td>
</tr>
<tr>
<td>13: DB6</td>
<td>AVR-PD6</td>
</tr>
<tr>
<td>14: DB7</td>
<td>AVR-PD7</td>
</tr>
</tbody>
</table>

### 7.2 AVR + Oscillator

**AVR:**

<table>
<thead>
<tr>
<th></th>
<th>Vcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB0</td>
<td>Vcc</td>
</tr>
<tr>
<td>PB1</td>
<td>PA0</td>
</tr>
<tr>
<td>PB2</td>
<td>PA1</td>
</tr>
<tr>
<td>PB3</td>
<td>PA2</td>
</tr>
<tr>
<td>PB4</td>
<td>PA3</td>
</tr>
<tr>
<td>PB5</td>
<td>PA4</td>
</tr>
<tr>
<td>PB6</td>
<td>PA5</td>
</tr>
<tr>
<td>PB7</td>
<td>PA6</td>
</tr>
<tr>
<td>-RESET</td>
<td>PA7</td>
</tr>
<tr>
<td>&lt; PD0</td>
<td>ICP</td>
</tr>
<tr>
<td>&lt; PD1</td>
<td>ALE</td>
</tr>
<tr>
<td>LCD &lt;</td>
<td>PD2</td>
</tr>
<tr>
<td>LCD &lt;</td>
<td>PC7</td>
</tr>
<tr>
<td>LCD &lt;</td>
<td>PC6</td>
</tr>
<tr>
<td>LCD &lt;</td>
<td>PC5</td>
</tr>
<tr>
<td>LCD &lt;</td>
<td>PC4</td>
</tr>
<tr>
<td>LCD &lt;</td>
<td>PC3</td>
</tr>
<tr>
<td>XTAL2</td>
<td>PC2</td>
</tr>
<tr>
<td>Gnd</td>
<td>GND</td>
</tr>
<tr>
<td>Gnd</td>
<td>PC0</td>
</tr>
</tbody>
</table>

**Oscillator**

<table>
<thead>
<tr>
<th></th>
<th>Vcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: N/C</td>
<td></td>
</tr>
<tr>
<td>14: Vcc</td>
<td>Vcc</td>
</tr>
</tbody>
</table>

---
8. Code

8.1. udpsend.pl

#!/usr/local/bin/perl -w

use strict;
use Socket;

my $paddr;

sub udpinit();
sub gui();
sub udpsend($);

udpinit();
while(1) {
  gui();
}
exit(0);

sub udpinit() {
  my $proto = getprotobyname('udp');
  socket(SOCKET, PF_INET, SOCK_DGRAM, $proto);
  my $remote = "192.168.0.14";
  my $port = 987;
  my $iaddr = gethostbyname($remote);
  $paddr = sockaddr_in($port, $iaddr);
}

sub gui() {
  $| = 1;
  print "enter a 4-line message (20 chars/line max):
";
  my $pad = ' ' x 20;
  my @lines = ();
  my $line;
  print " >....................<\n";
  while (@lines<4) {
    print scalar(@lines)+1, " >";
    $line = <STDIN>;
    chomp($line);
    $line .= $pad;
    $line = substr($line, 0, 20);
    push(@lines, $line);
  }
  print "\n\n Your message will look like this:\n";
  print " |----------------------|\n";
  foreach my $line (@lines) {
    print " |", $line," |\n";
  }
  print " |----------------------|\n";
  my $message = join('', @lines);
  print "\n\n Sending...\n";
  my $bytecount = udpsend($message);
  print $bytecount, " bytes sent.\n\n";
}

sub udpsend($) {
  my $message = $_[0];
  my $bytecount = send(SOCKET, $message, 0, $paddr);
  return($bytecount);
}
8.2. avr-ne2k.asm

;-------------------------------------------------------------------
; dave clausen
; ee281
; avr-ne2k.asm
; 11/26/2000
;-------------------------------------------------------------------

; setup
;-------------------------------------------------------------------

.nolist
.include "8515def.inc"
.list

; register definitions
;-------------------------------------------------------------------

.def ZERO = r2
.def ONE = r3
.def FF = r4
.def LED = r5

; constants
;-------------------------------------------------------------------

; *** LCD ***
; refer to AND "intelligent alphanumeric application notes" PDF document
; for a description of the LCD command API. (can be downloaded from
; http://www.purdyelectronics.com/).
;
; the brief summary: there are two modes for accessing the LCD:
; 4-bit and 8-bit. in addition, you need three extra control lines,
; so really these modes should be called 7-bit and 11-bit.
; anyway, to conserve pins on my AVR, i am using the slightly more
; complex 4-bit interface. the two modes are similar - the main difference
; is that in 4-bit mode, you send data in two chunks - the 4 high bits,
; followed immediately afterwards by the 4 low bits. the initialization
; sequence for 4-bit mode is a little different also.
;
; regardless of which mode you use, you follow the same rules:
;
; the RS line tells the LCD which register (instruction or data) you wish
; to access.
;
; the RW bit tells the LCD whether you are reading or writing.
;
; the 4 (or 8) DB lines contain the binary data being written to, or read
; from the register.
;
; the E line goes through a state change, indicating that the data on the
; other lines is valid and holding steady. to make life easy, i always
; set the E line low, then high, then low, while holding all of the other
; lines constant. (see the API doc for a more efficient technique).
;
; I have wired my controller so that it can be plugged into one of the
; 10-pin headers on the STK-200 development board. Pin 0 is not
; connected to anything (maybe later i will add an LED or a button or
; something). Pin 1 is tied to the "E" line on the LCD. Pin 2 is tied
; to the RS bit. Pin 3 is tied to the RW bit, and pins 4-7 are tied
; to DB4-DB7 on the LCD.
;
.equ LED_BIT = 0b00000001 ; bit used for the red LED
.equ LCD_BITS = 0b11111110 ; bits used by the LCD interface
.equ ENABLE_BIT = 0b00000010 ; the "E" bit (enable)
.equ RS_BIT = 0b00000100 ; the "RS" bit (register select)
.equ RW_BIT = 0b00001000 ; the "R/W" bit (read or write select)
.equ DB_BITS = 0b11110000 ; the 4 data bits
.equ NOT_LCD_BITS = 0b00000001
.equ NOT_ENABLE_BIT = 0b11111101
.equ NOT_RS_BIT = 0b11111011
.equ NOT_RW_BIT = 0b11110111
.equ NOT_DB_BITS = 0b00001111

; which port is the LCD using?
equ LCD_PORT = PORTD
.equ LCD_PORT_DDR = DDRD

; *** NE2000 ethernet card ***
; internal command registers for the DP8390 chip.
; definitions are taken from National Semiconductor document
; DP8390D.pdf: "DP8390D/NS32490D NIC Network Interface Controller"
; (this is the datasheet for the DP8390D chip, which was used in
; the original NE2000 cards).

; hardware has the base address hardwired. the software
; only needs to set the lowest 5 bits
.equ NE2K_BASE = 0

; internal registers (see page 17 of DP8390D.pdf).
; these are defined here for completeness. i doubt i will need to use
; all of them in the code.

; Page 0 readable registers
.equ NE2K_CR = NE2K_BASE + 0 ; command register
.equ NE2K_CLDA0 = NE2K_BASE + 1 ; current local DMA address 0
.equ NE2K_CLDA1 = NE2K_BASE + 2 ; local dma 1
.equ NE2K_BNRY = NE2K_BASE + 3 ; boundary pointer
.equ NE2K_TSR = NE2K_BASE + 4 ; transmit status register
.equ NE2K_NCR = NE2K_BASE + 5 ; number of collisions register
.equ NE2K_FIFO = NE2K_BASE + 6 ; FIFO
.equ NE2K_ISR = NE2K_BASE + 7 ; interrupt status register
.equ NE2K_CRDAO = NE2K_BASE + 8 ; current remote DMA address 0
.equ NE2K_CRDA1 = NE2K_BASE + 9 ; remote DMA 1
.equ NE2K_RESV1 = NE2K_BASE + 10 ; reserved
.equ NE2K_RESV2 = NE2K_BASE + 11 ; reserved
.equ NE2K_RSR = NE2K_BASE + 12 ; receive status register
.equ NE2K_CNTR0 = NE2K_BASE + 13 ; tally counter 0 (frame alignment
; errors)
.equ NE2K_CNTR1 = NE2K_BASE + 14 ; tally counter 1 (CRC errors)
.equ NE2K_CNTR2 = NE2K_BASE + 15 ; tally counter 2 (missed packet errors)

; Page 0 writable registers
; +0: CR is read/write
.equ NE2K_PSTART = NE2K_BASE + 1 ; page start register
.equ NE2K_PSTOP = NE2K_BASE + 2 ; page stop register
; +3: BNR is read/write
.equ NE2K_TPSR = NE2K_BASE + 4 ; transmit page start address
.equ NE2K_TBCR0 = NE2K_BASE + 5 ; transmit byte count register 0
.equ NE2K_TBCR1 = NE2K_BASE + 6 ; transmit byte count register 1
; +7: ISR is read/write
.equ NE2K_RSAR0 = NE2K_BASE + 8 ; remote start address register 0
.equ NE2K_RSAR1 = NE2K_BASE + 9 ; remote start address register 1
.equ NE2K_RBCR0 = NE2K_BASE + 10 ; remote byte count register 0
.equ NE2K_RBCR1 = NE2K_BASE + 11 ; remote byte count register 1
.equ NE2K_RCR = NE2K_BASE + 12 ; receive configuration register
.equ NE2K_TCR = NE2K_BASE + 13 ; transmit configuration register
.equ NE2K_DCR = NE2K_BASE + 14 ; data configuration register
.equ NE2K_IMR = NE2K_BASE + 15 ; interrupt mask register
; Page 1 registers
; CR spans pages 0,1, and 2
.equ NE2K_PAR0 = NE2K_BASE + 1 ; physical address register 0
.equ NE2K_PAR1 = NE2K_BASE + 2 ; physical address register 1
.equ NE2K_PAR2 = NE2K_BASE + 3 ; physical address register 2
.equ NE2K_PAR3 = NE2K_BASE + 4 ; physical address register 3
.equ NE2K_PAR4 = NE2K_BASE + 5 ; physical address register 4
.equ NE2K_PAR5 = NE2K_BASE + 6 ; physical address register 5
.equ NE2K_CURR = NE2K_BASE + 7 ; current page register
.equ NE2K_PAR6 = NE2K_BASE + 8 ; multicast address register 0
.equ NE2K_PAR7 = NE2K_BASE + 9 ; multicast address register 1
.equ NE2K_PAR8 = NE2K_BASE + 10; multicast address register 2
.equ NE2K_PAR9 = NE2K_BASE + 11; multicast address register 3
.equ NE2K_PAR10 = NE2K_BASE + 12; multicast address register 4
.equ NE2K_PAR11 = NE2K_BASE + 13; multicast address register 5
.equ NE2K_PAR12 = NE2K_BASE + 14; multicast address register 6
.equ NE2K_PAR13 = NE2K_BASE + 15; multicast address register 7

; Page 2 registers
; ... not implemented ...

; Page 3 registers
; ... not implemented ...

; other special locations
.equ NE2K_DATAPORT = NE2K_BASE + 0x10
.equ NE2K_RESET = NE2K_BASE + 0x1f
; .equ NE2K_IO_EXTENT = NE2K_BASE + 0x20

; bits in various registers
.equ NE2K_CR_STOP = 0x01 ; stop card
.equ NE2K_CR_START = 0x02 ; start card
.equ NE2K_CR_TRANSMIT = 0x04 ; transmit packet
.equ NE2K_CR_DMAREAD = 0x08 ; remote DMA read
.equ NE2K_CR_DMAMWRITE = 0x10 ; remote DMA write
.equ NE2K_CR_NODMA = 0x20 ; abort/complete remote DMA
.equ NE2K_CR_PAGE0 = 0x40 ; select register page 0
.equ NE2K_CR_PAGE1 = 0x80 ; select register page 1
.equ NE2K_CR_PAGE2 = 0x20 ; select register page 2

.equ NE2K_RCR_BCAST = 0x04
.equ NE2K_RCR_MCAST = 0x08
.equ NE2K_RCR_PROMISCUOUS = 0x10
.equ NE2K_RCR_MONITOR = 0x20

.equ NE2K_DCR_BYTEDMA = 0x00
.equ NE2K_DCR_WORDDMA = 0x01
.equ NE2K_DCR_NOLPBK = 0x08
.equ NE2K_DCR_FIFO2 = 0x00
.equ NE2K_DCR_FIFO4 = 0x20
.equ NE2K_DCR_FIFO8 = 0x40
.equ NE2K_DCR_FIFO12 = 0x60

.equ NE2K_TCR_NOLPBK = 0x00
.equ NE2K_TCR_INTLPBK = 0x02
.equ NE2K_TCR_EXTLPBK = 0x04
.equ NE2K_TCR_EXTLPBK2 = 0x06

; i don't have a spec sheet on it, but it seems that the ne2000 cards have 16kb of onboard ram mapped to locations 0x4000 - 0x8000. this is used as a buffer for packets, either before transmission, or after reception. the dp8390d spec sheet describes how the chip manages the buffer space. in summary, you need to mark off a relatively small section for your transmit buffer. it seems that you can use a chunk either at the beginning or the end of the ram segment. 6 pages is the typical size. you then use the rest of the remaining space as a receive buffer. the chip treats this as a ring - in other words if it reaches the end of the space, it wraps around to the beginning and continues filling from there. you need to empty the data out fast enough, otherwise it will wrap around and hit itself in the tail. (it will detect this situation, and
; just drop incoming data until you clear out some space. There are several pointers which are used to keep track of all this. Read the datasheet for more details.

.equ NE2K_TRANSMIT_BUFFER = 0x40 ; transmit buffer from 0x4000 - 0x45ff.
; we could add a second 6-byte buffer here to do ping-pong (back-to-back) transmissions, but let's leave that for later...

.equ NE2K_START_PAGE = 0x46 ; receive buffer ring from
.equ NE2K_STOP_PAGE = 0x80 ; 0x4600-0x7fff

; port assignments

.equ NE2K_DATA_OUT = PORTA
.equ NE2K_DATA_IN = PINA
.equ NE2K_DATA_DDR = DDRA

.equ NE2K_ADDR_OUT = PORTC
.equ NE2K_ADDR_DDR = DDRC

.equ NE2K_ISSA_ADDR = 0b00011111
.equ NE2K_ISSA_IOR = 0b00100000
.equ NE2K_ISSA_IOW = 0b01000000
.equ NE2K_ISSA_RESET =0b10000000

; hardcoded IP address:

.equ NE2K_IP_OCTET_1 = 192
.equ NE2K_IP_OCTET_2 = 168
.equ NE2K_IP_OCTET_3 = 0
.equ NE2K_IP_OCTET_4 = 14

; which UDP port number should I listen on?
.equ NE2K_LISTEN_PORT = 987

;-------------------------------------------------------------------
; begin eeprom segment
;-------------------------------------------------------------------

.ESEG
.db "This is some stuff in the eeprom"
.db 0

;-------------------------------------------------------------------
; begin data segment
;-------------------------------------------------------------------

.DSEG

; allocate space in the microcontroller's onboard ram for these things
ne2k_mac_addr: .byte 6 ; my hardware ethernet address
ne2k_peer_mac_addr: .byte 6 ; my partner's hardware ethernet address
ne2k_ip_addr: .byte 4 ; my ip address
ne2k_peer_ip_addr: .byte 4 ; my partner's ip address

;-------------------------------------------------------------------
; begin code segment
;-------------------------------------------------------------------

.CSEG

; interrupt vector

rjmp RESET ; external reset
rjmp IGNORE_INTERRUPT ; external int0
rjmp IGNORE_INTERRUPT ; external int1
rjmp IGNORE_INTERRUPT ; timer2 compare match
rjmp IGNORE_INTERRUPT ; timer2 overflow
rjmp IGNORE_INTERRUPT ; timer1 capture event
rjmp IGNORE_INTERRUPT ; timer1 compare match A
rjmp IGNORE_INTERRUPT ; timer1 compare match B
rjmp IGNORE_INTERRUPT ; timer1 overflow
rjmp IGNORE_INTERRUPT ; timer0 overflow
rjmp IGNORE_INTERRUPT ; SPI serial transfer complete
rjmp IGNORE_INTERRUPT ; UART Rx complete
rjmp IGNORE_INTERRUPT ; UART data register empty
rjmp IGNORE_INTERRUPT ; UART Tx complete
rjmp IGNORE_INTERRUPT ; ADC conversion complete
rjmp IGNORE_INTERRUPT ; EEPROM ready
rjmp IGNORE_INTERRUPT ; Analog comparator

; interrupt handlers

IGNORE_INTERRUPT:
    reti

RESET:
    rjmp main_program

; main program

main_program:
    ; block interrupts
    cli

    ; reset the stack pointer
    ldi r16, low(RAMEND)
    out SPL, r16
    ldi r16, high(RAMEND)
    out SPH, r16

    ; initialize special registers
    clr ZERO
    ldi r16, 1
    mov ONE, r16
    ldi r16, $FF
    mov FF, r16
    ldi r16, LED_BIT
    mov LED, r16 ; active low (0=on, 1=off)
    ;mov LED, ZERO

    ; turn off the analog comparator to save power
    ldi r16, 0b11100111
    out PORTB, r16

    ; port B is connected to the LEDs on the STK-200
    ; development board. (active low)
    ldi r16, $FF
    out DDRB, r16

    ; intialize the LCD
    rcall initialize_lcd

    ; show countdown sequence on the STK-200 LEDs
    ldi r16, 0b10000000 ; ACD - analog compare disable
    out ACSR, r16

    ; port B is connected to the LEDs on the STK-200
    ; development board. (active low)
    ldi r16, $FF
    out DDRB, r16

    ; intialize the LCD
    rcall initialize_lcd

    ; show countdown sequence on the STK-200 LEDs
    ldi r16, 0b10000000
    out PORTB, r16
    rcall delay_1s

    ldi r16, 0b10110111
    out PORTB, r16
    rcall delay_1s

    ldi r16, 0b10110111
    out PORTB, r16
    rcall delay_1s
ldi    r16, 0b10111101
out    PORTB, r16
rcall  delay_1s

ldi    r16, 0b01111110
out    PORTB, r16
rcall  delay_1s
ldi    r16, 0b11111111
out    PORTB, r16

; initialize the LCD (again)
rcall  initialize_lcd
rcall  write_silly_string

rcall  delay_1s
rcall  ne2k_init
rcall  ne2k_establish_ip_address

rcall  lcd_write_mac_addr
rcall  lcd_write_ip_addr

main_loop:
    rcall  ne2k_read_packet
rjmp   main_loop

;-------------------------------------------------------------------
; functions
;-------------------------------------------------------------------
;
;-------------------------------------------------------------------
; *** NE2000 ***
;-------------------------------------------------------------------
;
;-------------------------------------------------------------------

ne2k_write:
; address in r16 (5 bits)
; data in r17 (one byte)
; data will be written to the ne2000 NIC
push   r18
push   r19

; set both address and data ports for output
ldi    r18, 0xff
out    NE2K_ADDR_DDR, r18
out    NE2K_DATA_DDR, r18

; set data lines
out    NE2K_DATA_OUT, r17

; set address lines, plus read/write strobes
mov    r18, r16
andi   r18, NE2K_ISA_ADDR
ori    r18, NE2K_ISA_IOR
mov    r19, r18
ori    r18, NE2K_ISA_IOW
out    NE2K_ADDR_OUT, r18  ; IOW high
nop
nop
nop
out    NE2K_ADDR_OUT, r19  ; IOW low
nop
nop
nop
out    NE2K_ADDR_OUT, r18  ; IOW high
pop    r19
pop    r18
ret
ne2k_read:
; address in r16 (5 bits)
; data read from the ne2000 NIC will be put into r17 (1 byte)

    push    r18
    push    r19

    ; set address port for output
    ldi    r18, 0xff
    out    NE2K_ADDR_DDR, r18
    ; set data port for input
    ldi    r18, 0
    out    NE2K_DATA_DDR, r18

    ; set address lines, plus read/write strobes
    mov    r18, r16
    andi   r18, NE2K_ISA_ADDR
    or    r18, NE2K_ISA_IOW
    mov    r19, r18
    ori   r18, NE2K_ISA_IOR
    out    NE2K_ADDR_OUT, r18 ; IOR high
    nop
    nop
    nop
    nop
    out    NE2K_ADDR_OUT, r19 ; IOR low
    nop
    nop
    nop
    nop
    in     r17, NE2K_DATA_IN
    out    NE2K_ADDR_OUT, r18 ; IOR high
    pop    r19
    pop    r18
    ret

ne2k_hard_reset:
; set, then clear, the ISA RESET line, forcing a hard reset of the card

    push    r18

    ; set address port for output
    ldi    r18, 0xff
    out    NE2K_ADDR_DDR, r18

    ; reset line high
    ldi    r18, NE2K_ISA_RESET | NE2K_ISA_IOR | NE2K_ISA_IOW
    out    NE2K_ADDR_OUT, r18
    rcall  delay_100ms ; is this the right delay? i have no idea,
                        ; but it works ok

    ; reset line low
    ldi    r18, NE2K_ISA_IOR | NE2K_ISA_IOW
    out    NE2K_ADDR_OUT, r18
    rcall  delay_100ms ; another arbitrary delay
    pop    r18
    ret

ne2k_soft_reset:
; untested. i saw this in someone's driver.

    push    r16
push r17
ldi r16, NE2K_RESET
ldi r16, 0xff
rcall ne2k_write
rcall delay_25ms;
pop r17
pop r16
ret

-------------------------------------------------------------------
ne2k_show_cr:
; primitive debugging feature
; read the ne2000 command register, and show it on the STK-200 portB LEDs
push r16
push r17
ldi r17, 0b10101010
out PORTB, r17
ldi r16, NE2K_CR
rcall ne2k_read
eor r17, FF
out PORTB, r17 ; display the command register
rcall delay_1s ; wait
pop r17
pop r16
ret

-------------------------------------------------------------------
ne2k_init:
; follow the initialization sequence described on page 19 of
; DP8390D.pdf (er, i mean "sort of" follow). lots of modifications,
; taken mostly from the linux driver. comments indicate interesting
; deviations in cheung's driver, the national semiconductor sample
; driver, and the linux driver,
push r16
push r17
push r18
push r30
push r31
; my step 0a: force a hardware reset on the card
rcall ne2k_hard_reset
; my step 0b: read mac address from the card's onboard eeprom
rcall ne2k_read_mac_eeprom ; read the mac address from the eeprom
; step 1: program command register for page 0
; cheung, ns 0x21
ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_STOP | NE2K_CR_NODMA
rcall ne2k_write
; cheung does a soft reset here...
; step 2: initialize data configuration register
; cheung 0x48, ns 0x58
ldi r16, NE2K_DCR
ldi r17, NE2K_DCR_BYTEDMA | NE2K_DCR_FIFO8 | NE2K_DCR_NOLPBK
rcall ne2k_write
; step 3: clear remote byte count registers
; cheung, ns 0
ldi r16, NE2K_RBCR0
ldi r17, 0
rcall ne2k_write
ldi r16, NE2K_RBCR1
ldi r17, 0
rcall ne2k_write

; step 4: initialize receive configuration register
; cheung: 0x0c, ns: 0, linux: 0x20
ldi r16, NE2K_RCR
ldi r17, NE2K_RCR_BCAST
ldi r17, NE2K_RCR_MONITOR ; disable reception for now
rcall ne2k_write

; step 5: place the NIC in loopback mode (hey - don't i also have to set
; a bit in DCR in order to go into loopback mode? hmm...)
ldi r16, NE2K_TCR
ldi r17, NE2K_TCR_INTLPBK
rcall ne2k_write

; step 5 and a half: initialize the transmit buffer start page
ldi r16, NE2K_TPSR
ldi r17, NE2K_TRANSMIT_BUFFER
rcall ne2k_write

; step 6: initialize receive buffer ring (256 byte blocks)
; cheung: start=0x40, stop=0x76 (or 0x7c?)
; ns: start=0x26, stop=0x40
; linux: 0x26/0x40 or 0x46/0x80 (NE1SM or NESM)
ldi r16, NE2K_PSTART
ldi r17, NE2K_START_PAGE
rcall ne2k_write

ldi r16, NE2K_BNRY
ldi r17, NE2K_START_PAGE
rcall ne2k_write

ldi r16, NE2K_PSTOP
ldi r17, NE2K_STOP_PAGE
rcall ne2k_write

; step 7: clear interrupt status register
; cheung: performs this step earlier (after step #3)
ldi r16, NE2K_ISR
ldi r17, 0xff
rcall ne2k_write

; step 8: initialize the interrupt mask register
; cheung: 0 (out of order - after #7)
; ns: 0x0b
ldi r16, NE2K_IMR
ldi r17, 0 ; no interrupts, please
rcall ne2k_write

; step 9a: go to register page 1
ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE1 | NE2K_CR_STOP | NE2K_CR_NODMA
rcall ne2k_write

; step 9b: initialize hardware address
; (what?! shouldn't this already be set from EEPROM?)
ldi r30, low(ne2k_mac_addr) ; Load Z register low
ldi r31, high(ne2k_mac_addr) ; Load Z register high

ldi r16, NE2K_PAR0
ld r17, Z+
rcall ne2k_write

ldi r16, NE2K_PAR1
ld r17, Z+
rcall ne2k_write

ldi r16, NE2K_PAR2
ld r17, Z+
rcall ne2k_write
ldi  r16, NE2K_PAR3
dl  r17, Z+  
rcall  ne2k_write

ldi  r16, NE2K_PAR4  
dl  r17, Z+  
rcall  ne2k_write

ldi  r16, NE2K_PAR5  
dl  r17, Z+  
rcall  ne2k_write

; step 9c: initialize multicast address (i don't care about multicast)  
; ... not implemented ...

; step 9d: initialize CURRent pointer  
ldi  r16, NE2K_CURR  
ldi  r17, NE2K_START_PAGE  
rcall  ne2k_write

; step 10: put NIC in START mode  
ldi  r16, NE2K_CR  
ldi  r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_NODMA  
rcall  ne2k_write

; step 11: initialize transmit control register (disable loopback mode)  
ldi  r16, NE2K_TCR  
ldi  r17, NE2K_TCR_NOLPBK  
rcall  ne2k_write

; should i re-set DCR here to cancel loopback?  

; my step 12: initialize recieve configuration register so that we can  
; get packets  
ldi  r16, NE2K_RCR  
ldi  r17, NE2K_RCR_BCAST  
rcall  ne2k_write

; cheung reads the mac address from eeprom here. seems too late to me!  
pop  r31  
pop  r30  
pop  r18  
pop  r17  
pop  r16  
ret

;-------------------------------------------------------------------

ne2k_read_mac_eeprom:  
; read the mac address from the onboard EEPROM.  
; store the 6-byte value into the designated RAM location (ne2k_mac_addr).  
; copied functionality from linux ne.c driver initialization code.  
; apparently the mac address from the nic's onboard eeprom is mapped to  
; locations 0x0000 - 0x001f. i wish i had a spec sheet which told me these  
; things. it is a pain in the neck to have to infer these facts by reading  
; somebody else's sourcecode.

push  r16  
push  r17  
push  r30  
push  r31

ldi r30,low(ne2k_mac_addr) ; Load Z register low  
ldi r31,high(ne2k_mac_addr) ; Load Z register high

; set register page 0  
ldi  r16, NE2K_CR  
ldi  r17, NE2K_CR_PAGE0 | NE2K_CR_STOP | NE2K_CR_NODMA  
rcall  ne2k_write
; select byte wide transfers
ldi r16, NE2K_DCR
ldi r17, NE2K_DCR_BYTEMDMA | NE2K_DCR_FIFO8 | NE2K_DCR_NOLPBK
rcall ne2k_write

ldi r16, NE2K_RBCR0
ldi r17, 0
rcall ne2k_write

ldi r16, NE2K_RBCR1
ldi r17, 0
rcall ne2k_write

ldi r16, NE2K_IMR
ldi r17, 0
rcall ne2k_write

ldi r16, NE2K_ISR
ldi r17, 0xff
rcall ne2k_write

ldi r16, NE2K_RCR
ldi r17, NE2K_RCR_MONITOR ; receive off
rcall ne2k_write

ldi r16, NE2K_TCR
ldi r17, NE2K_TCR_INTLPBK ; transmit off
rcall ne2k_write

ldi r16, NE2K_RBCR0
ldi r17, 32 ; intend to read 32 bytes
rcall ne2k_write

ldi r16, NE2K_RBCR1
ldi r17, 0
rcall ne2k_write

ldi r16, NE2K_RSAR0
ldi r17, 0 ; low byte of start address (0x0000)
rcall ne2k_write

ldi r16, NE2K_RSAR1
ldi r17, 0 ; high byte of start address
rcall ne2k_write

ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_DMAREAD
rcall ne2k_write

ldi r16, NE2K_DATAPORT
rcall ne2k_read ; for some reason, 2 reads are required, otherwise you get duplicate values.
rcall ne2k_read ; the comments in the linux driver talk about values being "doubled up", but
st Z+, r17 ; i don't know why. whatever - it works this way and i don't have time to
investigate ;)

ldi r16, NE2K_DATAPORT
rcall ne2k_read
rcall ne2k_read
st Z+, r17

ldi r16, NE2K_DATAPORT
rcall ne2k_read
rcall ne2k_read
rcall ne2k_read
st Z+, r17

ldi r16, NE2K_DATAPORT
rcall ne2k_read
rcall ne2k_read
; end (abort) the DMA transfer
ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_NODMA
rcall ne2k_write
pop r31
pop r30
pop r17
pop r16
ret

-------------------------------------------------------------------
ne2k_read_packet:
; workhorse loop for processing network traffic.
push r10
push r11
push r12
push r13
push r14
push r15
push r16
push r17
push r18
push r19
push r20
push r30
push r31
ne2k_read_packet_start:
; goto register page 1
ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_NODMA
rcall ne2k_write
; read the CURRent pointer
ldi r16, NE2K_CURR
rcall ne2k_read
mov r10, r17 ; copy CURR into r10
; goto register page 0
ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_NODMA
rcall ne2k_write
; read the boundary pointer
ldi r16, NE2K_BNRY
rcall ne2k_read
mov r11, r17 ; copy BNRY into r11
cp r10, r11 ; compare CURR and BNRY
brne ne2k_read_packet_data ; if not equal, then there is data
; waiting
; to be read from the receive
; buffer ring.
rjmp ne2k_read_packet_end ; otherwise the receive buffer is empty,
; so we have nothing to do here.
; there is data in the NIC's rx buffer which we need to read out

; read packet data:

ldi r16, NE2K_RBCR0
ldi r17, 0xff ; i don't know how many bytes i intend
rcall ne2k_write ; to read, so just set this to the
; maximum

ldi r16, NE2K_RBCR1
ldi r17, 0xff
rcall ne2k_write

ldi r16, NE2K_RSAR0
ldi r17, 0 ; low byte of start address (0)
rcall ne2k_write

ldi r16, NE2K_RSAR1
mov r17, r11 ; high byte of start address (BNRY)
rcall ne2k_write

ldi r16, NE2K_CR ; begin the dma read
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_DMAREAD
rcall ne2k_write

ldi r16, NE2K_DATAPORT ; all dma reads come out of this
location

; the first 6 bytes are not part of the actual received ethernet packet.
; instead they contain some status information about the packet from
; the dp8390 chip. (see page 11 of the dp8390d spec)

rcall ne2k_read
mov r12, r17 ; receive status code (same structure as
; RSR - the receive status register)
rcall ne2k_read
mov r13, r17 ; next packet pointer
rcall ne2k_read
mov r14, r17 ; receive byte count low
rcall ne2k_read
mov r15, r17 ; receive byte count high

; i probably should check that the status code is "good", but for now
; just assume that it is ok.

; i probably should check that the length is reasonable, but for now
; let's just assume it is ok.

; now start reading the actual ethernet frame. (refer to Stevens "TCP/IP
; Illustrated Volume 1", page 23, for a nice diagram of the ethernet
; frame)
rcall ne2k_read ; destination mac address
rcall ne2k_read ; i'm not paying attention to this,
since
rcall ne2k_read ; the card should have already discarded
rcall ne2k_read ; packets not meant for me or broadcast
rcall ne2k_read
rcall ne2k_read

; the next 6 bytes are the source mac address. save this for my reply
ldi r30, low(ne2k_peer_mac_addr) ; Load Z register low
ldi r31, high(ne2k_peer_mac_addr); Load Z register high
rcall ne2k_read
st r30, r17
rcall ne2k_read
st r31, r17
rcall ne2k_read
st r30, r17
rcall ne2k_read
st r31, r17
rcall ne2k_read
st r30, r17
rcall ne2k_read
st r31, r17
rcall ne2k_read

st  Z+, r17
rcall ne2k_read
st  Z+, r17

; figure out if this is an 802.3 or Ethernet frame
rcall ne2k_read
ldi   r18, 0x06 ; if this byte is 0x06 or higher, it
cp    r17, r18 ; must be a "type" field, since a
brsh  ne2k_read_packet_eth ; "length" field cannot be 0x0600
    ; (1536) or higher.
rcall ne2k_read ; length low byte (ignore)
rcall ne2k_read ; DSAP
ldi   r18, 0xaa
cpse  r17, r18
rjmp  ne2k_read_packet_cleanup
rcall ne2k_read ; SSAP
ldi   r18, 0xaa
cpse  r17, r18
rjmp  ne2k_read_packet_cleanup
rcall ne2k_read ; cnt1
ldi   r18, 0x03
cpse  r17, r18
rjmp  ne2k_read_packet_cleanup
rcall ne2k_read ; org code 1
ldi   r18, 0
cpse  r17, r18
rjmp  ne2k_read_packet_cleanup
rcall ne2k_read ; org code 2
ldi   r18, 0
cpse  r17, r18
rjmp  ne2k_read_packet_cleanup
rcall ne2k_read ; org code 3
ldi   r18, 0
cpse  r17, r18
rjmp  ne2k_read_packet_cleanup
rcall ne2k_read ; type

ne2k_read_packet_eth:
; look at the "type" field in the ethernet frame. the types i
; understand are 0x0800 (IP) and 0x0806 (ARP)
ldi   r18, 0x08 ; type high byte
cpse  r17, r18
rjmp  ne2k_read_packet_cleanup
rcall ne2k_read ; type low byte
ldi   r18, 0x00 ; 0x0800: IP
cp    r17, r18
breq  ne2k_read_packet_ip
ldi   r18, 0x06 ; 0x0806: ARP
cp    r17, r18
breq  ne2k_read_packet_arp

; fallthrough: other type which i don't recognize
rjmp  ne2k_read_packet_cleanup

ne2k_read_packet_ip:
rjmp   ne2k_read_packet_ip2 ; do a long jump
ne2k_read_packet_arp:
    ; decode an ARP packet, and respond appropriately.
    ; see Stevens p.56
    ; confirm hardware type 0x0001
    rcall ne2k_read ; hardware type high byte
    ldi r18, 0x00
    cpse r17, r18
    rjmp ne2k_read_packet_cleanup
    rcall ne2k_read ; hardware type low byte
    ldi r18, 0x01
    cpse r17, r18
    rjmp ne2k_read_packet_cleanup

    ; confirm protocol type 0x0800
    rcall ne2k_read ; protocol type high byte
    ldi r18, 0x08
    cpse r17, r18
    rjmp ne2k_read_packet_cleanup
    rcall ne2k_read ; protocol type low byte
    ldi r18, 0x00
    cpse r17, r18
    rjmp ne2k_read_packet_cleanup

    ; confirm hardware size 6
    rcall ne2k_read
    ldi r18, 6
    cpse r17, r18
    rjmp ne2k_read_packet_cleanup

    ; confirm protocol size 4
    rcall ne2k_read
    ldi r18, 4
    cpse r17, r18
    rjmp ne2k_read_packet_cleanup

    ; confirm op code 0x0001 (ARP request)
    rcall ne2k_read
    ldi r18, 0x00
    cpse r17, r18
    rjmp ne2k_read_packet_cleanup
    rcall ne2k_read
    ldi r18, 0x01
    cpse r17, r18
    rjmp ne2k_read_packet_cleanup

    ; ignore sender's hardware address (we already recorded it)
    rcall ne2k_read
    rcall ne2k_read
    rcall ne2k_read
    rcall ne2k_read
    rcall ne2k_read
    rcall ne2k_read

    ; record sender's IP address
    ldi r30, low(ne2k_peer_ip_addr) ; Load Z register low
    ldi r31, high(ne2k_peer_ip_addr) ; Load Z register high
    rcall ne2k_read
    st Z+, r17
    rcall ne2k_read
    st Z+, r17
    rcall ne2k_read
    st Z+, r17
    rcall ne2k_read
    st Z+, r17

    ; ignore target hardware address (meaningless for this packet type)
    rcall ne2k_read
    rcall ne2k_read
rcall ne2k_read
rcall ne2k_read
rcall ne2k_read
rcall ne2k_read

; compare target IP address to our own. if its a match, then we should
; reply with an ARP reply. if it doesn't match, then this packet was
; meant for someone else, so we can ignore it.

ldi r30, low(ne2k_ip_addr) ; Load Z register low
ldi r31, high(ne2k_ip_addr) ; Load Z register high

ld r18, Z+ ; read first octet of my IP address
rcall ne2k_read ; read first octet of target IP address
cpse r17, r18
rjmp ne2k_read_packet_cleanup

ld r18, Z+ ; read next octet of my IP address
rcall ne2k_read ; read next octet of target IP address
cpse r17, r18
rjmp ne2k_read_packet_cleanup

ld r18, Z+ ; read next octet of my IP address
rcall ne2k_read ; read next octet of target IP address
cpse r17, r18
rjmp ne2k_read_packet_cleanup

ld r18, Z+ ; read next octet of my IP address
rcall ne2k_read ; read next octet of target IP address
cpse r17, r18
rjmp ne2k_read_packet_cleanup

; fallthrough: the target IP address is the same as my IP address.
; goodie!

; i've read all there is to read from this packet.
; end (abort) the DMA transfer

ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_NODMA
rcall ne2k_write

; update the BNRY (receive buffer ring boundary) pointer.

ldi r16, NE2K_BNRY
mov r17, r13; next packet pointer
rcall ne2k_write

; now send an ARP reply packet.
; rcall send_arp_reply

; ****
; ... i should test to make sure the card is not transmitting. otherwise
; ... i might stomp over the data to be transmitted ...
; ****

; set the remote byte count to 60 (arp packets are 60 bytes)

ldi r16, NE2K_RBCR0
ldi r17, 60
rcall ne2k_write

ldi r16, NE2K_RBCR1
ldi r17, 0
rcall ne2k_write

ldi r16, NE2K_RSAR0
ldi r17, 0 ; low byte of start address
rcall ne2k_write

ldi r16, NE2K_RSAR1
ldi r17, NE2K_TRANSMIT_BUFFER ; high byte of start address
rcall ne2k_write

; begin DMA write
ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_DMAMWRITE
rcall ne2k_write

ldi r16, NE2K_DATAPORT

; destination hardware address
ldi r30, low(ne2k_peer_mac_addr) ; Load Z register low
ldi r31, high(ne2k_peer_mac_addr); Load Z register high
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write

; source hardware address
ldi r30, low(ne2k_mac_addr) ; Load Z register low
ldi r31, high(ne2k_mac_addr) ; Load Z register high
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write

; "Ethernet" (not 802.3) type 0x0806 (=ARP)
ldi r17, 0x08
rcall ne2k_write
ldi r17, 0x06
rcall ne2k_write

; hardware type 0x0001
ldi r17, 0x00
rcall ne2k_write
ldi r17, 0x01
rcall ne2k_write

; protocol type 0x0800
ldi r17, 0x08
rcall ne2k_write
ldi r17, 0x00
rcall ne2k_write

; hardware size 6
ldi r17, 6
rcall ne2k_write

; protocol size 4
ldi r17, 4
rcall ne2k_write

; op 0x0002 (ARP reply)
ldi r17, 0x00
rcall ne2k_write
ldi r17, 0x02
rcall ne2k_write

; source hardware address
ldi r30, low(ne2k_mac_addr) ; Load Z register low
ldi r31, high(ne2k_mac_addr) ; Load Z register high
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
ld r17, Z+
rcall ne2k_write
rcall ne2k_write
rcall ne2k_write

; ****
; ... do i need wait for dma to end??? ...
; (see PCtoNIC from natsemi demo driver)
; ****

; end the DMA transfer
ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_NODMA
rcall ne2k_write

; how many bytes to send
ldi r16, NE2K_TBCR0
ldi r17, 60
rcall ne2k_write

ldi r16, NE2K_TBCR1
ldi r17, 0
rcall ne2k_write

; starting where
ldi r16, NE2K_TPSR
ldi r17, NE2K_TRANSMIT_BUFFER
rcall ne2k_write

; issue transmit command!
ldi r16, NE2K_CR
ldi r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_NODMA | NE2K_CR_TRANSMIT
rcall ne2k_write
rjmp ne2k_read_packet_cleanup

ne2k_read_packet_ip2:
; decode an IP packet, and respond appropriately

; first process the IP header (Stevens p.34)

; read version and length
rcall ne2k_read ; version (4 bits) + header length (4 bits)
mov r20, r17; store the header length in r20
andi r20, 0x0f ; mask out the version part
andi r17, 0xf0 ; mask out the length part
ldi r18, 0x40 ; IPv4 is the only version we accept
cpse r17, r18
rjmp ne2k_read_packet_cleanup
rcall ne2k_read ; ignore TOS
rcall ne2k_read ; ignore total length
rcall ne2k_read
rcall ne2k_read ; ignore identification number
rcall ne2k_read
rcall ne2k_read ; ignore fragmentation stuff
call ne2k_read
call ne2k_read ; ignore TTL
rcall ne2k_read ; read protocol
mov r19, r17 ; save for later in r19
rcall ne2k_read ; ignore checksum
call ne2k_read

; record sender's IP address
ldi r30,low(ne2k_peer_ip_addr) ; Load Z register low
ldi r31,high(ne2k_peer_ip_addr) ; Load Z register high
rcall ne2k_read
st Z+, r17
rcall ne2k_read
st Z+, r17
rcall ne2k_read
st Z+, r17
rcall ne2k_read
st Z+, r17

; compare destination IP address to our own. if its a match, then this
packet
; is for us. otherwise, this belongs to someone else.

ldi r30,low(ne2k_ip_addr) ; Load Z register low
ldi r31,high(ne2k_ip_addr) ; Load Z register high

ld r18, Z+ ; read first octet of my IP address
rcall ne2k_read ; read first octet of target IP address
cpse r17, r18
rjmp ne2k_read_packet_cleanup

ld r18, Z+ ; read next octet of my IP address
rcall ne2k_read ; read next octet of target IP address
cpse r17, r18
rjmp ne2k_read_packet_cleanup

ld r18, Z+ ; read next octet of my IP address
rcall ne2k_read ; read next octet of target IP address
cpse r17, r18
rjmp ne2k_read_packet_cleanup

ld r18, Z+ ; read next octet of my IP address
rcall ne2k_read ; read next octet of target IP address
cpse r17, r18
rjmp ne2k_read_packet_cleanup

; fallthrough: the destination IP address is the same as my IP address.
goodie!

; skip over any "options" in the ip header
subi r20, 5 ; 5 = size of ip header without any options
            ; (in 32-bit words)
ldi r17, 0
ne2k_read_packet_header1:
cp r20, r17
breq ne2k_read_packet_header2
subi r20, 1
rcall ne2k_read ; read 4-byte option field
rcall ne2k_read
rcall ne2k_read
rcall ne2k_read
rjmp ne2k_read_packet_header1

ne2k_read_packet_header2:
; we have now advanced the read pointer up to the first byte of
; the "data" portion of the IP packet

; ok, now look back at the protocol field and jump to the right
; code to handle the packet type
ldi r18, 1 ; icmp
cp r19, r18
breq ne2k_read_packet_icmp

ldi r18, 6 ; tcp
cp r19, r18
breq ne2k_read_packet_tcp

ldi r18, 17 ; udp
cp r19, r18
breq ne2k_read_packet_udp
ne2k_read_packet_icmp:
    ; ... icmp not implemented ...
    ldi    r16, 0
    ldi    r17, 0
    rcall  move_cursor
    ldi    r16, 'i'
    rcall  print_to_lcd
    ldi    r16, 'c'
    rcall  print_to_lcd
    ldi    r16, 'm'
    rcall  print_to_lcd
    ldi    r16, 'p'
    rcall  print_to_lcd
    ldi    r16, '!
    rcall  print_to_lcd
    ldi    r16, NE2K_DATAPORT
    rjmp   ne2k_read_packet_cleanup

ne2k_read_packet_tcp:
    ; ... tcp not implemented ...
    ldi    r16, 0
    ldi    r17, 0
    rcall  move_cursor
    ldi    r16, 't'
    rcall  print_to_lcd
    ldi    r16, 'c'
    rcall  print_to_lcd
    ldi    r16, 'p'
    rcall  print_to_lcd
    ldi    r16, '!
    rcall  print_to_lcd
    ldi    r16, NE2K_DATAPORT
    rjmp   ne2k_read_packet_cleanup

ne2k_read_packet_udp:
    rcall  ne2k_read ; ignore source port number
    rcall  ne2k_read
    rcall  ne2k_read ; test destination port number
    ldi    r18, high(NE2K_LISTEN_PORT)
    cpse   r17, r18
    rjmp   ne2k_read_packet_cleanup
    rcall  ne2k_read
    ldi    r18, low(NE2K_LISTEN_PORT)
    cpse   r17, r18
    rjmp   ne2k_read_packet_cleanup
    rcall  ne2k_read ; ignore udp length
    rcall  ne2k_read
    rcall  ne2k_read ; ignore udp checksum
    rcall  ne2k_read
    ; now we're finally at the interesting part - the text string to print
    ; onto the LCD screen.
    ldi    r18, 255 ; start at row #-1
    ne2k_read_packet_printloop1:
        inc    r18 ; go to next row
        cpi    r18, 4 ; if we've moved below the end of the
                       ; screen...
        breq   ne2k_read_packet_printloop3 ; ...exit the loop
    ldi    r19, 0 ; go back to column #0
    mov    r16, r18

mov    r17, r19
rcall  move_cursor            ; issue carriage return instruction

ne2k_read_packet_printloop2:
inc    r19             ; increment column pointer
cpi    r19, 21           ; if we've moved off the right of the
screen...
breq   ne2k_read_packet_printloop1 ; ...do a carriage return
ldi    r16, NE2K_DATAPORT ; read a byte from the packet
rcall  ne2k_read
mov    r16, r17
rcall  print_to_lcd       ; print it on the screen
rjmp   ne2k_read_packet_printloop2 ; loop

ne2k_read_packet_printloop3:

; voila - the data is on the lcd screen. ignore whatever may be left.
rjmp   ne2k_read_packet_cleanup

ne2k_read_packet_cleanup:

; end (abort) the DMA transfer
ldi    r16, NE2K_CR
ldi    r17, NE2K_CR_PAGE0 | NE2K_CR_START | NE2K_CR_NODMA
rcall  ne2k_write

; update the BNRY (recv buffer ring boundary) pointer.
; r13 = next packet pointer from NIC packet header.
; note: there seem to be 2 ways of setting this pointer. you can
; set it to one less than the next packet pointer, or equal
; to the next packet pointer. it seems simpler to make it equal -
; i'm not sure why you would want to do it the other way.
ldi    r16, NE2K_BNRY
mov    r17, r13; next packet pointer
rcall  ne2k_write

call   set_led_on
rcall  delay_100ms
rcall  set_led_off

ne2k_read_packet_end:
pop    r31
pop    r30
pop    r20
pop    r19
pop    r18
pop    r17
pop    r16
pop    r15
pop    r14
pop    r13
pop    r12
pop    r11
pop    r10
ret

;-------------------------------------------------------------------
ne2k_establish_ip_address:
; stick our hardcoded IP address into the proper spot in the microcontroller's
; RAM
; TBD: replace this routine with a DHCP or BOOTP client!
push   r16
push   r30
push   r31
ldi    r30, low(ne2k_ip_addr)   ; Load Z register low
ldi r31, high(ne2k_ip_addr) ; Load Z register high
ldi r16, NE2K_IP_OCTET_1
st Z+, r16
ldi r16, NE2K_IP_OCTET_2
st Z+, r16
ldi r16, NE2K_IP_OCTET_3
st Z+, r16
ldi r16, NE2K_IP_OCTET_4
st Z+, r16
pop r31
pop r30
pop r16
ret

;-------------------------------------------------------------
; *** LCD ***
;-------------------------------------------------------------

blink_led:
  push r16
  ldi r16, 0
blink_led_loop:
    rcall delay_5ms
    add r16, ONE
    cpi r16, 0
    brne blink_led_loop
    cp LED, ZERO
    breq blink_led_on
    rcall set_led_off
    ldi r16, 0
    rjmp blink_led_loop
blink_led_on:
  rcall set_led_on
  ldi r16, 0
  rjmp blink_led_loop
pop r16
ret

;-------------------------------------------------------------

set_led_on:
  push r16
  ldi r16, LED_BIT
  mov LED, r16
  ldi r16, $FF ; all I/O pins output
  out LCD_PORT_DDR, r16
  mov r16, LED
  out LCD_PORT, r16
pop r16
ret

;-------------------------------------------------------------

set_led_off:
  push r16
  ldi r16, 0
mov  LED, r16

ldi  r16, $FF ; all I/O pins output
out  LCD_PORT_DDR, r16

mov  r16, LED
out  LCD_PORT, r16

pop  r16
ret

-------------------------------------------------------------------

write_lcd_ir:
; writes an 8-bit value to the LCD instruction register.
; the value to be written is presumed to be in R16.
; (RS low, R/W low, E high-to-low)

push r17
push r18
push r19
push r20

ldi  r17, $FF ; all I/O pins output
out  LCD_PORT_DDR, r17

mov  r17, r16
andi  r17, DB_BITS ; grab the high bits
or  r17, LED
mov  r18, r17
ori  r18, ENABLE_BIT

mov  r19, r16
lsl r19 ; grab the low bits...
lsl r19 ; ...and shift them up
lsl
lsl
lsl
or  r19, LED
mov  r20, r19
ori  r20, ENABLE_BIT

out  LCD_PORT, r17 ; upper 4 bits (E low)
out  LCD_PORT, r18 ; upper 4 bits (E high)
out  LCD_PORT, r17 ; upper 4 bits (E low)
out  LCD_PORT, r19 ; lower 4 bits (E low)
out  LCD_PORT, r20 ; lower 4 bits (E high)
out  LCD_PORT, r19 ; lower 4 bits (E low)

pop  r20
pop  r19
pop  r18
pop  r17
ret

-------------------------------------------------------------------

write_lcd_dr:
; writes an 8-bit value to the LCD data register.
; the value to be written is presumed to be in R16.
; (RS high, R/W low, E high-to-low)

push r17
push r18
push r19
push r20

ldi  r17, $FF ; all I/O pins output
out  LCD_PORT_DDR, r17

mov  r17, r16
andi  r17, DB_BITS ; grab the high bits
ori r17, RS_BIT
or r17, LED
mov r18, r17
ori r18, ENABLE_BIT

mov r19, r16
lsl r19 ; grab the low bits...
  lsl r19 ; ...and shift them up
  lsl r19
lor r19, RS_BIT
or r19, LED
mov r20, r19
ori r20, ENABLE_BIT

out LCD_PORT, r17 ; upper 4 bits (E low)
out LCD_PORT, r18 ; upper 4 bits (E high)
out LCD_PORT, r17 ; upper 4 bits (E low)
out LCD_PORT, r19 ; lower 4 bits (E low)
out LCD_PORT, r20 ; lower 4 bits (E high)
out LCD_PORT, r19 ; lower 4 bits (E low)

pop r20
pop r19
pop r18
pop r17
ret

read_lcd_ir:
  ret

read_lcd_dr:
  ret

;-------------------------------------------------------------------

write_silly_string:
push r16
push r17
  ldi r16, 0 ; row
  ldi r17, 0 ; col
  rcall move_cursor

  ldi r16, 'D'
  rcall print_to_lcd
  ldi r16, 'a'
  rcall print_to_lcd
  ldi r16, 'v'
  rcall print_to_lcd
  ldi r16, 'e'
  rcall print_to_lcd
  ldi r16, '
  rcall print_to_lcd
  ldi r16, 'C'
  rcall print_to_lcd
  ldi r16, 'l'
  rcall print_to_lcd
  ldi r16, 'a'
  rcall print_to_lcd
  ldi r16, 'u'
  rcall print_to_lcd
  ldi r16, 's'
  rcall print_to_lcd
  ldi r16, 's'
  rcall print_to_lcd
  ldi r16, 'n'
  rcall print_to_lcd

  ldi r16, 1 ; row
  ldi r17, 0 ; col
rcall move_cursor
ldi r16, 'E'
rcall print_to_lcd
ldi r16, 'E'
rcall print_to_lcd
ldi r16, 'Z'
rcall print_to_lcd
ldi r16, '8'
rcall print_to_lcd
ldi r16, '1'
rcall print_to_lcd
ldi r16, ':'
rcall print_to_lcd
ldi r16, 'A'
rcall print_to_lcd
ldi r16, 'V'
rcall print_to_lcd
ldi r16, 'R'
rcall print_to_lcd
ldi r16, '
'
rcall print_to_lcd
ldi r16, 'N'
rcall print_to_lcd
ldi r16, '2'
rcall print_to_lcd
ldi r16, 'K'
rcall print_to_lcd
pop r17
pop r16
ret

;-------------------------------------------------------------------

lcd_write_mac_addr:
push r16
push r17
push r30
push r31
ldi r16, 2; row
ldi r17, 0; col
rcall move_cursor
ldi r30, low(ne2k_mac_addr); Load Z register low
ldi r31, high(ne2k_mac_addr); Load Z register high
ld r16, Z+
rcall lcd_print_hex_byte
ldi r16, ':'
rcall print_to_lcd
ld r16, Z+
rcall lcd_print_hex_byte
ldi r16, ':'
rcall print_to_lcd
ld r16, Z+
rcall lcd_print_hex_byte
ldi r16, ':'
rcall print_to_lcd
ld r16, Z+
rcall lcd_print_hex_byte
ldi r16, ':'
rcall print_to_lcd
ld r16, Z+
rcall lcd_print_hex_byte
pop r31
pop r30
pop r17
pop r16
ret

;-------------------------------------------------------------------
lcd_write_ip_addr:
push r16
push r17
push r30
push r31
ldi r16, 3          ; row
ldi r17, 0          ; col
rcall move_cursor
ldi r30, low(ne2k_ip_addr) ; Load Z register low
ldi r31, high(ne2k_ip_addr) ; Load Z register high
ld r16, Z+
rcall lcd_print_dec_byte
ldi r16, '.'
rcall print_to_lcd
ld r16, Z+
rcall lcd_print_dec_byte
ldi r16, '.'
rcall print_to_lcd
ld r16, Z+
rcall lcd_print_dec_byte
ldi r16, '.'
rcall print_to_lcd
ld r16, Z+
rcall lcd_print_dec_byte
pop r31
pop r30
pop r17
pop r16
ret

;-------------------------------------------------------------------
lcd_print_hex_byte:
; r16 contains a byte
push r16
push r17
push r18

mov r17, r16; MSB
lsr r17
lsr r17
lsr r17
lsr r17
andi r17, 0b00001111

mov r18, r16; LSB
andi r18, 0b00001111

mov r16, r17
rcall lcd_print_hex_nibble
mov r16, r18
rcall lcd_print_hex_nibble
pop r18
pop r17
pop r16
ret
; ----------------------------------

lcd_print_hex_nibble:
; r16 contains a nibble
    push r16
    push r17
    ldi r17, '0'; ascii '0' character
    add r16, r17
    cpi r16, ':'
    brlo lcd_print_hex_nibble_ok ; if number < 10, can use
    ; ascii '0' through '9'
    ldi r17, 'A' - ':' ; jump over some punctuation to get to
    ; the letters (ABCDEF ascii codes)
    add r16, r17
   lcd_print_hex_nibble_ok:
    rcall print_to_lcd
    pop r17
    pop r16
    ret

; ----------------------------------

lcd_print_dec_byte:
; r16 contains a byte
; EEK this is awful code!
    push r16
    push r17
    push r18
    mov r17, r16
    dec_byte_200:
    ldi r18, 200
    cp r17, r18
    brlo dec_byte_100
    sub r17, r18
    ldi r16, '2'
    rcall print_to_lcd
    rjmp dec_byte_last
    dec_byte_100:
    ldi r18, 100
    cp r17, r18
    brlo dec_byte_90
    sub r17, r18
    ldi r16, '1'
    rcall print_to_lcd
    dec_byte_90:
    ldi r18, 90
    cp r17, r18
    brlo dec_byte_80
    sub r17, r18
    ldi r16, '9'
    rcall print_to_lcd
    rjmp dec_byte_last
    dec_byte_80:
    ldi r18, 80
    cp r17, r18
    brlo dec_byte_70
    sub r17, r18
    ldi r16, '8'
    rcall print_to_lcd
    rjmp dec_byte_last
    dec_byte_70:
    ldi r18, 70
    ; ----------------------------------
cp r17, r18
brlo dec_byte_60
sub r17, r18
ldi r16, '7'
rcall print_to_lcd
rjmp dec_byte_last
dec_byte_60:
ldi r18, 60
cp r17, r18
brlo dec_byte_50
sub r17, r18
ldi r16, '6'
rcall print_to_lcd
rjmp dec_byte_last
dec_byte_50:
ldi r18, 50
cp r17, r18
brlo dec_byte_40
sub r17, r18
ldi r16, '5'
rcall print_to_lcd
rjmp dec_byte_last
dec_byte_40:
ldi r18, 40
cp r17, r18
brlo dec_byte_30
sub r17, r18
ldi r16, '4'
rcall print_to_lcd
rjmp dec_byte_last
dec_byte_30:
ldi r18, 30
cp r17, r18
brlo dec_byte_20
sub r17, r18
ldi r16, '3'
rcall print_to_lcd
rjmp dec_byte_last
dec_byte_20:
ldi r18, 20
cp r17, r18
brlo dec_byte_10
sub r17, r18
ldi r16, '2'
rcall print_to_lcd
rjmp dec_byte_last
dec_byte_10:
ldi r18, 10
cp r17, r18
brlo dec_byte_last
sub r17, r18
ldi r16, '1'
rcall print_to_lcd
dec_byte_last:
ldi r16, '0'
add r17, r16
mov r16, r17
rcall print_to_lcd
pop r18
pop r17
pop r16
ret

;-------------------------------------------------------------------
move_cursor:
; line (0-3) is presumed to be in r16
; column (0-15) is presumed to be in r17
; sends appropriate instruction to the LCD to relocate the cursor
push r16
push r18
push r19

cpi r16, 0
breq move_cursor_line_0

cpi r16, 1
breq move_cursor_line_1

cpi r16, 2
breq move_cursor_line_2

rjmp move_cursor_line_3

move_cursor_line_0:
ldi r18, $0
rjmp move_line_ok

move_cursor_line_1:
ldi r18, $40
rjmp move_line_ok

move_cursor_line_2:
ldi r18, $14
rjmp move_line_ok

move_cursor_line_3:
ldi r18, $54
rjmp move_line_ok

move_line_ok:
mov r19, r17
cpi r19, 21
brlo move_column_ok

ldi r19, 20

move_column_ok:
add r18, r19
ori r18, 0b10000000
mov r16, r18
rcall write_lcd_ir
rcall delay_120us

pop r19
pop r18
pop r16
ret

-------------------------------------------------------------------
print_to_lcd:
; send the character in r16 to the screen
; (at the current cursor position)
        rcall write_lcd_dr
        rcall delay_120us
        ret

-------------------------------------------------------------------
initialize_lcd:
; go through the 4-bit LCD initialization command sequence.
push r16

        ; initialize LCD I/O port
        ldi r16, $FF      ; all bits output
        out LCD_PORT_DDR, r16

        ldi r16, 0        ; all bits low
or r16, LED ; set the LED
out LCD_PORT, r16

rcall delay_5ms;
rcall delay_5ms;
rcall delay_5ms;
rcall delay_5ms;

ldi r16, 0b00110000
ori r16, ENABLE_BIT
out LCD_PORT, r16
ldi r16, 0b00110000
out LCD_PORT, r16

rcall delay_5ms

ldi r16, 0b00110000
ori r16, ENABLE_BIT
out LCD_PORT, r16
ldi r16, 0b00110000
out LCD_PORT, r16

rcall delay_120us

ldi r16, 0b00100000
ori r16, ENABLE_BIT
out LCD_PORT, r16
ldi r16, 0b00100000
outLCD_PORT, r16

rcall delay_120us

ldi r16, 0b00101000
rcall write_lcd_ir
rcall delay_120us

; function set, DL=0, N=1, F=0:
; data length = 4 bits
; number of display lines = 4
; font = 5x7
ldi r16, 0b00101000
rcall write_lcd_ir
rcall delay_120us

; display on, D=1, C=0, B=0
; display = ON
; cursor visible = OFF
; cursor blink = OFF
ldi r16, 0b00001100
rcall write_lcd_ir
rcall delay_120us

; display clear
ldi r16, 0b00000001
rcall write_lcd_ir
rcall delay_5ms

; entry mode set, I/D=1, S=0
; increment/decrement cursor = increment
; shift = OFF
ldi r16, 0b000000110
rcall write_lcd_ir
rcall delay_120us

pop r16
ret
; spin delays
;
; at 4MHz, a single-cycle instruction takes 0.25 microseconds.
; a branch should consume 2 cycles, since it breaks the pipelining

delay_40us:  
    push r16
    ;ldi r16, $40
    ;ldi r16, 54 ; 54 * 3 * 0.25 = 40.5 usec (64?)
    delay_40us_loop:  ; 3-cycle loop
    dec r16
    brne delay_40us_loop
    pop r16
    ret

delay_100us:
    push r16
    ;ldi r16, $88
    ldi r16, 136; 136 * 3 * 0.25 = 102 usec
    delay_100us_loop:  ; 3-cycle loop
    dec r16
    brne delay_100us_loop
    pop r16
    ret

delay_120us:
    push r16
    ldi r16, 160; 160 * 3 * 0.25 = 120 usec
    delay_120us_loop:  ; 3-cycle loop
    dec r16
    brne delay_120us_loop
    pop r16
    ret

delay_5ms:
    push r16
    push r17
    ;ldi r16, $1c
    ldi r16, 27 ; 27 * 192 = 5184 usec = 5.2ms
    delay_5ms_outer_loop:
    ldi r17, 0
    ; 256 * 3 * 0.25 = 192 usec
    delay_5ms_inner_loop:  ; 3-cycle loop
    dec r17
    brne delay_5ms_inner_loop
    dec r16
    brne delay_5ms_outer_loop
    pop r17
    pop r16
    ret

delay_25ms:
    rcall delay_5ms;
    rcall delay_5ms;
    rcall delay_5ms;
    rcall delay_5ms;
    rcall delay_5ms;
    ret
9. References
9.1. Similar projects on the web

During the course of this project, I found several similar projects on the web. Some use AVR's, but most some use PICs or other micorontrollers (or FPGAs). Some use an ISA Ethernet card, and some use an Ethernet controller chipset such as the Crystal 8900 or a similar product from RealTek. I borrowed ideas from several of them, especially for dealing with the ISA/NE2000 interface, since I didn't have a good spec. Many of these are incomplete "works in progress", at least as of December 2000. Here are the URLs I know of:

PIC webcam
http://members.bellatlantic.net/~echeung/awards/pic2k/pic2k.htm
PIC+NE2000 - includes PIC sourcecode which I used as a reference.

Circuit Cellar Magazine: The Ethernet Development Board
http://www.chipcenter.com/circuitcellar/october00/c1000fe1.htm
A Crystal CS8900A based project

Circuit Cellar Magazine: A $25 web server
AVR+NE2000 in 16-bit mode. schematic but no sourcecode

PicoWeb
http://www.picoweb.net/
commercial evolution of the above project. uses RealTek chip
AVR Projects by Jason Kyle
http://www.eavr.jpke.co.nz
AVR+NE2000. schematic but no sourcecode. work in progress.

Embedded 10BaseT Ethernet
http://www.embeddedethernet.com/
CAD documents for a CS8900 daughterboard.

Liquorice project
http://liquorice.sourceforge.net/hardware/
Very little info. Just getting started.

EtherNut Embedded Ethernet Board
http://www.egnite.de/ethernut/
Commercial product. AVR Mega + RealTek. no sourcecode.

Sedat
http://www.cs.tamu.edu/course-info/cpsc483/spring98/rabi/99a/g4/intro.html
FPGA-based project.

9.2. Other references and documentation.
Linux Ethernet NE2000 driver
ne.c, ne.h, from any Linux distribution

Linux Ethernet HOWTO

Ethernet FAQ
http://ilima.eng.hawaii.edu/XCoNET/Ethernet.htm

TCP/IP Illustrated
Stevens

National Semiconductor Datasheets
http://www.national.com/parametric/0,1850,2649,00.html

NE2000.386

Embedded Ethernet links
http://www.3beans.com/ether.html

comp.arch.embedded FAQ
http://www.execpc.com/~geezer/embed/cae.htm
AVR RISC Microcontroller Data Book
Atmel corporation

Pletronics P1100-HC series oscillator spec sheet
p1100-hc.pdf
http://www.pletronics.com/

AND721GST Spec Sheet
AND721GST.pdf
http://www.purdyelectronics.com/

AND Application Note: Intelligent Alphanumeric Displays
AlphanumericAppNotes.pdf
http://www.purdyelectronics.com/

TechFest - ISA Bus Technical Summary
http://www.techfest.com/hardware/bus/isa.htm

The ISA Bus
http://users.supernet.com/sokos/ISA.HTM

Another ISA web page
http://sunsite.tut.fi/hwb/co_ISA_Tech.html

10. Notes
This document and its content are Copyright 2000 Dave Clausen, all rights reserved. Content is provided "as-is", with no warranties of any kind. Sourcecode and schematics in this document are covered by the GNU General Public License (GPL), which is described on the web at http://www.fsf.org/copyleft/gpl.html.