Version 1.0

HYDRA EtherX

Programming and User Manual

Shane Avery

& Andre' LaMothe

Developed by Avery Digital

HYDRA™ ETHERX Card User Manual v1.0
Copyright © 2008 Avery Digital

Authors
Shane Avery
Andre' LaMothe

Editor/Technical Reviewer
The "Collective"

Printing
0001

ISBN
Pending

Licensing, Terms & Conditions

AVERY DIGITAL, . END-USER LICENSE AGREEMENT FOR HYDRA ETHERX HARDWARE, SOFTWARE , EBOOKS, AND USER MANUALS

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS PRODUCT. IT CONTAINS SOFTWARE, THE USE OF WHICH IS LICENSED BY AVERY DIGITAL, TO ITS CUSTOMERS FOR THEIR USE ONLY AS SET FORTH BELOW. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT USE THE SOFTWARE OR HARDWARE. USING ANY PART OF THE SOFTWARE OR HARDWARE INDICATES THAT YOU ACCEPT THESE TERMS.

GRANT OF LICENSE: AVERY DIGITAL (the "Licensor") grants to you this personal, limited, non-exclusive, non-transferable, non-assignable license solely to use in a single copy of the Licensed Works on a single computer for use by a single concurrent user only, and solely provided that you adhere to all of the terms and conditions of this Agreement. The foregoing is an express limited use license and not an assignment, sale, or other transfer of the Licensed Works or any Intellectual Property Rights of Licensor.

ASSENT: By opening the files and or packaging containing this software and or hardware, you agree that this Agreement is a legally binding and valid contract, agree to abide by the intellectual property laws and all of the terms and conditions of this Agreement, and further agree to take all necessary steps to ensure that the terms and conditions of this Agreement are not violated by any person or entity under your control or in your service.

OWNERSHIP OF SOFTWARE AND HARDWARE: The Licensor and/or its affiliates or subsidiaries own certain rights that may exist from time to time in this or any other jurisdiction, whether foreign or domestic, under patent law, copyright law, publicity rights law, moral rights law, trade secret law, trademark law, unfair competition law or other similar protections, regardless of whether or not such rights or protections are registered or perfected (the "Intellectual Property Rights"), in the computer software and hardware, together with any related documentation (including design, systems and user) and other materials for use in connection with such computer software and hardware in this package (collectively, the "Licensed Works"). ALL INTELLECTUAL PROPERTY RIGHTS IN AND TO THE LICENSED WORKS ARE AND SHALL REMAIN IN LICENSOR.

RESTRICTIONS:
(a) You are expressly prohibited from copying, modifying, merging, selling, leasing, redistributing, assigning, or transferring in any matter, Licensed Works or any portion thereof.
(b) You may make a single copy of software materials within the package or otherwise related to Licensed Works only as required for backup purposes.
(c) You are also expressly prohibited from reverse engineering, decompiling, translating, disassembling, deciphering, decrypting, or otherwise attempting to discover the source code of the Licensed Works as the Licensed Works contain proprietary material of Licensor. You may not otherwise modify, alter, adapt, port, or merge the Licensed Works.
(d) You may not remove, alter, deface, overprint or otherwise obscure Licensor patent, trademark, service mark or copyright notices.
(e) You agree that the Licensed Works will not be shipped, transferred or exported into any other country, or used in any manner prohibited by any government agency or any export laws, restrictions or regulations.
(f) You may not publish or distribute in any form of electronic or printed communication the materials within or otherwise related to Licensed Works, including but not limited to the object code, documentation, help files, examples, and benchmarks.

TERM: This Agreement is effective until terminated. You may terminate this Agreement at any time by uninstalling the Licensed Works and destroying all copies of the Licensed Works both HARDWARE and SOFTWARE. Upon any termination, you agree to uninstall the Licensed Works and return or destroy all copies of the Licensed Works, any accompanying documentation, and all other associated materials.

WARRANTIES AND DISCLAIMER: EXCEPT AS EXPRESSLY PROVIDED OTHERWISE IN A WRITTEN AGREEMENT BETWEEN LICENSOR AND YOU, THE LICENSED WORKS ARE NOW PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THE WARRANTY OF NON-INFRINGEMENT. WITHOUT LIMITING THE FOREGOING, LICENSOR MAKES NO WARRANTY THAT (i) THE LICENSED WORKS WILL MEET YOUR REQUIREMENTS, (ii) THE USE OF THE LICENSED WORKS WILL BE UNINTERRUPTED, TIMELY, SECURE, OR ERROR-FREE, (iii) THE RESULTS THAT MAY BE OBTAINED FROM THE USE OF THE LICENSED WORKS WILL BE ACCURATE OR RELIABLE, (iv) THE QUALITY OF THE LICENSED WORKS WILL MEET YOUR EXPECTATIONS, (v) ANY ERRORS IN THE LICENSED WORKS WILL BE CORRECTED, AND/OR (vi) YOU MAY USE, PRACTICE, EXECUTE, OR ACCESS THE LICENSED WORKS WITHOUT VIOLATING THE INTELLECTUAL PROPERTY RIGHTS OF OTHERS. SOME STATES OR JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY MAY LAST, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IF CALIFORNIA LAW IS NOT HELD TO APPLY TO THIS AGREEMENT FOR ANY REASON, THEN IN JURISDICTIONS WHERE WARRANTIES, GUARANTEES, REPRESENTATIONS, AND/OR CONDITIONS OF ANY TYPE MAY NOT BE DISCLAIMED, ANY SUCH WARRANTY, GUARANTEE, REPRESENATION AND/OR WARRANTY IS: (1) HEREBY LIMITED TO THE PERIOD OF EITHER (A) Five (5) DAYS FROM THE DATE OF OPENING THE PACKAGE CONTAINING THE LICENSED WORKS OR (B) THE SHORTEST PERIOD ALLOWED BY LAW IN THE APPLICABLE JURISDICTION IF A FIVE (5) DAY LIMITATION WOULD BE UNENFORCEABLE; AND (2) LICENSOR'S SOLE LIABILITY FOR ANY BREACH OF ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION, AND/OR CONDITION SHALL BE TO PROVIDE YOU WITH A NEW COPY OF THE LICENSED WORKS. IN NO EVENT SHALL LICENSOR OR ITS SUPPLIERS BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, THOSE RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT LICENSOR HAD BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE LICENSED WORKS. SOME JURISDICTIONS PROHIBIT THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

SEVERABILITY: In the event any provision of this License Agreement is found to be invalid, illegal or unenforceable, the validity, legality and enforceability of any of the remaining provisions shall not in any way be affected or impaired and a valid, legal and enforceable provision of similar intent and economic impact shall be substituted therefore.

ENTIRE AGREEMENT: This License Agreement sets forth the entire understanding and agreement between you and AVERY DIGITAL, supersedes all prior agreements, whether written or oral, with respect to the Software, and may be amended only in a writing signed by both parties.

AVERY DIGITAL
800 Sonja Ave.
Ridgecrest, CA 93555

Version & Support/Web Site

This document is valid with the following hardware, software and firmware versions:

- HYDRA Game Console Revision A. or greater.
- Propeller Tool 1.0 or greater.

The information herein will usually apply to newer versions but may not apply to older versions. Please contact Avery Digital for any questions you may have.

Visit **www.xgamestation.com** for downloads, support and access to the XGameStation/HYDRA user community and more!

Visit **www.averydigital.com/products.html** for additional downloads, examples, and documentation!

For technical support, sales, general questions, share feedback, please contact Avery Digital at:

support@averydigital.com

Table of Contents

HYDRA ETHERX Card User Manual

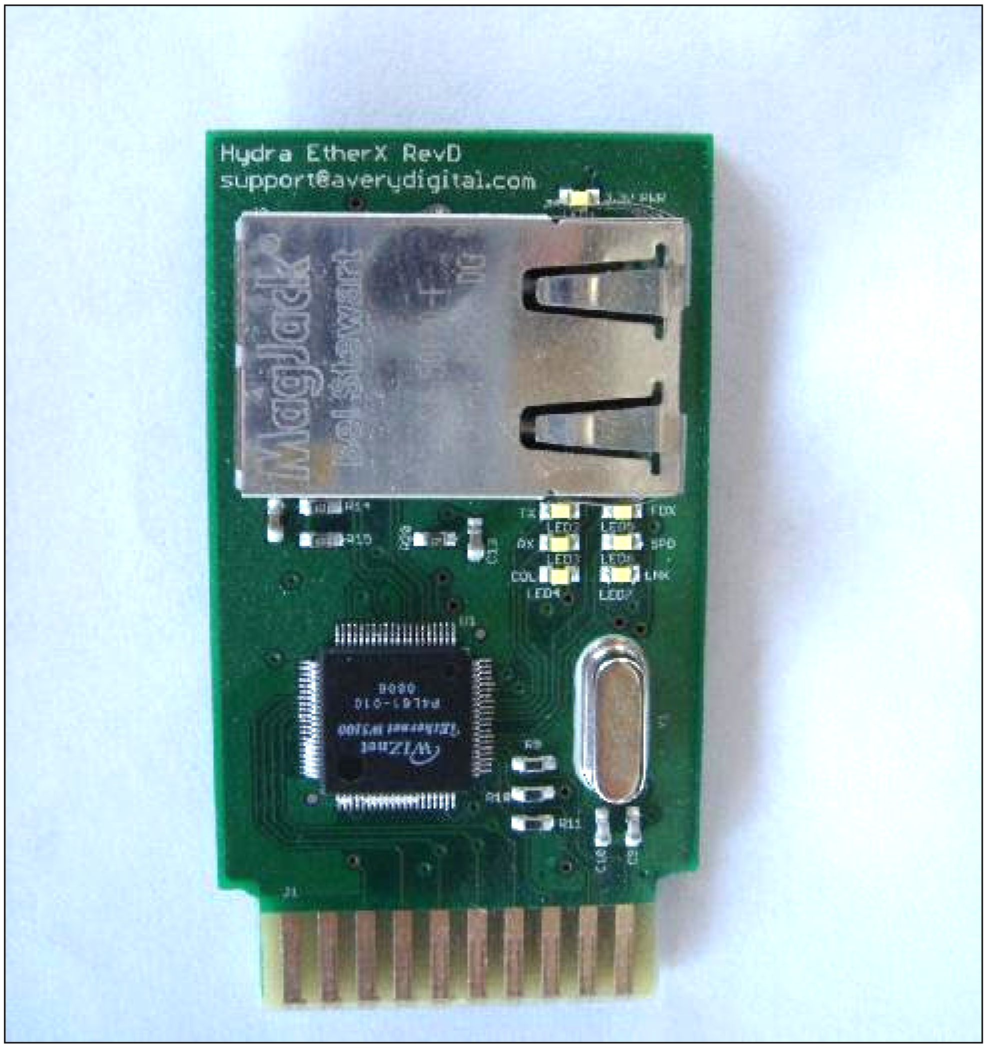
1.0 HYDRA ETHERX Card Manual Overview

Welcome to the user manual for the **HYDRA ETHERX Card**. The **HYDRA ETHERX Card** enhances the functionality of the HYDRA by adding the ability to send and receive data via Ethernet. The hardware interface is facilitated by a standard HYDRA expansion card which communicates directly to the W5100 chip using **SPI** (serial peripheral interface).

There is a lot you need to know to write software yourself to access the W5100. From the bottom up, first, there is the electrical interface to the card itself which is a number of signals and power, then the actual communications to the device is based on the **SPI** (serial peripheral interface) which is a **3-line serial protocol** (data in, data out, clock) in SPI mode, of course. That's how you talk to the EtherX card. Then you need to know what data to actually read and write the W5100 to set it up to read and write data using standard internet protocols such as TCP or UDP.

This is a lot of work no doubt, but hopefully this manual will give you insight into how to do this yourself. And of course, I will provide pre-made drivers and an API written in **SPIN** for ease of understanding. Nevertheless, I highly recommend that you try and conquer everything from the electrical interface, the SPI protocol, the W5100 itself (which is configurable beyond what is described in this document), as well as the internet protocols that we all use on a daily basis.

Figure 1.0 – The HYDRA EtherX Card.



2.0 Product Contents

The HYDRA EtherX Card kit consists of the following items:

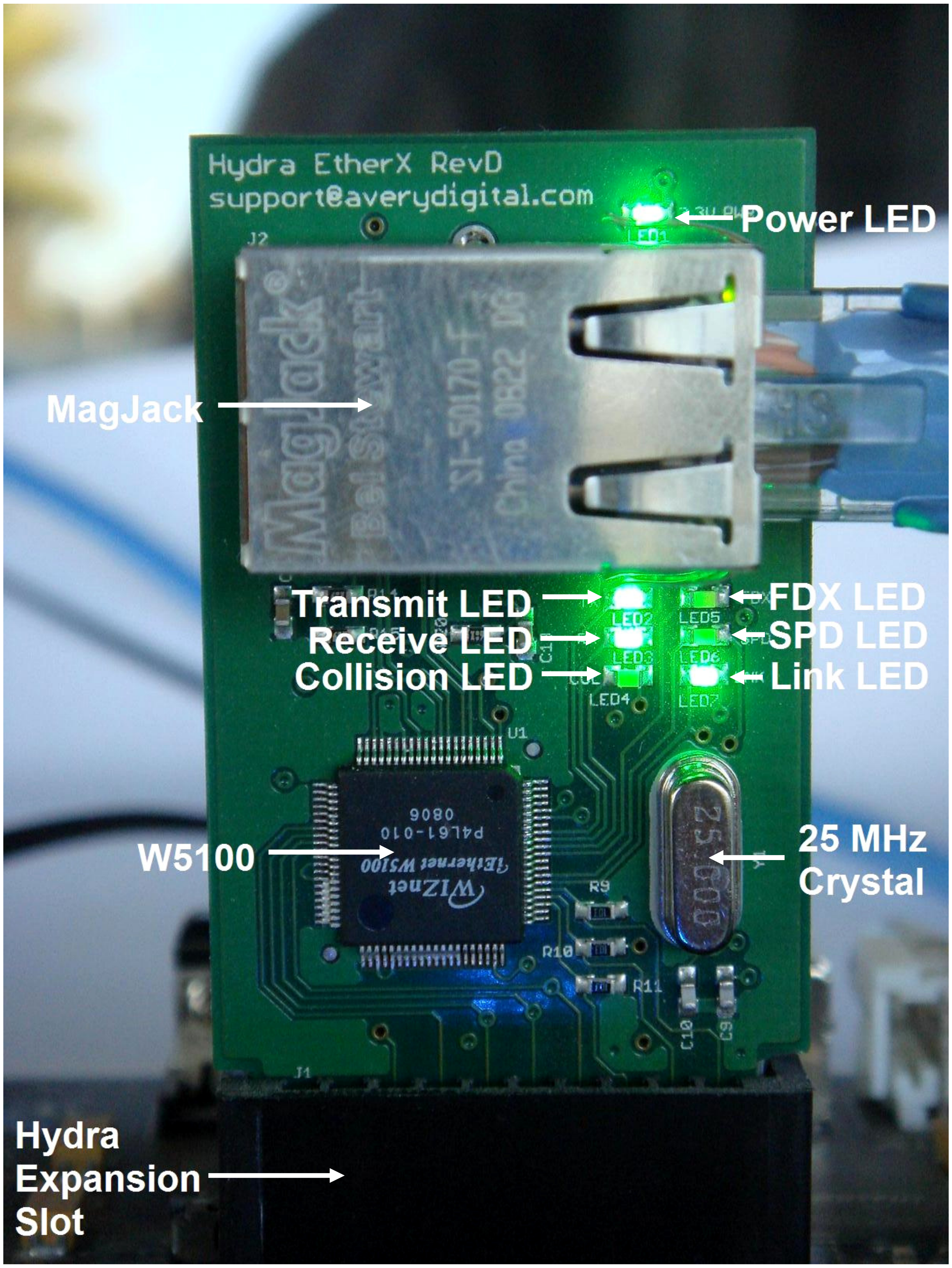
- The HYDRA EtherX Card itself.
- CD-ROM with source, tools, API, etc. (not shown).
- Printed Quick Start Sheet (not shown).

2.1 CD-ROM Contents

The CD-ROM contains the documentation, drivers, demos, and all source code for the EtherX. The CD-ROM layout is as follows:

```
CD_ROM:\    autorun.inf
            LICENSE.TXT
            CodeExamples\
                    \Ping
                    \SimpleTCPClient
                    \SimpleTCPServer
                    \SimpleUDP
                    \SquishKevin
                    \TicTacToe
                    \SimpleWebServer
            DataSheets\
            Documentation\
            Downloads\
            Schematic\
            WhitePapers\

NOTE: "CD_ROM" is your CD-ROM drive letter; "D:", "E:", etc.
```

Figure 2.0 – Annotated close up of the HYDRA EtherX card.



3.0 Introduction and Quick Start

A close-up of the **HYDRA EtherX Card** shown in Figure 2.0. The heart of the card is the W5100 chip from Wiznet. It also contains a MagJack connector which contains the magnetics needed for Ethernet signals, some discretes (resistors, capacitors, etc.), and LEDs.

The card has 7 LED indicators:

- Power
- RX
- TX
- FullDuplex
- Collision
- Speed
- Link

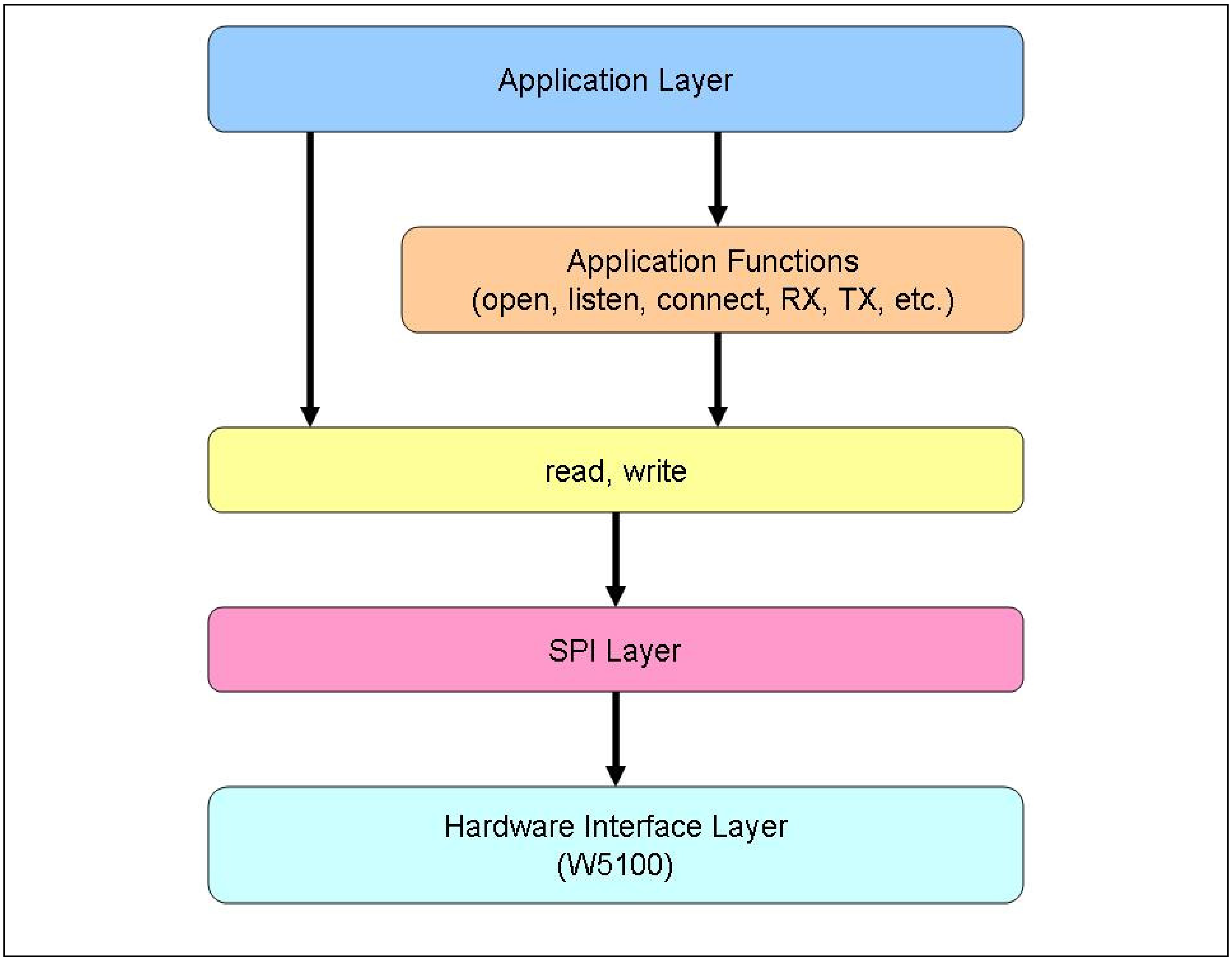
The power LED is located in the upper right portion of the card and should be on at all times. The RX LED will turn on when the W5100 is receiving Ethernet data. The TX LED will turn on when the W5100 is transmitting Ethernet data. The FullDuplex LED will turn on when the W5100 is in full duplex mode (default). The Collision LED will turn on when the W5100 detects a collision on the Ethernet line (common for HUBs). The Speed LED will turn on when the transfer rate is 100Mbps (default is 10Mbps). The Link LED will turn on when the W5100's internal PLLs have locked to the Ethernet signal and an electrical link has been established with the Ethernet device the EtherX card in plugged into.

The W5100 has no internal power on reset (POR) so the reset pin (active low) is pulled down via a resistor. This means at power up the W5100 is automatically reset and the Hydra will need to raise the pin high to bring the device out of reset. In addition there are three mode pins that determine the speed and full/half duplex of the device. These are pulled up/down via resistors on the W5100 to make the device operate at 10Mbps with full duplex. The Hydra can change these to make the device operate differently.

NOTE

The W5100 can operate at 100Mbps. However, during testing it was found that the device could become unstable (unusable). This appears to be a defect in the W5100 itself not the EtherX card. This is why the default configuration is 10Mbps. The performance gained from 100Mbps is very minuscule as the actual SPI bandwidth between the Hydra and the EtherX card is so much smaller than 10Mbps. Since all Ethernet devices are backward compatible to 10Mbps, there is no harm leaving the EtherX card in the default configuration. You can definitely change the speed to 100Mbps but BEWARE!!!

Figure 3.0 – A simplified illustration of the EtherX software model.



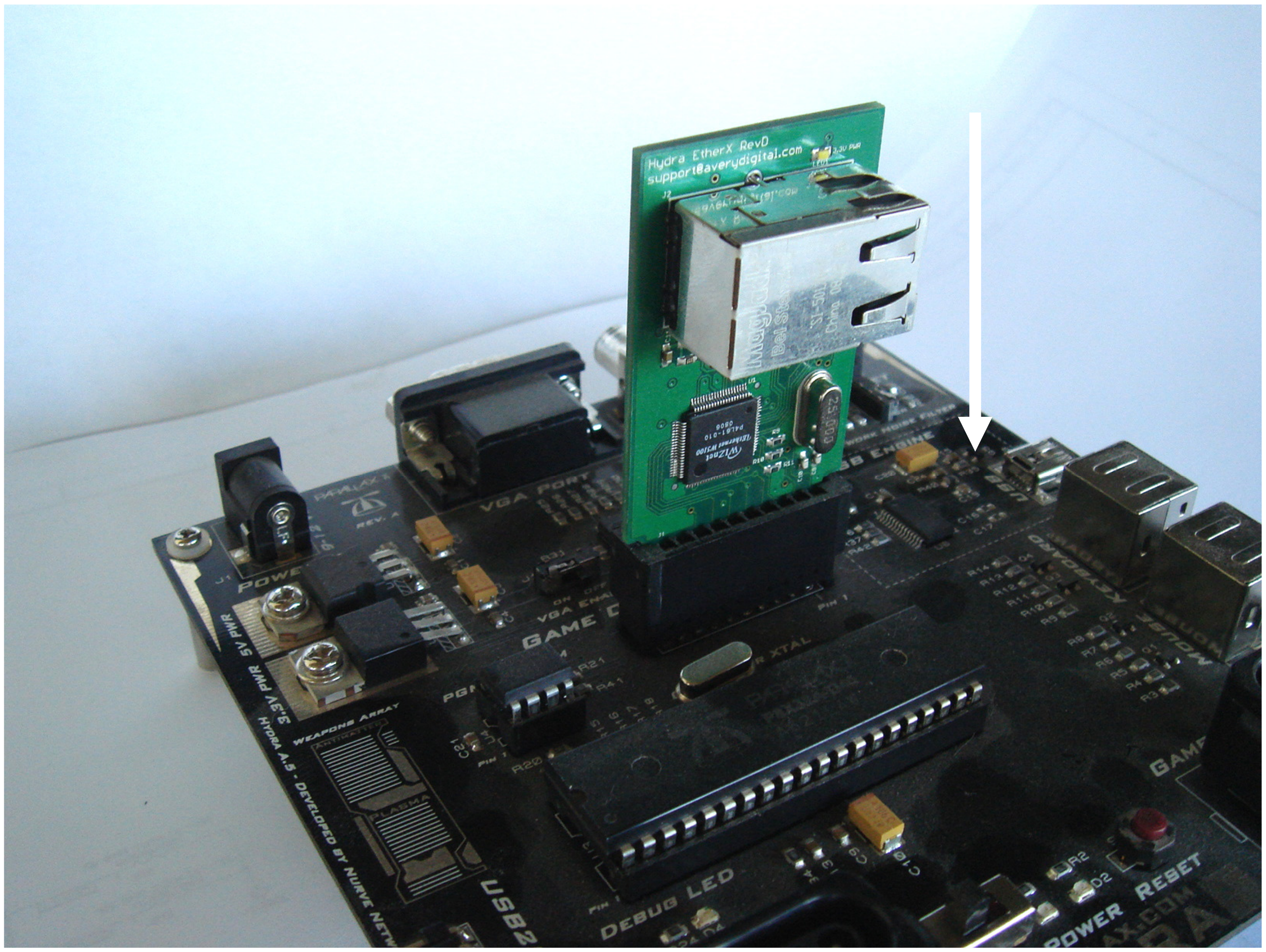
The software interface to the EtherX card is a layered model as shown in Figure 3.0. At the lowest level there is a **SPI driver** which does nothing except send and receive bytes to and from the W5100. It is written in assembly and it designed to fit nicely into one COG. The application layer has no direct access to this layer.

The next level contains two spin functions, read and write, which will access all the internal registers of the W5100. The application layer can use these functions to configure, monitor, or use the W5100 anyway it wishes.

The last level contains the spin functions that most application layer programs will use. It uses the second level spin functions to configure the W5100, open ports, and transceive data.

The EtherX card can be used for all kinds of applications from web servers to two player games over the internet. You could even write a program on the PC that sends data to the Hydra when needed providing essentially unlimited storage to the Hydra (handy for loading a multiple levels for a game when needed for example).

Figure 4.0 – Inserting the HYDRA EtherX card into the HYDRA.



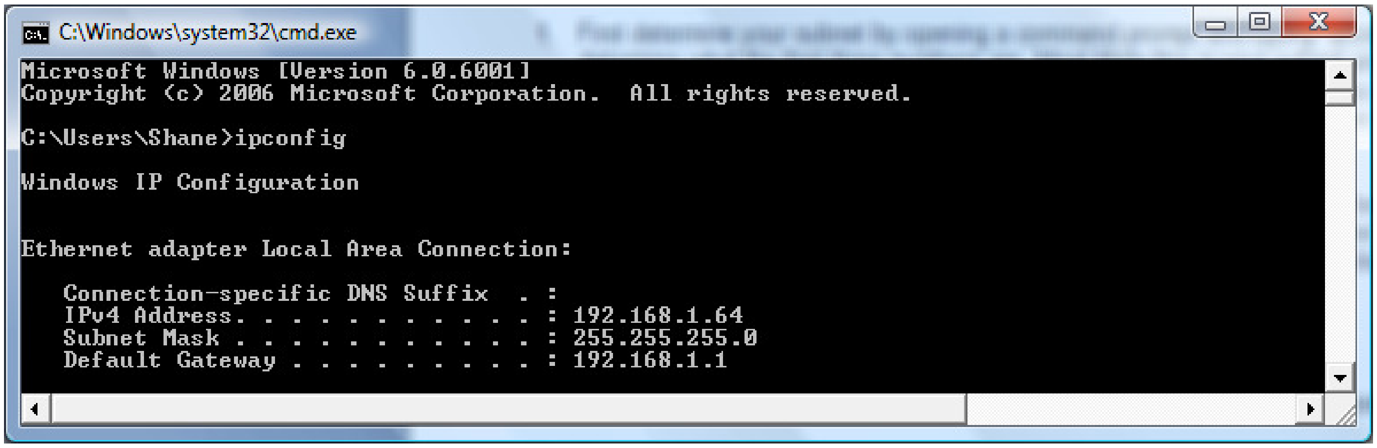
3.1 Quick Start Guide

This quick start guide will guide you through loading a simple program into the Hydra that will allow you to ping the EtherX card from your PC. The important thing here is that if you can ping the EtherX card then you can be certain that the EtherX is working and interfaced your hub/switch/router properly. A YouTube clip to supplement this quick start which demonstrates each of these steps can be found at:

http://www.youtube.com/v/7SZKPr7Lbww

1. First determine your subnet by opening a command prompt and typing "ipconfig". You should see something similar to figure 5.0. Look at your IP address and determine what the first three numbers are. Most likely this is your subnet and the IP address you assign the HydraEtherX should have the same first three numbers. For example, if you had an IP address of 192.168.1.64 then the subnet would likely be 192.168.1 and when you assign an IP to the HydraEtherX card the first three numbers in the IP address need to be 192.168.1.

Figure 5.0 – Command prompt showing ipconfig.



```
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\Users\Shane>ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . :
   IPv4 Address. . . . . . . . . . . : 192.168.1.64
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

2. With the HYDRA hooked up to power, TV and the PC, simply insert the EtherX card into the expansion slot facing the front of HYDRA as shown in Figure 4.0, be careful not to force it. Connect the EtherX card to your hub/switch/router via Ethernet cable. Make sure you have the game controller in the ***left*** controller port of the HYDRA. The HYDRA should be off.

3. Once everything is plugged in, turn the Hydra on.

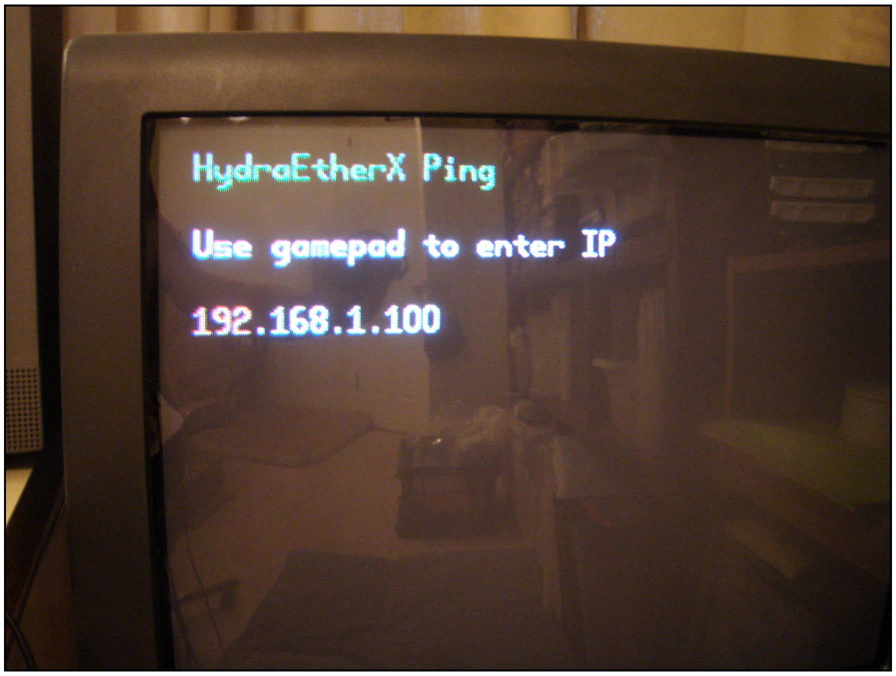
4. Open up the Propeller Tool on the PC and steer to the directory on the CD that contains the Ping code. It is located at:

   CD_ROM:\CodeExamples\Ping\

5. Double click on **Ping.spin** and then go to **Run->Compiler Current->Load RAM** in the Propeller Tool.

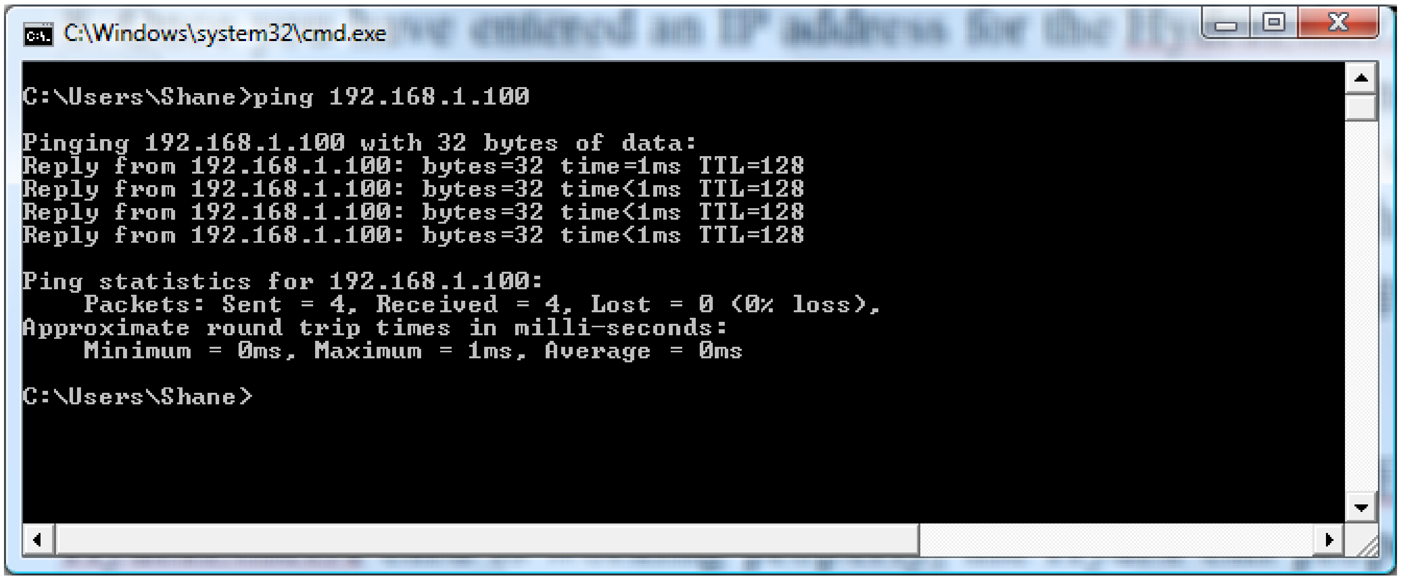
6. Once loaded up the Hydra will display a title screen saying "HydraEtherX Ping". The Hydra will then wait for the HydraEtherX device to come up and will then allow you to enter an IP address for the HydraEtherX using the game pad. You should see Figure 6.0 on the TV.

Figure 6.0 – TV screen grab of the Ping program



7. Once you have entered an IP address for the HydraEtherX on the game pad, you can ping the HydraEtherX using your PC. Open a command prompt and ping the HydraEtherX using the IP address you specified with the game pad. For example if you specified IP address **192.168.1.100** you would type "**ping 192.168.1.100**" in the command prompt. You should get back a reply from the HydraEtherX and it should look something like Figure 7.0. If you instead get something that says **"Destination Host unreachable**" then re-check the subnet, IP addresses, and your router configuration.

Figure 7.0 – Ping reply.



```
C:\Users\Shane>ping 192.168.1.100

Pinging 192.168.1.100 with 32 bytes of data:
Reply from 192.168.1.100: bytes=32 time=1ms TTL=128
Reply from 192.168.1.100: bytes=32 time<1ms TTL=128
Reply from 192.168.1.100: bytes=32 time<1ms TTL=128
Reply from 192.168.1.100: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.1.100:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 1ms, Average = 0ms

C:\Users\Shane>
```

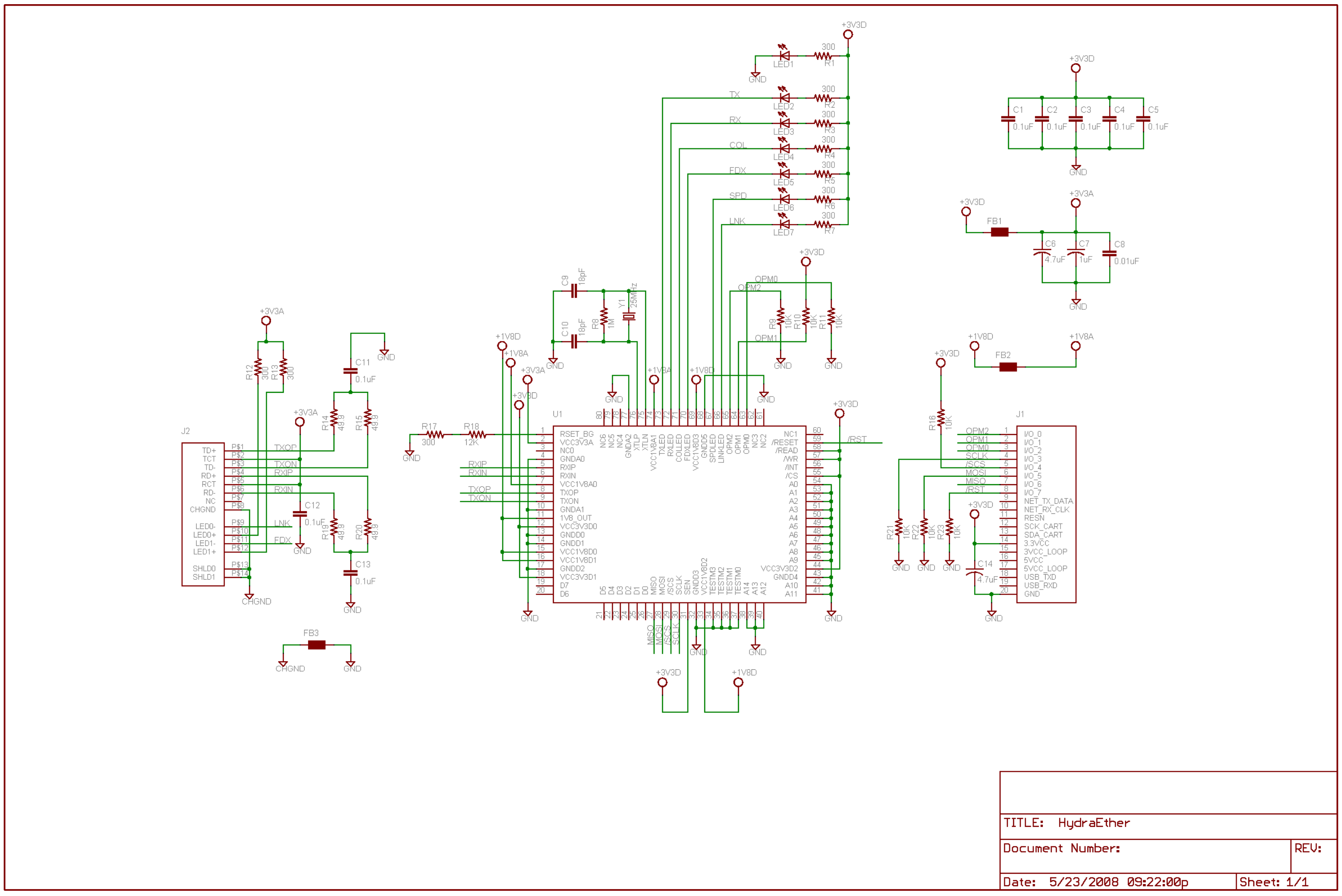
4.0 Circuit Design and Hydra Electrical Interface

The HYDRA EtherX card design may not look easy at first but don't let it intimidate you. When it all boils down there is an SPI interface from the Hydra which connects directly to the W5100 and Ethernet signals from the W5100 to the MagJack connector. Once you get your head wrapped around that the rest is the easy stuff like LEDs, a crystal, etc. We'll start with the circuit design and break it down to help you understand the design.

4.1 Circuit Design

The circuit is based on the W5100 reference schematic provided by Wiznet. It is included in the CD under the datasheet directory. Figure 8.0 shows the schematic of the EtherX card.

Figure 8.0 – W5100 schematic.



The heart of the whole design of course is the W5100. The chip itself has its own on chip voltage regulator to regulate the 1.8V voltage that it needs. Pin 11 is the voltage output for 1.8V and it needs to get connected to the 1.8V power pins on the chip. The digital 1.8V power pins (pins 15, 16, 33, and 69) are connected directly to pin 11. The analog 1.8V pins needs to be filtered from the digital supply so that switching noise from the digital portion of the chip won't find its way into the analog portion of the chip.

This is accomplished simply using a ferrite bead, FB2, which effectively is an inductor (inductors are used to filter out current spikes in the supply). These beads are selected based on how much current is passed through them, what their resistance is when there is no switching (DC), and what the resistance is to particular frequencies. The larger the DC current through the device the lower the voltage will drop over it. How much the voltage will drop depends not only on the current that passes through but also how much DC resistance the bead has. An ideal ferrite bead would have zero DC resistance and infinite resistance and the switching frequency you care to filter. Obviously these don't exist so when you are designing such power filters you need to consider how much current runs through the bead, how much voltage will drop over the bead (need to know how much a voltage drop you can withstand), and what frequency you want to filter out.

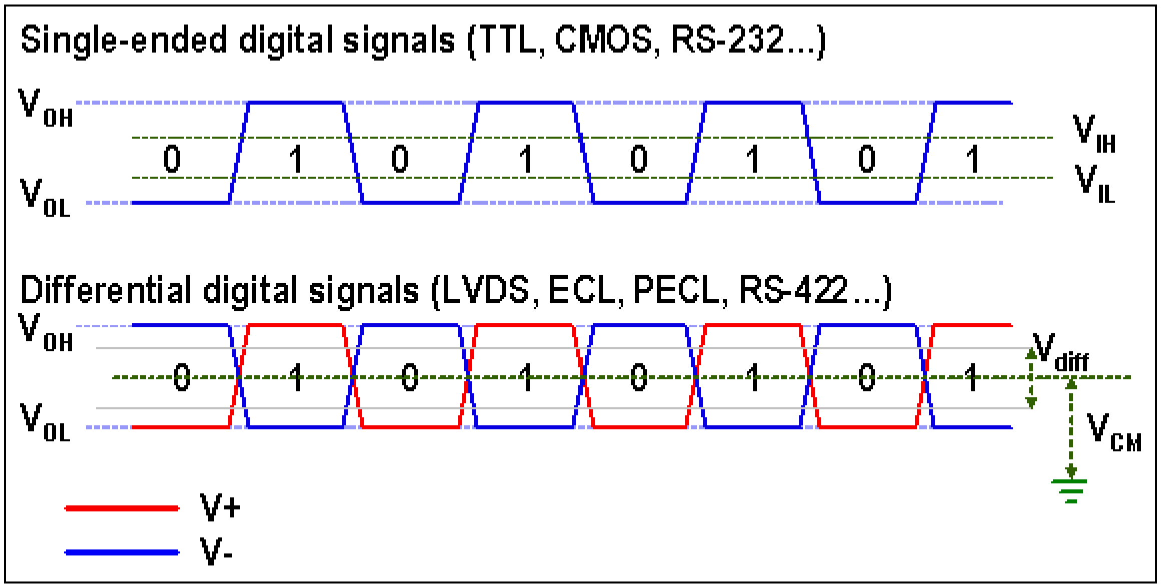
Similarly, we filter the power 3.3V analog power supply. The 3.3V supply that is supplied from the Hydra is a digital supply that contains the all the switching noise from not only the Hydra but the W5100 as well. The analog supply needs to be clean from this noise and FB1, C6, C7, and C8 make up the filter network to clean this supply from that noise.

Last issue concerning power filtering, you also need to be sure that the ground is clean as well. Just as your voltage supply can have noise the ground signal can "bounce" and/or contain noise which is filtered with FB3.

The W5100 connects to the MagJack via two differential pairs. The MagJack contains a magnetic phy needed for all Ethernet devices. Ethernet is magnetically coupled. This means that there is no physical connection between Ethernet devices. Instead there are small transformers that magnetically induce current in one another. The physics of this have to do with how inductors work. When a changing current (called alternating current or AC) passes through an inductor a magnetic field is created. This goes both ways. When an inductor is placed in a magnetic field it will produce a changing current. When you pass AC current through an inductor and place another inductor very close to it (but not connected) you get an AC current through the second inductor as well and this is called a transformer. The MagJack contains these transformers for us so we don't have to worry about it.

All the MagJack needs are the transmit and receive pairs that will go to these transformers. Differential pairs are interesting signals that are used to transfer data at very high rates. Single ended signals have a practical limit to how fast they can change from high to low or low to high. You can pick many signal integrity books and read your heart out to understand the physics of PCB design but the bottom line is that there is a limit to how fast you can go. Differential signals use two signals to send one bit of information. When one signal is a high voltage the other signal is low voltage. The differential receiver simply looks at the difference between the two signals to determine if it should be a logic one or logic zero. Figure 9.0 graphically shows the difference between single ended and differential signals.

Figure 9.0 – Single ended vs. differential signals.


Single-ended digital signals (TTL, CMOS, RS-232...)
V_{OH}
V_{OL}
V_{IH}
V_{IL}
0 1 0 1 0 1 0 1
Differential digital signals (LVDS, ECL, PECL, RS-422...)
V_{OH}
V_{OL}
0 1 0 1 0 1 0 1
V_{diff}
V_{CM}
V+
V-

These signals are run on the PCB very close together so that any noise that is picked on one differential signal is picked up on the other and thus the difference is always exactly the same. USB, firewire, SATA, and of course Ethernet are all examples of differential signals. The RX and TX signals that go from the W5100 to the MagJack are differential signals that are impedance matched to 100 ohms. This just means that the driver strength exactly matches what the receiver expects to see and is imperative for not only differential signals but any high speed signal.

You can communicate with the W5100 via standard address/data bus interface or SPI. There are 15 bits for the address bus and 8 bits for the data bus. Obviously, there are not enough pins to interface to the Hydra over the expansion connector. Therefore, we use an SPI interface. Since we don't need the address or data bus we ground the address pins, ground the bus control signals, and leave the data bus unconnected. To configure the device for SPI communication we need to tie the SPI enable pin (pin 31) to 3.3V.

Configuring the W5100 mode is accomplished with the OPMODE pins (W5100 pins 63, 64, and 65). These pins specify whether to device is full duplex or half duplex and whether the device is 10Mbps or 100Mbps. In testing it was found that the W5100 can become unstable at 100Mbps and since the SPI transfer rate between the Hydra and the W5100 is so much slower than 10Mbps the performance gain from making the W5100 operate at 100Mbps is minuscule. All Ethernet devices are backward compatible to 10Mbps there is no harm in configuring the W5100 at 10Mbps. Thus the default configuration for the W5100 is for full duplex and 10Mbps. This is accomplished by pulling down OPMODE0 and OPMODE2 and pulling up OPMODE1. However, the OPMODE pins are routed to the Hydra expansion connector for you change the configuration if you wish. Be sure that the OPMODE pins are set to what you want before you take the W5100 out of reset.

The reset pin is pulled down so that the W5100 is powered up in reset by default. This is because the W5100 doesn't include an internal power on reset and needs to be reset after power up. The Hydra is then responsible for taking the W5100 out of reset.

The only other thing to discuss about the circuit design is the LEDs. The W5100 provides signals that indicate certain states of the device (TX, RX, collision, duplex, speed, and link). These signals are then routed on the PCB to LEDs to light up to show these states. A current limiting LED is placed in series with these LEDs to limit how much current is passed through them. These LEDs are super bright. Watch your eyes!!

4.2 Hydra Electrical Interface

Clearly the EtherX card connects the Hydra via the expansion connector. Many of the signals on the expansion connector are unused. In fact, the only signals used are the eight IO pins and the power and ground. A table of the connections is shown below in table 1.0.

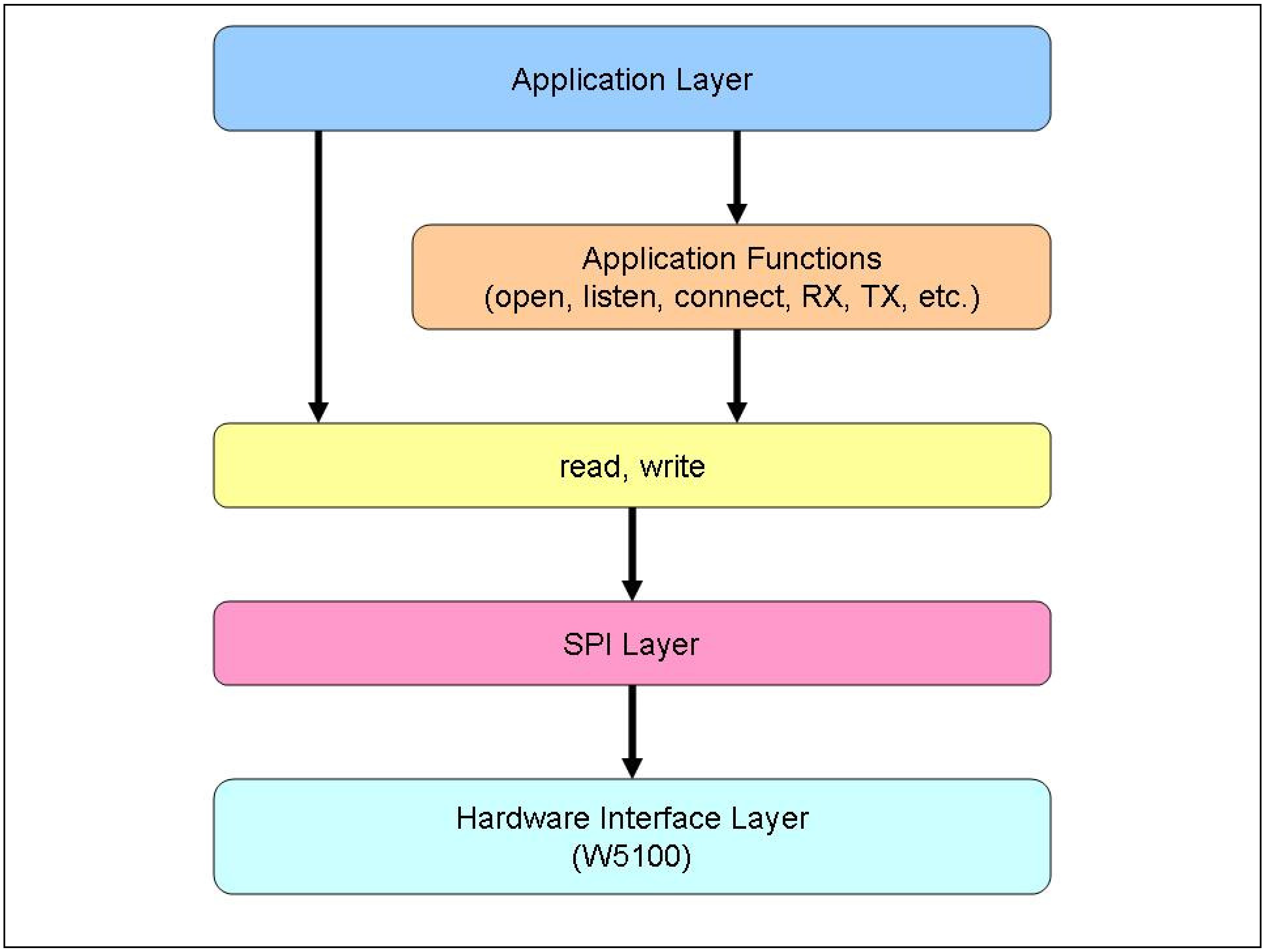
Table 1.0 – EtherX interface to Hydra expansion slot.

EtherX Card Pin	Hydra Pin	Function
I/O_0 Pin 1	P16/Pin #21	OPMODE 2 – Selects W5100 mode
I/O_1 Pin 2	P17/Pin #22	OPMODE 1 – Selects W5100 mode
I/O_2 Pin 3	P18/Pin #23	OPMODE 0 – Selects W5100 mode
I/O_3 Pin 4	P19/Pin #24	SCLK – SPI Clock
I/O_4 Pin 5	P20/Pin #25	SCS – SPI Chip Select (active low)
I/O_5 Pin 6	P21/Pin #26	MOSI – SPI Master out/Slave in
I/O_6 Pin 7	P22/Pin #27	MISO – SPI Master in/Slave out
I/O_7 Pin 8	P23/Pin #28	RST – W5100 Reset (active low)
3.3V Pin 14	POWER	3.3V – Power
GND Pin 20	POWER	Ground

The remaining pins on the Hydra expansion card are left unconnected. The SPI pins are pulled up/down as needed so that the pins will not drive to an unknown value before the Hydra sets the appropriate pins to output.

5.0 Interfacing to the HYDRA EtherX Card

Figure 10.0 - A simplified illustration of the EtherX software model.



In this section we'll discuss interfacing to the EtherX. Referring to Figure 10.0 above you can see that there is the **application layer** at the top. Then under it are the application functions. These are spin functions that are part of the W5100 driver which includes functions that are most commonly used by any application. Most application need only call these functions to interface to the W5100. The next layer is the **read**/**write** layer which are spin functions in the W5100 driver that properly communicate with the W5100 using the SPI address/data structure. This is available to the application layer so that the user application can directly communicate to the W5100 if needed. The last layer is the **SPI layer** which is written in assembly and lives in a separate COG. It simply sends or receives a byte to/from the W5100.

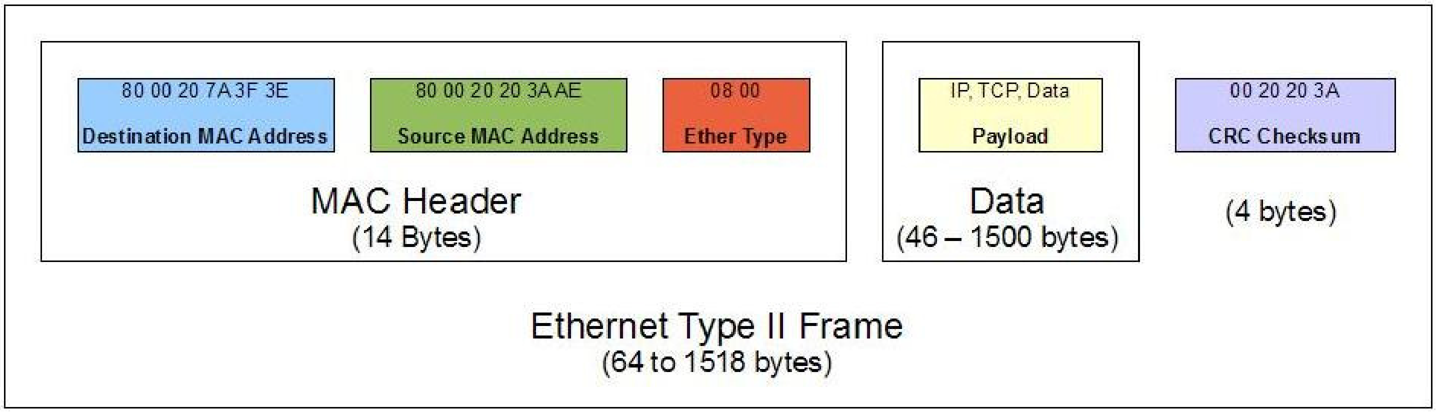
We will begin with a short explanation of Ethernet, internet protocols, and their history.

5.1 Ethernet and Internet Overview & History

Before we begin talking about the W5100 and how to use it we need a good understanding of basic networks. The W5100 is an Ethernet device. Ethernet was developed by Xerox PARC in the 70's as an implementation for a local area network and was standardized by the IEEE. In theory Ethernet can pass any kind of data the user would want.

Every Ethernet device must have a unique address called a MAC address. You may also hear this referred to as the physical address. A valid Ethernet frame contains a destination MAC, source MAC, a type, then the user data, and lastly a checksum. Figure 11.0 illustrates this.

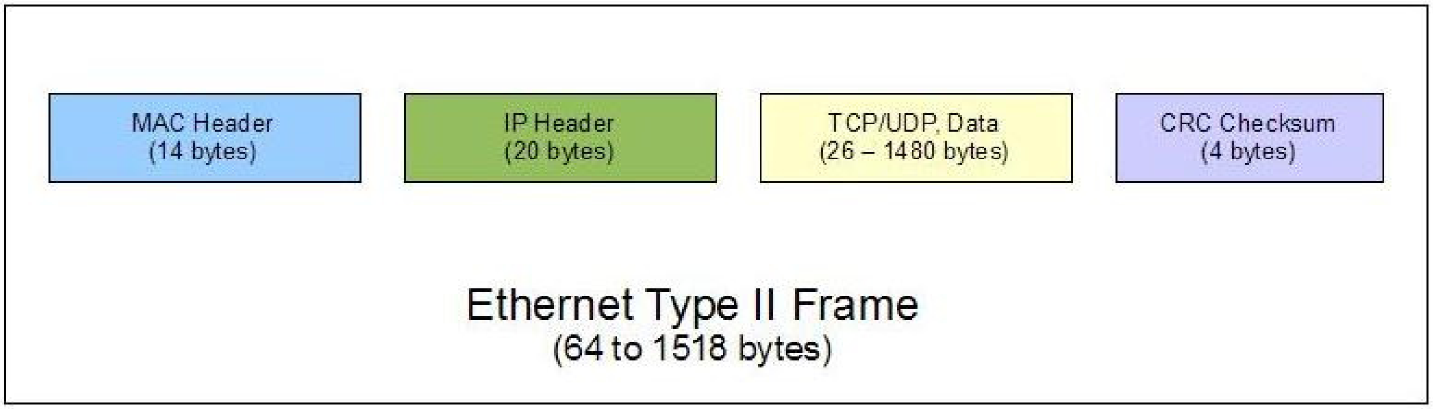
Figure 11.0 – Ethernet frame.



The W5100 is smart enough to fill a lot of this out for us. We need to give it a MAC address and it will either compute or learn the rest.

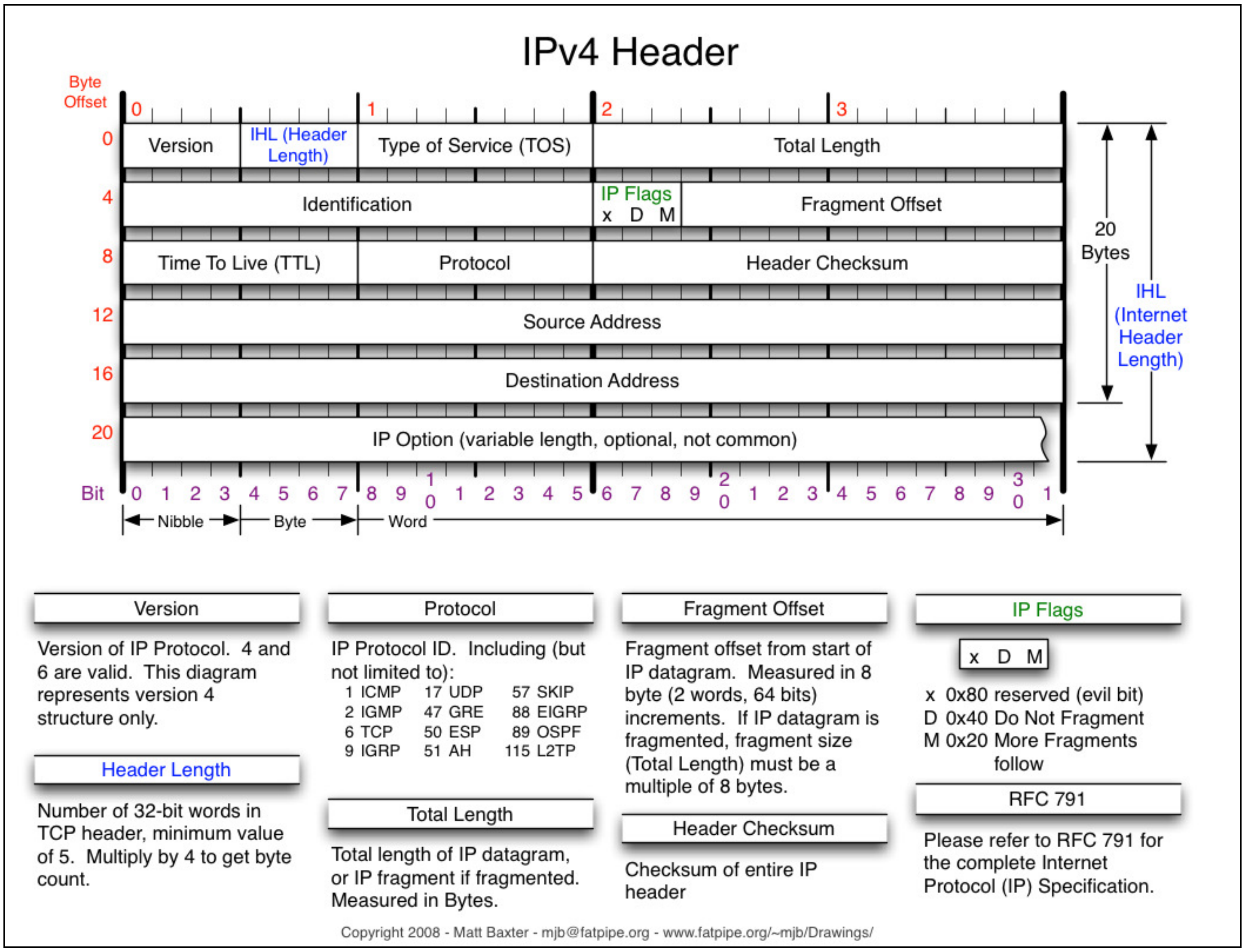
The payload data is then filled with an IP header (along with other data). The IP header contains an IP address. Every computer connected to the Internet has a unique IP address (not strictly true like a MAC address but we will make this statement for now). The IP address is sort of like a phone number. Computers called routers have an idea (or exactly know) where an IP address is located on the Internet. An IP header contains a lot of information but the two fields we care about most is the source and destination IP addresses. The W5100 is smart enough to fill everything else out for us. We need to assign the W5100 an IP address and tell it what IP address we want to send to. So now we have an Ethernet MAC header followed by an IP header that looks like Figure 12.0.

Figure 12.0 – Ethernet frame with IP and TCP/UDP headers.



Where the IP header looks like Figure 13.0:

Figure 13.0 – IP header fields.



Version

Version of IP Protocol. 4 and 6 are valid. This diagram represents version 4 structure only.

Header Length

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

Protocol

IP Protocol ID. Including (but not limited to):

1 ICMP	17 UDP	57 SKIP
2 IGMP	47 GRE	88 EIGRP
6 TCP	50 ESP	89 OSPF
9 IGRP	51 AH	115 L2TP

Total Length

Total length of IP datagram, or IP fragment if fragmented. Measured in Bytes.

Fragment Offset

Fragment offset from start of IP datagram. Measured in 8 byte (2 words, 64 bits) increments. If IP datagram is fragmented, fragment size (Total Length) must be a multiple of 8 bytes.

Header Checksum

Checksum of entire IP header

IP Flags

x D M

x 0x80 reserved (evil bit)
D 0x40 Do Not Fragment
M 0x20 More Fragments follow

RFC 791

Please refer to RFC 791 for the complete Internet Protocol (IP) Specification.

Now we know where to go but we need some more information about what to do with the data. Think of your computer right now. Lots of Internet data comes and goes. There's your web browser, mail client, IM, BitTorrent, etc. How does your computer know that a piece of data it received from the Internet came for your web browser and not your email. The answer is a number called a port number. Your web browser works on a specific port, your email on a different port and so on. Another header is added to help tell Internet devices about the port number and other information and this layer is a TCP or UDP header. As a side note the next layer doesn't need to be TCP or UDP. It could be something else such as ARP, DNS, or PING.

What's the difference between TCP and UDP? Both have port information but TCP guarantees that the data will get there. That means if a packet of data gets lost it will try to send it again. Also, it provides flow control meaning it will try to find the optimum speed to send the data (as fast as possible without loosing too much data). UDP simply sends the data and hopes for the best.

Which should you choose? Typically UDP is chosen when speed matters. It takes more time to verify that packets have arrived and resend when needed. Typically in UDP, data is constantly sent so if you loose a data packet it’s not that big a deal because you'll have another packet come along very soon. Halo would be an example of a UDP game. TCP based games would be used when data loss matter a lot more than time. A networked chess game is a good TCP example because you would only send the data when a player makes a move and that data must arrive. A good rule of thumb for games would be that real time games would use UDP and turn based games would use TCP.

Figure 14.0 shows the Ethernet frame now with the **TCP** header and the data. Figure 15.0 shows the **TCP** header fields.

Figure 14.0 - Ethernet frame with IP header, TCP headers, and data.

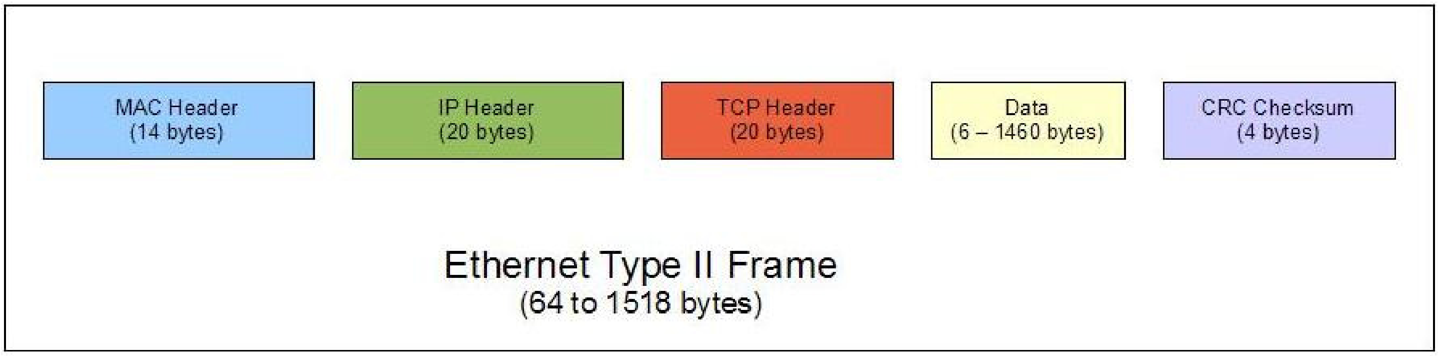
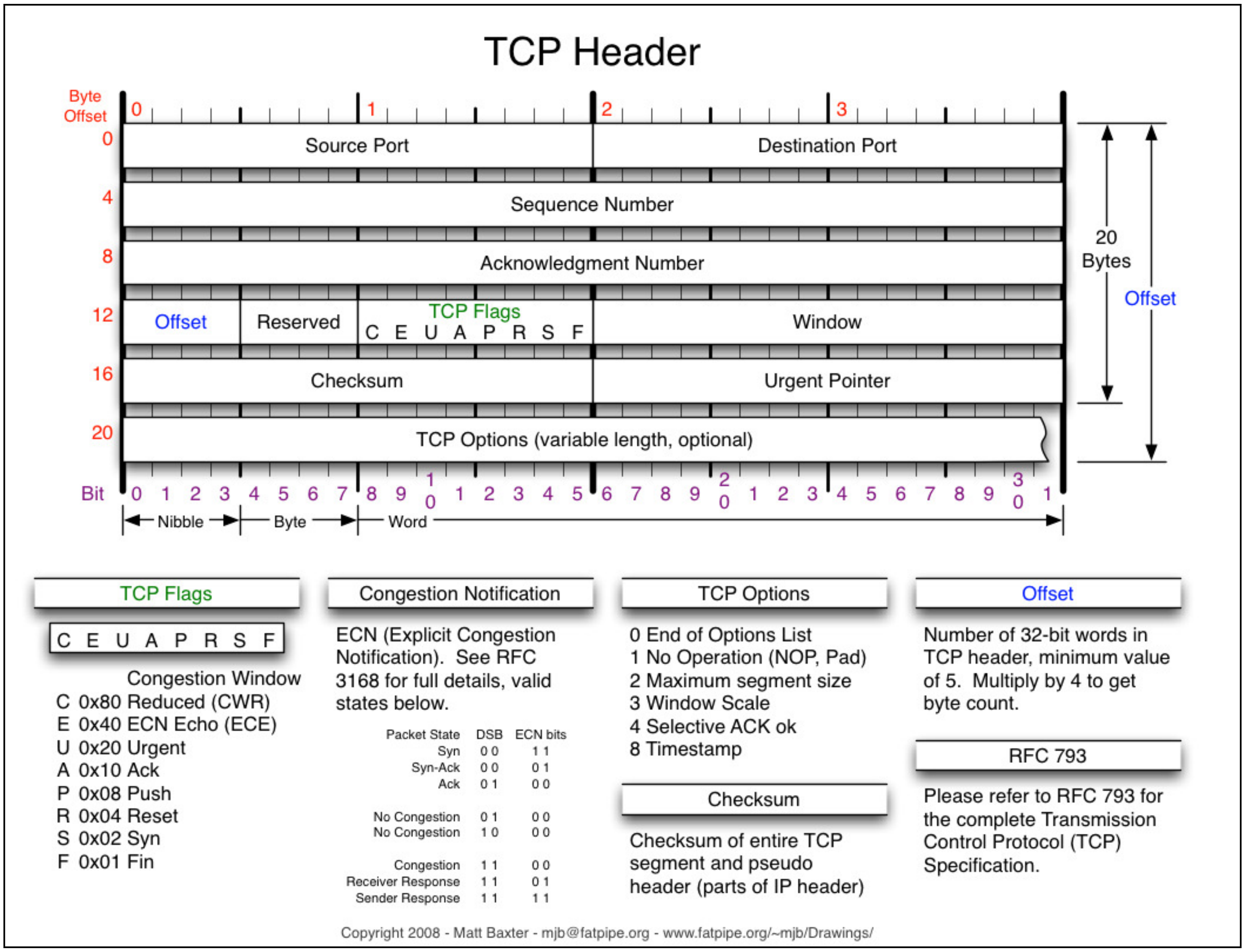


Figure 15.0 – TCP header fields.



TCP Flags

C E U A P R S F

		Congestion Window
C	0x80	Reduced (CWR)
E	0x40	ECN Echo (ECE)
U	0x20	Urgent
A	0x10	Ack
P	0x08	Push
R	0x04	Reset
S	0x02	Syn
F	0x01	Fin

Congestion Notification

ECN (Explicit Congestion Notification). See RFC 3168 for full details, valid states below.

Packet State	DSB	ECN bits
Syn	0 0	1 1
Syn-Ack	0 0	0 1
Ack	0 1	0 0
No Congestion	0 1	0 0
No Congestion	1 0	0 0
Congestion	1 1	0 0
Receiver Response	1 1	0 1
Sender Response	1 1	1 1

TCP Options

0 End of Options List
1 No Operation (NOP, Pad)
2 Maximum segment size
3 Window Scale
4 Selective ACK ok
8 Timestamp

Checksum

Checksum of entire TCP segment and pseudo header (parts of IP header)

Offset

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

RFC 793

Please refer to RFC 793 for the complete Transmission Control Protocol (TCP) Specification.

Copyright 2008 - Matt Baxter - mjb@fatpipe.org - www.fatpipe.org/~mjb/Drawings/

Figure 16.0 shows the Ethernet frame now with the **UDP** header and the data. Figure 17.0 shows the **UDP** header fields.

Figure 16.0 - Ethernet frame with IP header, UDP headers, and data.

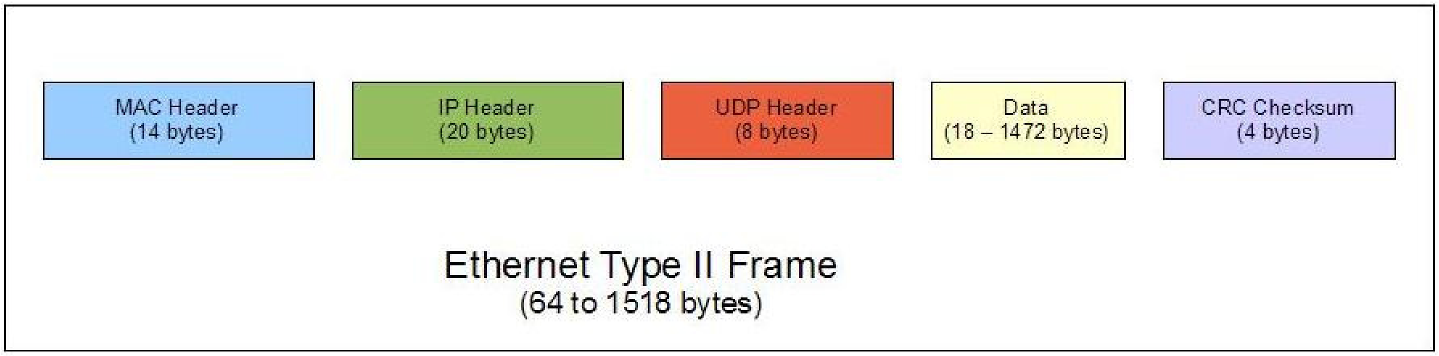
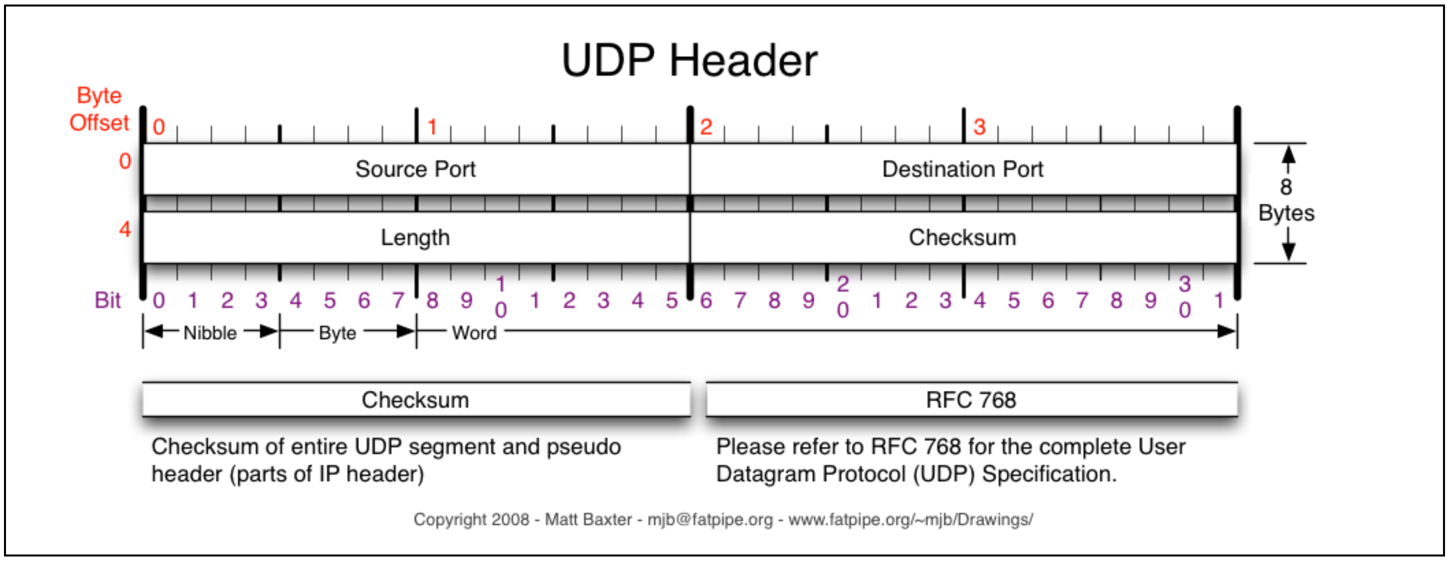


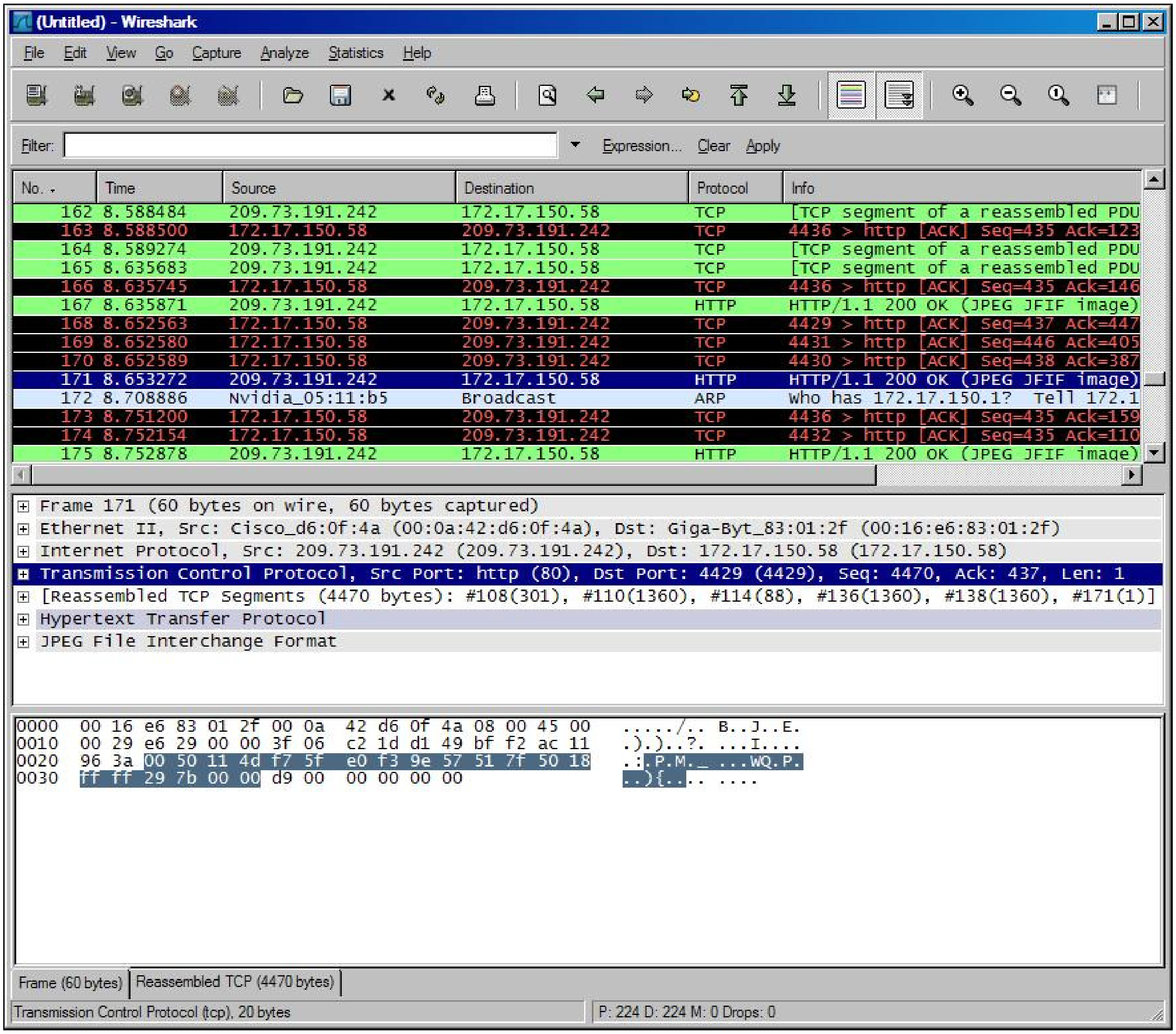
Figure 17.0 – UDP header fields.


UDP Header
Byte Offset
0
4
Source Port
Destination Port
Length
Checksum
8 Bytes
Bit
Nibble
Byte
Word
Checksum
Checksum of entire UDP segment and pseudo header (parts of IP header)
RFC 768
Please refer to RFC 768 for the complete User Datagram Protocol (UDP) Specification.
Copyright 2008 - Matt Baxter - mjb@fatpipe.org - www.fatpipe.org/~mjb/Drawings/

As you can see there is more information in these headers than what we described above but fortunately for us the W5100 handles this for us.

The best way to learn about the networking fields is to actually see them. Wireshark is a free program that not only acts as a packet sniffer that grab every bit of Internet traffic coming and going on your computer but graphically breaks down each header for inspection. Below is a screen grab of Wireshark capturing data on a request to a web server. The upper pane shows the packets that have been grabbed (it shows these in real time as they come into the computer). The middle pane then shows a header breakdown of the packet. The lower pane shows a complete hex dump of the entire data. Clicking on a header in the middle pane will highlight the corresponding data in the hex dump. In the screen capture below the TCP header is highlighted. Try Wireshark for yourself. It is included in the CD under the downloads directory. Figure 18.0 shows a screen grab of Wireshark in action.

Figure 18.0 – Screen Capture of Wireshark


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No.  Time      Source          Destination     Protocol  Info
162  8.588484  209.73.191.242  172.17.150.58   TCP       [TCP segment of a reassembled PDU
163  8.588500  172.17.150.58   209.73.191.242  TCP       4436 > http [ACK] Seq=435 Ack=123
164  8.589274  209.73.191.242  172.17.150.58   TCP       [TCP segment of a reassembled PDU
165  8.635683  209.73.191.242  172.17.150.58   TCP       [TCP segment of a reassembled PDU
166  8.635745  172.17.150.58   209.73.191.242  TCP       4436 > http [ACK] Seq=435 Ack=146
167  8.635871  209.73.191.242  172.17.150.58   HTTP      HTTP/1.1 200 OK (JPEG JFIF image)
168  8.652563  172.17.150.58   209.73.191.242  TCP       4429 > http [ACK] Seq=437 Ack=447
169  8.652580  172.17.150.58   209.73.191.242  TCP       4431 > http [ACK] Seq=446 Ack=405
170  8.652589  172.17.150.58   209.73.191.242  TCP       4430 > http [ACK] Seq=438 Ack=387
171  8.653272  209.73.191.242  172.17.150.58   HTTP      HTTP/1.1 200 OK (JPEG JFIF image)
172  8.708886  Nvidia_05:11:b5 Broadcast       ARP       Who has 172.17.150.1?  Tell 172.1
173  8.751200  172.17.150.58   209.73.191.242  TCP       4436 > http [ACK] Seq=435 Ack=159
174  8.752154  172.17.150.58   209.73.191.242  TCP       4432 > http [ACK] Seq=435 Ack=110
175  8.752878  209.73.191.242  172.17.150.58   HTTP      HTTP/1.1 200 OK (JPEG JFIF image)

⊞ Frame 171 (60 bytes on wire, 60 bytes captured)
⊞ Ethernet II, Src: Cisco_d6:0f:4a (00:0a:42:d6:0f:4a), Dst: Giga-Byt_83:01:2f (00:16:e6:83:01:2f)
⊞ Internet Protocol, Src: 209.73.191.242 (209.73.191.242), Dst: 172.17.150.58 (172.17.150.58)
⊞ Transmission Control Protocol, Src Port: http (80), Dst Port: 4429 (4429), Seq: 4470, Ack: 437, Len: 1
⊞ [Reassembled TCP Segments (4470 bytes): #108(301), #110(1360), #114(88), #136(1360), #138(1360), #171(1)]
⊞ Hypertext Transfer Protocol
⊞ JPEG File Interchange Format

0000  00 16 e6 83 01 2f 00 0a  42 d6 0f 4a 08 00 45 00   ...../.. B..J..E.
0010  00 29 e6 29 00 00 3f 06  c2 1d d1 49 bf f2 ac 11   .).)..?. ...I....
0020  96 3a 00 50 11 4d f7 5f  e0 f3 9e 57 51 7f 50 18   .:.P.M._ ...WQ.P.
0030  ff ff 29 7b 00 00 d9 00  00 00 00 00               ..){.... ....

Frame (60 bytes) | Reassembled TCP (4470 bytes)
Transmission Control Protocol (tcp), 20 bytes     P: 224 D: 224 M: 0 Drops: 0
```

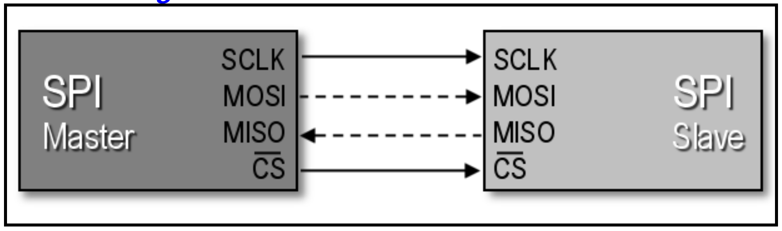
5.2 SPI Bus Basics

SPI stands for **Serial Peripheral Interface** originally developed by **Motorola**. Its one of two very popular modern serial standards including **I^{2}C** which stands for **Inter Integrated Circuit** by **Phillips**. SPI unlike I^{2}C (which has no separate clock) is a clocked synchronous serial protocol that supports full duplex communication. However I^{2}C only takes **2 wires** and a ground. Where SPI needs **3 wires**, ground, and potentially chip select lines to enable the slave devices. But, SPI is much faster, so in many cases speed wins over and the extra clock line is warranted. The advantage of I^{2}C is that you can potentially hook hundreds of I^{2}C devices on the same 2-bus lines since I^{2}C devices have addresses that they respond to. SPI bus protocol on the other hand requires that every SPI slave has a chip select line.

Figure 19.0 shows a simple diagram between a **master** (left) and a **slave** (right) SPI device and the signals between them which are:

- **SCLK** - Serial Clock (output from master).
- **MOSI/SIMO** - Master Output, Slave Input (output from master).
- **MISO/SOMI** - Master Input, Slave Output (output from slave).
- **SS** - Slave Select (active low; output from master).

Figure 19.0 – The SPI electrical interface.



Note: You might find some devices with slightly different naming conventions, but the idea is there is data out and data in, a clock, and some kind of chip select.

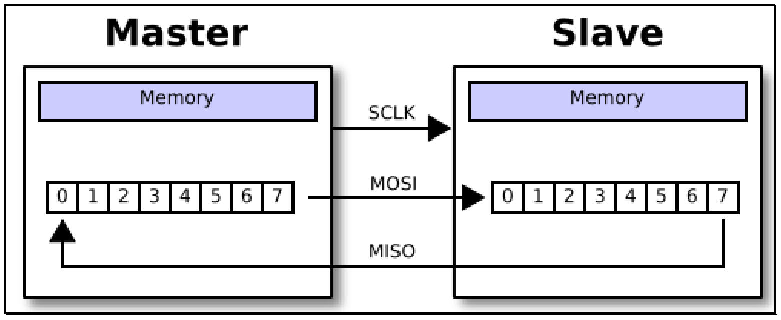
If you look at the EtherX interface signals then you will see that the EtherX card signals have slightly different names. Assuming the HYDRA is the Master and the EtherX card is the slave, the mapping of SPI signals are shown in Table 2.0 below:

Table 2.0 – Mapping of EtherX card signals to official SPI signal names.

Card Pin	EtherX Name	SPI Signal Name	Function
5	SCS	SS	Chip select active low.
6	MOSI	MOSI	Master out, slave in.
4	SCLK	SCLK	Clock signal.
7	MISO	MISO	Master in, slave out.
Note: *Power and ground signals of course need to be connected as well.*			

SPI is very fast since not only is it clocked, but it's a simultaneous ***full duplex*** protocol which means that as you clock data out of the master into the slave, data is clocked from the slave into the master. This is facilitated by a transmit and receive bit buffer that constantly re-circulates as shown in Figure 20.0.

Figure 20.0 – Circular SPI buffers.



The use of the circular buffers means that you can send and receive a byte in only 8 clocks rather than clocking out 8-bits to send, then clocking in 8-bits to receive. Of course, in some cases the data clocked out or in is “dummy” data, meaning when you write data and you are ***not*** expecting a result the data you clock in is garbage and you can throw it away. Likewise when you do a SPI read, typically you would put a $00 or $FF in the transmit buffer as dummy data since something has to be sent and it might as well be predictable.

Sending bytes with SPI is similar to the serial RS-232 protocol, you place a bit of information on the transmit line, then strobe the clock line (of course RS-232 has no clock). As you do this, you also need to read the receive line since data is being transmitted in both directions. This is simple enough, but SPI protocol has some very specific details attached to it about ***when*** signals should be read and written that is, on the rising or falling edge of the clock as well as the polarity of the clock signal. This way there is no confusion about edge, level, or phase of the signals. These various modes of operation are logical called the SPI mode and are listed in Table 3.0 below:

Table 3.0 – SPI clocking modes.

Mode #	CPOL (Clock Polarity)	CPHA (Clock Phase)
0	0	0
1	0	1
2	1	0
3	1	1

Mode Descriptions

Mode 0 – The clock is **active** when **HIGH**. Data is **read** on the **rising edge** of the clock. Data is **written** on the **falling edge** of the clock (default mode for most SPI applications).

Mode 1 – The clock is **active** when **HIGH**. Data is **read** on the **falling edge** of the clock. Data is **written** on the **rising edge** of the clock.

Mode 2 – The clock is **active** when **LOW**. Data is **read** on the **rising edge** of the clock. Data is **written** on the **falling edge** of the clock.

Mode 3 – The clock is **active** when **LOW**. Data is **read** on the **falling edge** of the clock. Data is **written** on the **rising edge** of the clock.

Note: Most SPI slaves **default to mode 0**, so typically this mode is what is used to initiate communications with a SPI device.

Figure 21.0(a) – SPI timing diagrams for clock phase polarity (CPHA=0).

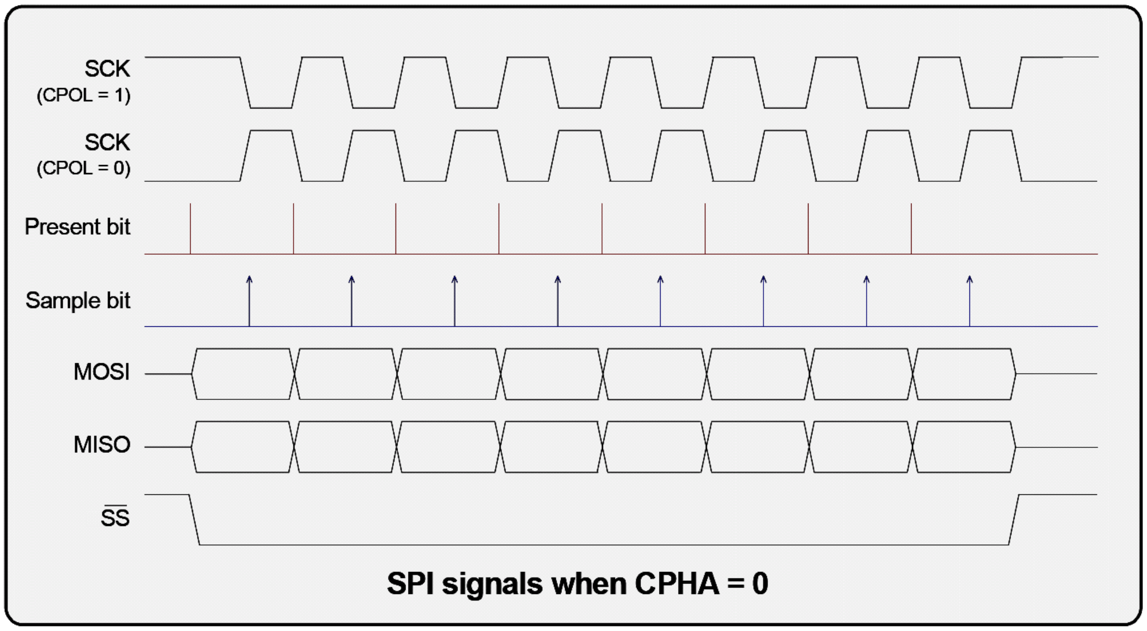
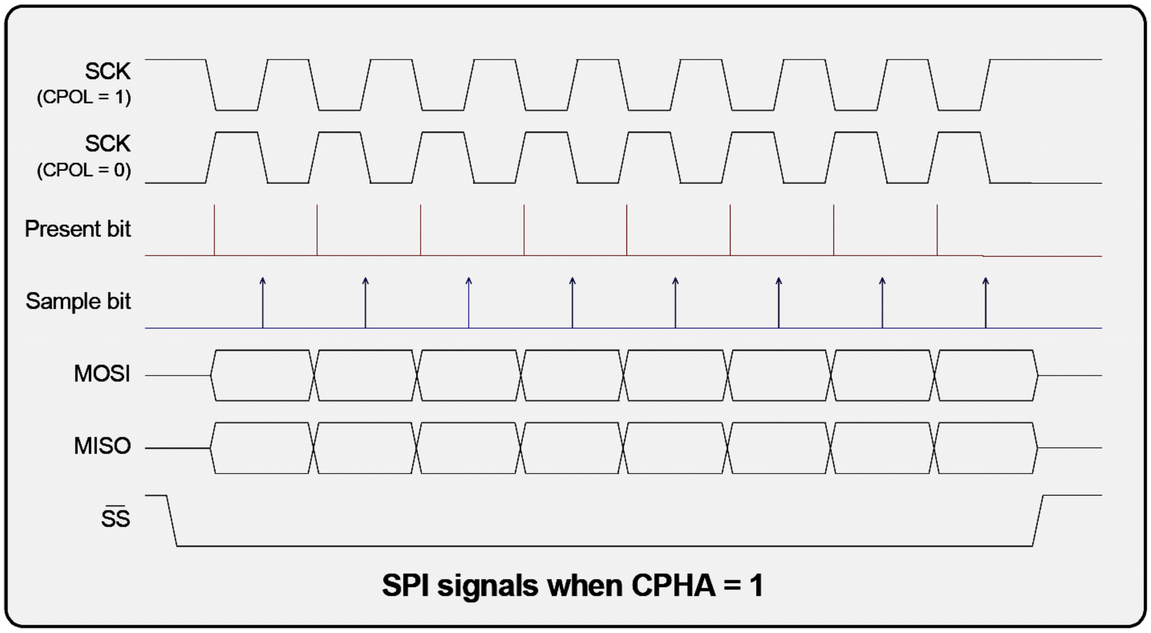


Figure 21.0(b) – SPI timing diagrams for clock phase polarity (CPHA=1).



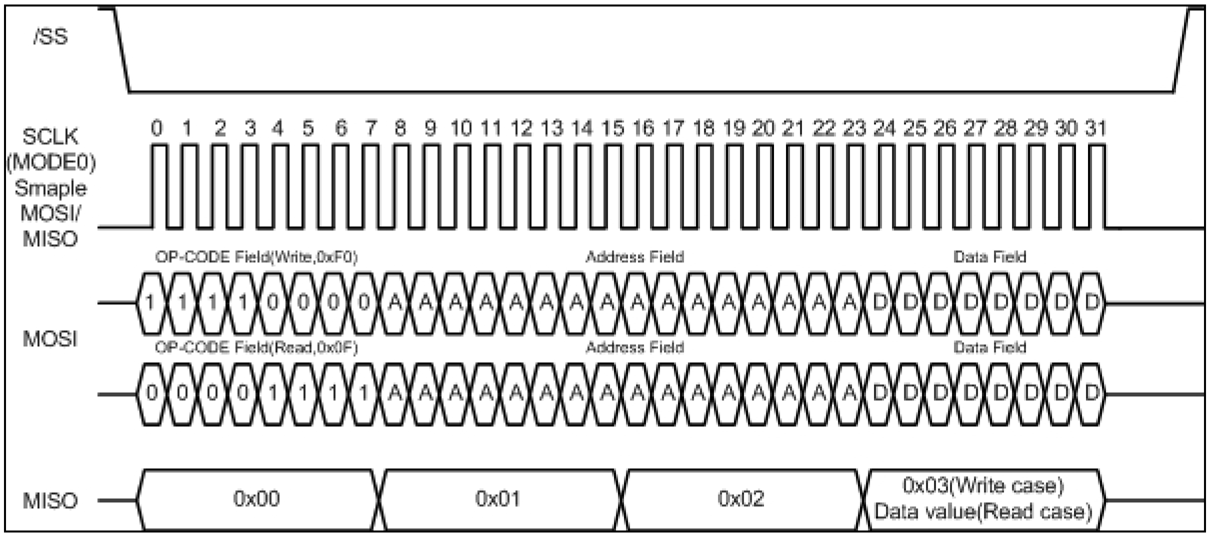
Notice that there is no timing related logic in the function. Since SPI protocol is totally **synchronous** the master in charge of the clock can run the clock as fast (to maximum speed) or slow (static if desired). Also, notice the data is sent out most significant bit (msb) to least significant bit (lsb).

For the EtherX card we will operate in SPI Mode 0. This is the only mode that the W5100 operates in. The HYDRA has no built in support for SPI so we must bit bang the protocol ourselves. The bit banging of the SPI protocol lives as assembly code that fits completely in a COG.

5.3 EtherX Communications Protocol

Communication with the W5100 is exclusively on a register addressed basis. This means that any read or write to the W5100 will be to a register that has an address. All registers will have a 16 bit address and 8 bit data. Every read or write to the W5100 will be four bytes long. The first byte is the command (read or write), the next two bytes are the address, and the last byte is the data. Writing a byte from the Hydra to the W5100 on the MOSI line will only be valid for a write. During a read the final byte that you need to read will be on the MISO line. The SPI format is shown in Figure 22.0.

Figure 22.0 – Format of EtherX card commands in SPI mode.

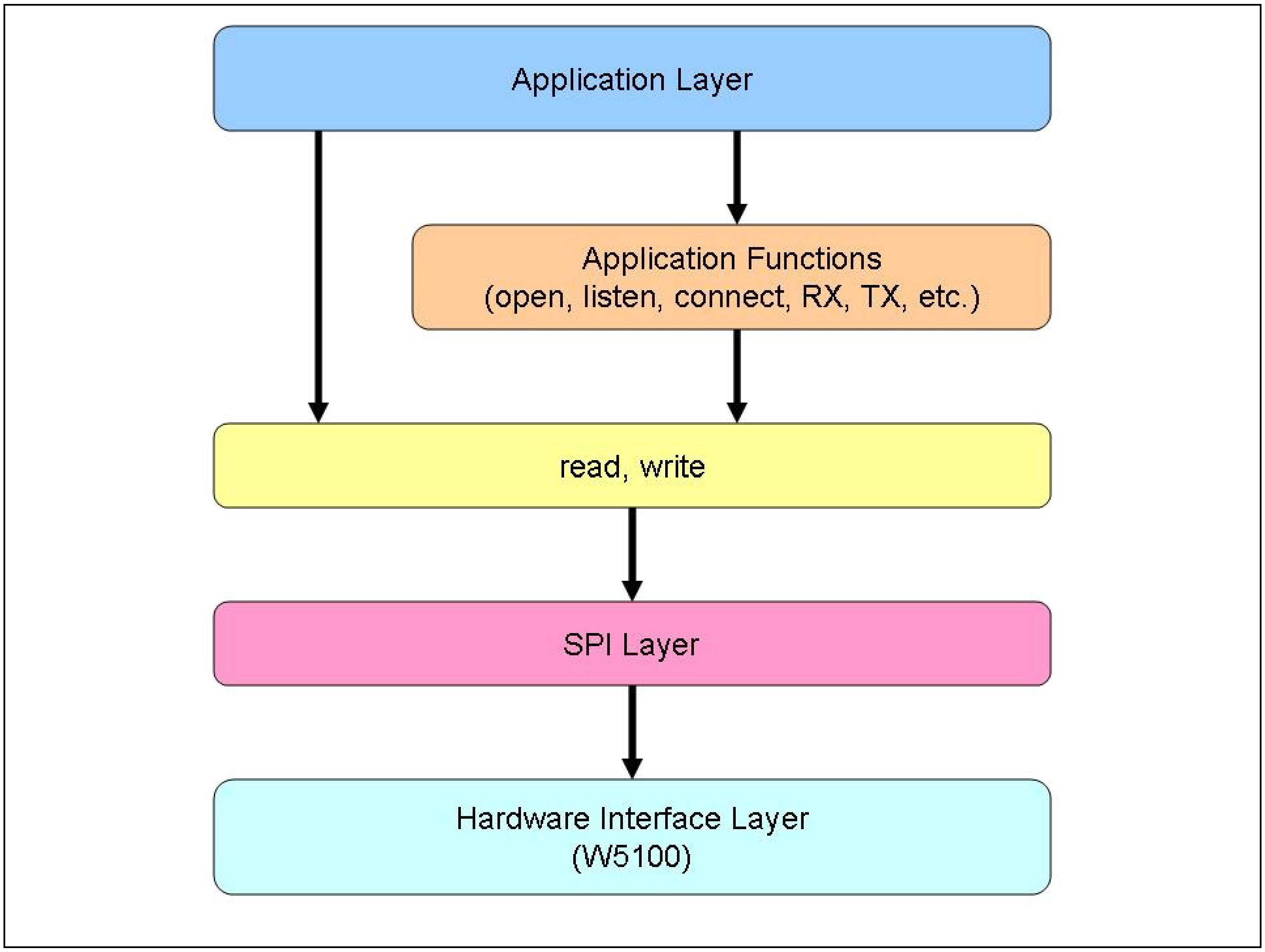

/SS
SCLK (MODE0) Smaple MOSI/ MISO
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
OP-CODE Field(Write,0xF0)
Address Field
Data Field
1 1 1 1 0 0 0 0 A A A A A A A A A A A A A A A A D D D D D D D D
MOSI
OP-CODE Field(Read,0x0F)
Address Field
Data Field
0 0 0 0 1 1 1 1 A A A A A A A A A A A A A A A A D D D D D D D D
MISO
0x00
0x01
0x02
0x03(Write case) Data value(Read case)

The 16-bit address must be formatted in ***Big Endian*** format, that is high byte to low byte. This is the opposite of the Intel and Propeller format which are ***Little Endian*** machines. Therefore if you want to send $FF_AA in Big Endian then you would send $FF, followed by $AA to complete the address.

6.0 Unleashing the EtherX Driver API

The EtherX driver is a starting point for you to begin experimenting with Ethernet and the W5100. A copy of the driver is located in every directory in the CodeExamples directory of the CD. It is also located on the Avery Digital website at www.averydigital.com/priducts.html. Figure 23 shows the overall driver architecture and its relationship to the application and hardware.

Figure 23.0 – A simplified illustration of the EtherX software model.



As you can see from the figure between the application and the EtherX card there are three levels. We will discuss these three levels now in detail in next sections. These sections will show all the API function calls and explain how to use them. Table 4.0 shows the master API listing.

Table 4.0 – Master API function call reference.

Start and Stop Functions

start – Starts the SPI Layer assembly code in a new COG.
stop – Stops and frees the COG containing the SPI Layer assembly code.

Read and Write Functions

read (a0, a1) – Reads a byte from address a0,a1 from the W5100 via SPI.
write (a0, a1, dout) – Writes byte dout at address a0,a1 to the W5100 via SPI.

Application Functions

init (gway, subnet, ip, mac) – Initialize the W5100's gateway, subnet, IP, and MAC.
open (type, source_port, dest_port, dest_ip) – Open socket with socket type, ports, and destination IP.
close – Close socket.
listen – Use when TCP server to listen for connection from client.
connect – Use when TCP client to establish connection with server.
con_est – Use to determine if a TCP connection has been established.
TX (dataptr, size) – Transmits size bytes from main memory pointed to by dataptr to the W5100.
RX (dataptr, size, block) – Receives size bytes from W5100 to main memory pointed to by dataptr.
read_rsr – Returns the number of bytes the W5100 has stored in its internal receive buffer.

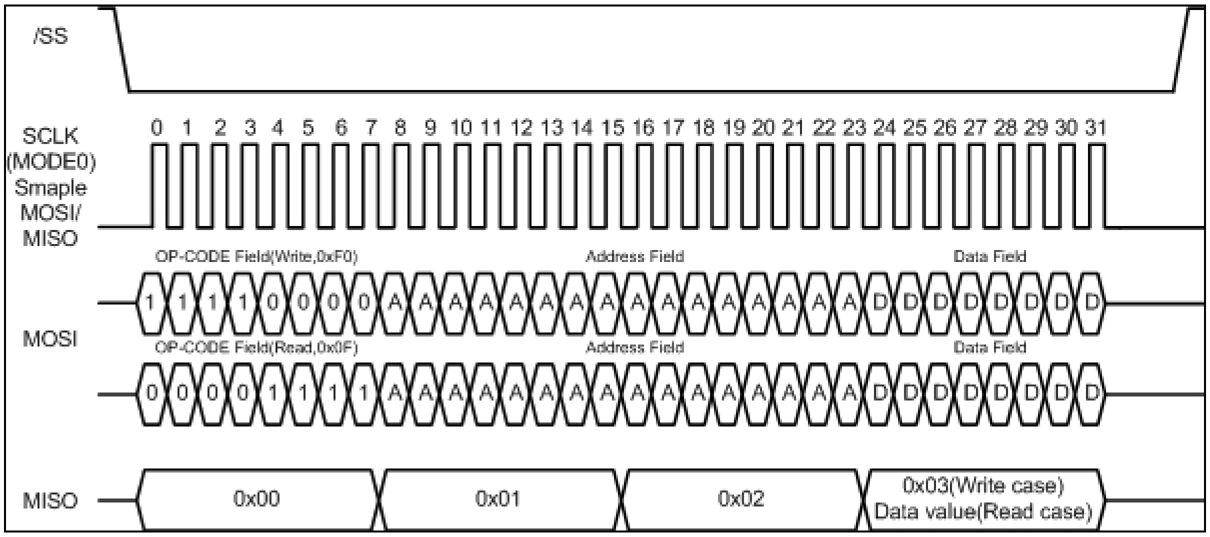
Mode Function

mode(opmode) – Change the operating mode of the W5100.

6.1 EtherX Driver – SPI Layer

The lowest level of the driver is the SPI layer. We'll discuss in detail here how it works but all details aside all it really does is send and receive bytes using SPI. The SPI layer is written completely in assembly and is designed to live wholly in a COG with its 396 bytes. Figure 24 shows the format that data is sent to the W5100.

Figure 24.0 – Format of EtherX card commands in SPI mode.


/SS
SCLK (MODE0) Smaple MOSI/ MISO
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
OP-CODE Field(Write,0xF0)
Address Field
Data Field
1 1 1 1 0 0 0 0 A A A A A A A A A A A A A A A A D D D D D D D D
MOSI
OP-CODE Field(Read,0x0F)
Address Field
Data Field
0 0 0 0 1 1 1 1 A A A A A A A A A A A A A A A A D D D D D D D D
MISO
0x00
0x01
0x02
0x03(Write case) Data value(Read case)

The first thing that the SPI layer does is set the correct pins to input or output as needed to communicate SPI with the W5100. Once that is done we take the W5100 out of reset. Since there is no power up reset on the W5100, the device needs to be reset at power up. We take care of this by pulling down the reset pin of the W5100 which will hold the W5100 in reset when the device is turned on. It is up to the Hydra to take the W5100 out of reset by setting the reset line to the W5100 to logic high. After that the SPI layer looks at global variables to determine what to do next.

There are five global variables that are used to interface the SPIN functions of the driver to the SPI layer. Since the variables are global they reside in main memory and are accessible by all COGs including the one that our SPI layer is located in. The first global variable is the **SPIRW** variable. The main loop of the SPI layer will constantly look at this variable to determine if it should take any action. As long as the variable is equal to zero then it will do nothing. When the variable is equal to one then the SPI layer will read a byte from the W5100 on the EtherX card. This will cause the SPI layer to send a 0x0F as a first byte to the W5100 to indicate a read operation (see figure 24). When the variable is equal to two then the SPI layer will write a byte to the W5100 on the EtherX card. This will cause the SPI layer to send a 0xF0 as a first byte to the W5100 to indicate a write operation (see figure 24). After reading or writing a byte the SPI layer will clear this variable back to zero.

If you look at the assembly code the first part simply loops waiting for the **SPIRW** variable to not be zero. Once the variable is not zero it determines based on the **SPIRW** variable if the first nibble should be 0x0 (for a read) or 0xF (for a write). If the value should be 0x0 then it sets the MOSI line logic low and pulses the SCLK line four times. If the value should be 0xF then it set the MOSI line logic high and pulses the SCLK line four times. The assembly code then does the same thing for the second nibble (which should be 0xF for a read and 0x0 for a write). This would complete the first byte sent to the W5100 to indicate whether we will read or write a byte from the W5100. Note that most SPI interfaces allow for full duplex (read and write at the same time) but the W5100 command structure only allows one read or one write per SPI transaction thus making it half-duplex.

The next two variables, called **add0** and **add1**, are the address that the SPI layer will read form or write to. The interface to the W5100 is an addressable register interface. This means that every read or write will be to a register in the W5100 and every register has an address. As you can see in figure 24 there are two bytes needed to address the register you want to read or write. Whether the operation is a read or write the address portion of the transaction is exactly the same. Thus, after send the first byte to indicate whether the operation is a read or a write the SPI layer will read the two address variable and sends them to the W5100.

The assembly code to accomplish acts as shift register. It looks at the most significant bit of the address variable and sets the MOSI line logic high if the most significant bit of the address variable is high. Similarly, if the most significant bit of address variable is low then the MOSI is set low. We then pulse the SCLK line to clock in that bit. The SPI layer will then shift the address variable left one bit and repeat the process until the entire address variable is clock into the W5100. The assembly code will do this twice as there are two address variables.

NOTE	Using the term "most significant bit" here is a bit of a misnomer. Technically, the global variables are 32 bits. This means that the most significant bit is the 32nd bit of the variable. When we say "most significant bit" here we mean the most significant bit that we care about which is the 8th bit because we send data to the W5100 a byte at a time.

After the address is sent to the W5100 the SPI layer will shift out (just like the address variables) the value in the next global variable, called **dataout**, at the same time it shifts in the data from the MISO line to the global variable called **datain**. It does this no matter if the operation is a read or a write. In the case of a write **datain** will contain 0x03 as per the W5100 documentation. In the case of a read the MOSI line will output whatever data is located in the **dataout** which is fine since the W5100 will ignore whatever this value is during a read operation. Thus, will be read and write to the W5100 every SPI transaction not matter whether we are reading or writing. This keeps the driver small and simple to write. So, keep in mind that whenever you do a write operation to the W5100 the previous value of **datain** will be wiped out and now contains 0x03.

NOTE	Technically, the SPI layer will write the datain value to a local temporary value that will write the data to the global **datain** variable after the read and write operations are complete.

The last thing the SPI layer will do it set the **SPIRW** variable back to zero. The SPI layer was never intended to be directly accessible to the application layer. Although there is nothing stopping you from modifying it so that you can. You just need to make the global variables (**SPIRW, add0, add1, datain,** and **dataout**) visible to both the application and the SPI layer. Instead two small wrapper SPIN functions called **read** and **write** should be used to interface to the SPI layer and we will discuss these in the next section.

6.2 EtherX Driver- read and write

The two main functions for interfacing to the SPI layer are the **read** and **write** functions. These are two very simple wrapper SPIN functions that are easy to use and understand.

The **read** function in its entirety is shown below:

```
PUB read(a0, a1) : ret_val

  '' Call this function to read a byte from the W5100 via SPI.
  '' The arguments are two bytes that contains address byte 0
  '' and address byte 1. See W5100 data sheet for what registers
  '' these actually address. The function will return the byte
  '' that came via SPI from the W5100.

  'Read data from the address specified in the arguments
  add0 := a0                   'Set the arguments to the global
  add1 := a1                   'variables values
  SPIRW := SPI_RD              'Set SPIRW to the read value

  'Wait until the driver clears SPIRW. This indicates that it is done.
  repeat until SPIRW == SPI_DONE

  'The driver wrote the result to datain. Thus, that is the value
  'that we will return
  ret_val := datain
```

You can see that the **read** function has two input parameters, a0 and a1. These represent the address you want to read from. The **read** function will then set these address parameters to the global variables **add0** and **add1**. It will then set the **SPIRW** global variable to 1 which is equal to the constant variable **SPI_RD** (set in the CON section of the driver).
It then waits until the **SPIRW** global variable is set to 0 which is equal to the constant variable **SPI_DONE** (set in the CON section of the driver). Remember that the SPI layer will clear **SPIRW** global variable to 0 when it is done with the SPI transaction. Thus, this function will block (which means it will wait, doing nothing) until the SPI layer completes the transaction. It then returns the value that is stored in the **datain** global variable (which is where the SPI layer writes the result of the transaction).

The **write** function is just as simple as the read function is shown below:

```
PUB write(a0, a1, dout)

  '' Call this function to write a byte from the W5100 via SPI.
  '' The arguments are three bytes that contains address byte 0,
  '' address byte 1 and the data we wish to write. See W5100
  '' data sheet for what registers these actually address.
  '' Note that the datain global variable will be overwritten
  '' during this process.

  'Write data to the address specified in the arguments
  add0 := a0                   'Set the arguments to the global
  add1 := a1                   'variables values
  dataout := dout
  SPIRW := SPI_WR              'Set SPIRW to the write value

  'Wait until the driver clears SPIRW. This indicates that it is done.
  repeat until SPIRW == SPI_DONE
```

You can see that the **write** function has three input parameters, a0, a1, and dout. These represent the address you want to write to and the data would want to write. The **write** function will set the address parameters to the global variables **add0** and **add1**. It will then set the dout parameter to the **dataout** global variable. It will then set the **SPIRW** global variable to 2 which is equal to the constant variable **SPI_WR** (set in the CON section of the driver). It then waits until the **SPIRW** global variable is set to 0 which is equal to the constant variable **SPI_DONE** (set in the CON section of the driver). Remember that the SPI layer will clear **SPIRW** global variable to 0 when it is done with the SPI transaction. Thus, this function will block (which means it will wait, doing nothing) until the SPI layer completes the transaction.

That’s it! These functions are meant to be used by the application layer to read and write bytes to and from the W5100. If you wanted to you could completely use the Hydra EtherX card just using these two functions. In fact, if you wanted to configure or use the W5100 beyond what the Hydra EtherX driver provides then you would need to use these functions. In actuality all the other functions that are provided for the application layer in the EtherX driver use these two functions. We will now explore these other application functions in the next section.

6.3 EtherX Driver – Application Functions

While you could simply use **read** and **write** to use the EtherX there are still a lot of sticky details about what data you need to send and to what addresses to actually make the EtherX card useful. The application functions exist to provide a means for you to use the EtherX card without actually needing to know to gritty details of how the W5100 works. This is great to get an application running and learning about Ethernet and in the Internet in general but I encourage you to look at the functions and the W5100 documentation itself to understand what is going on.

6.3.1 Mode Function

One of the only application function that does not use the **read** and **write** functions is the **mode** function. This allows you to change the mode of the W5100 to change the speed and half/full duplex of the W5100. By default the W5100 operates at 10Mbps and at full duplex. Pull-up and pull-down resistors on the EtherX card set this so if you never call the **mode** function the card will power up and come out of reset as 10Mbps, full duplex. However, you can change this if you are so willing. Be careful though as the EtherX card can become unstable at 100Mbps. It is not tragic that the default of the EtherX operates at 10Mbps as the SPI interface to the W5100 is so much slower than 10Mbps you would never noticed the difference and all Ethernet devices are backward compatible to 10Mbps. But, if you wanted to operate the device at 100Mbps this is the function you would call to do so.

The **mode** function must be called before you start the EtherX driver. This is because the one of the first things the EtherX driver does is take the W5100 out of reset and once the W5100 is out of reset you shouldn't change the mode of the device. Thus, the **mode** function first checks to see if the EtherX driver has already been started and if so, it will do nothing. The function is shown below:

```
PUB mode(opmode) | temp

  '' Call this function with the value wanted for opmode before
  '' the start function is called. If the start function has
  '' already been called then this function will do nothing.

  if (cogon == 0)               'Be sure driver not already running
    DIRA |= $70000              'Set pins 16-18 to output
    temp := opmode << 16        'Shift our arg 16 bits left
    OUTA &= $FFF8FFFF           'Clear all outputs on pins 16-18
    OUTA |= opmode              'Set only required pins on pins 16-18
```

Functionally, all the **mode** function does is set the pins that map to the W5100 mode pins to out and set them according to the opmode input parameter. To make the W5100 100Mbps, full duplex call the mode function with opmode equal to 0.

6.3.2 Start and Stop

The only other two functions that don't use the **read** and **write** functions are the **start** and **stop** functions. You should recognize these as they are used in pretty much all drivers for the Hydra.

We'll discuss the **stop** function first as it the most simple. It simply stops the SPI layer and frees the COG it runs on. There, done!

The **start** function simply starts the SPI layer (written in assembly) which handles the actual bit banging of the SPI data to the W5100 in a new COG and returns the new COG number that the SPI layer lives in. If you choose to call the **mode** function will should be called before the **start** function. All other SPIN function calls (including **read** and **write**) should be called after the start function call been called.

Well that was rough. The code for **start** and **start** is shown below:

```
PUB start : okay

  '' This is the public start function. It starts
  '' a new cog at the assembly entry point
  'Start the SPI code in a new COG
  stop
  okay := cogon := (cog := cognew(@entry, @SPIRW)) > 0

PUB stop

'' Stops driver - frees a cog

  if cogon~
       cogstop(cog)
```

6.3.3 Init

The **init** function should be called right after the **start** function. It initializes the W5100 by telling it what our gateway, subnet, IP address, and MAC address are. The **init** function is shown below:

```
PUB init (gway, subnet, ip, mac)

  'Init the registers
  'Gateway
  write($00,$01,byte[gway])
  write($00,$02,byte[gway+1])
  write($00,$03,byte[gway+2])
  write($00,$04,byte[gway+3])

  'Subnet
  write($00,$05,byte[subnet])
  write($00,$06,byte[subnet+1])
  write($00,$07,byte[subnet+2])
  write($00,$08,byte[subnet+3])

  'MAC
  write($00,$09,byte[mac])
  write($00,$0a,byte[mac+1])
  write($00,$0b,byte[mac+2])
  write($00,$0c,byte[mac+3])
  write($00,$0d,byte[mac+4])
  write($00,$0e,byte[mac+5])

  'IP
  write($00,$0f,byte[ip])
  write($00,$10,byte[ip+1])
  write($00,$11,byte[ip+2])
  write($00,$12,byte[ip+3])
```

You can see that the function takes the four arguments to the **init** function, which are pointers to arrays, and writes their values to the corresponding register addresses of the W5100. Keep in mind that the arguments are array pointers and you would fill in the values on the arrays in your application before calling this function. Also, you need to make sure that the arrays are **byte** arrays, not the default 32 bit arrays. An example of calling this function from your application might look like the following from the SimpleTCPClient program that is included in the CodeExamples directory of the CD:

```
VAR
        byte  dip[4]
        byte  subnet[4]
        byte  ip[4]
        byte  gateway[4]
        byte  mac[8]
        byte  buffer[256]

OBJ

        term    : "tv_terminal_010.spin"     ' instantiate the terminal object
        eth     : "W5100_drv_011.spin"       ' instantiate the W5100 driver

PUB begin | size

  'Start the tv terminal.
  term.start

  'Start the W5100 driver.
  eth.start

  'Display a title string and indicate we are
  'waiting for the ethernet to come up.
  term.out($02)
  term.pstring(string("HydraEtherX TCP Client Test",13,13))
  term.out($01)
  term.pstring(string("Waiting for Eth",13))

  'Wait for awhile for the ethernet to come up.
  waitcnt(cnt + $17d78400)

  'Indicate that we are done waiting.
  term.pstring(string("Done waiting",13,13))

  'Fill out the arrays needed for initialization.

  'Destination IP is IP we are sending to
  dip[0] := 192
  dip[1] := 168
  dip[2] := 1
  dip[3] := 2

  'Fill out the rest of the arrays needed for initialization.
  'W5100 IP address.
  ip[0] := 192
  ip[1] := 168
  ip[2] := 1
  ip[3] := 100

  'Gateway.
  gateway[0] := 192
  gateway[1] := 168
  gateway[2] := 1
  gateway[3] := 1

  'Subnet.
  subnet[0] := 255
  subnet[1] := 255
  subnet[2] := 255
  subnet[3] := 0

  'W5100 MAC address.
  mac[0] := $00
  mac[1] := $70
  mac[2] := $6e
  mac[3] := $69
  mac[4] := $73
  mac[5] := $0a

  'Initialize the W5100.
  eth.init(@gateway, @subnet, @ip, @mac)
```

Notice that the gateway, subnet, ip, and mac arrays are byte arrays and to pass the address of the pointer to the **init** function you use the @ symbol. This example would set your IP to address 192.168.1.100, your gateway to 192.168.1.1, your subnet to 255.255.255.0 and your MAC address would be 00.70.6e.73.0a. Technically you should apply to IEEE to get your own MAC address and if you intend to produce a commercial product based on the W5100 you should do so. But in reality, for educational purposes, all you need to so is be sure that the MAC address isn't the same as any other Ethernet device in your local network.

Again, as you see here you should call the **init** function after the **start** function and before you call the **open** function which we will discuss next.

6.3.4 Open and Close

The next two functions we will discuss are the **open** and **close** functions. Socket programming has been around for a very long time as an attempt to standardize network interfaces. The W5100 loosely models the socket programming scheme. The W5100 supports up to four independent sockets at a time. The EtherX driver only supports one socket at a time but there is nothing stopping you from modifying the driver to support all four. Sockets need to be "opened" in the W5100 and that is why we have the **open** function call. To open a socket on the W5100 we need to tell it whether the socket is a UDP socket or a TCP socket, what the source and destination ports are, and the destination IP address. Actually, if you setup the W5100 to be a TCP server then the destination IP address is not known at the time when you open the socket so that value can be anything as it will be ignored. The **open** function is shown below:

```
PUB open (type, source_port, dest_port, dest_ip) | temp, temp2

  '' This function will open either a TCP or UDP socket based on
  '' the type argument. The source_port and dest_port are the
  '' source and destination port numbers. The last argument
  '' is meant to be a four byte array that stores the destination
  '' IP. We could have just made this a long arg but this
  '' format is more human friendly.

  'Set the socket type so that other function know if we are
  'TCP or UDP
  socket_type := type

  'Configure socket for UDP for TCP (TCP is the default)
  if(socket_type == UDP)
    write($04,$00,2)
  else
    write($04,$00,1)

  'Configure source port
  temp := (source_port >> 8) & $FF
  temp2 := source_port & $FF
  write($04,$04,temp)
  write($04,$05,temp2)

  'Configure dest port
  temp := (dest_port >> 8) & $FF
  temp2 := dest_port & $FF
  write($04,$10,temp)
  write($04,$11,temp2)

  'Dest IP
  write($04,$0c,byte[dest_ip])
  write($04,$0d,byte[dest_ip+1])
  write($04,$0e,byte[dest_ip+2])
  write($04,$0f,byte[dest_ip+3])

  'Open socket
  write($04,$01,1)
```

You can see that the function has four parameters passed into it. The first is the type which is either 0 or 1. A zero means UDP and a one means TCP. The CON sections defines UDP to be 0 and TCP to be 1 for easier readability. The next two arguments are the source and destination ports. The final argument is a pointer to a byte array of the destination IP. This value doesn't make sense if we intend to be a TCP server as we cannot possibly know the IP address of the client that may attempt to connect to us. Thus, if we intend to be a TCP server then this value can be anything as it will be ignored. Notice that the **open** function will set the socket type, source and destination port, and then the destination IP address before actually opening the socket. If you choose to not use the **open** function and instead open a port yourself using the **write** function, be sure not to open the socket until these are set. Do not attempt to change the socket type, ports, or destination IP after the socket has been opened. If you need to change one of these parameters then "close" the port first.

The **close** function will close socket. After the **close** function has been called the socket will no longer send and receive data on that socket. Once closed however, you can change the socket type, source and destination ports, and destination IP address. Then if you wanted to you could re-open the socket and those new parameters will take affect.

6.3.5 Listen, Connect, and Connection Established

If you choose open the socket as a TCP socket then you need to determine if you will be a TCP server or a TCP client. TCP will establish a connection before data is exchanged. There are two functions that will work to establish a connection and which one you use will be based on whether you want to the W5100 to be a server or a client. If you choose to implement your socket as a UDP socket then there is no need to use the **listen**, **connect**, or **con_est** function calls because UDP doesn't establish a connection before sending and/or receiving data.

If the EtherX card will be a TCP server then it will call the **listen** function after it calls the **open** function. This function will tell the W5100 that it will be a server and that it should wait for a client to try and establish a connection with it. After you call the **listen** function you will need to call the **con_est** function which returns true if a connection has been established with a client and false if a connection has not been established. Below is a snippet of code from the SimpleTCPServer code that shows how to use **listen** and **con_est**. It be found in the CodeExamples directory of the CD.

```
  'Initialize the W5100.
  eth.init(@gateway, @subnet, @ip, @mac)

  'Open the socket for TCP communication.
  eth.open(TCP, PORT, PORT, @dip)

  'Listen for a connection.
  eth.listen

  'Display message.
  term.pstring(string("Listening...",13))

  'Wait for connection to be established.
  repeat while eth.con_est == false

  'Display connected message.
  term.pstring(string("Connected to PC",13,13))
```

You can see the order of the functions that needs to be called to setup the W5100 for data exchange as a TCP server in the snippet above. Call the **init**, **open**, and **listen** functions in that order and then call **con_est** as/when needed to determine when and if a connection has been made. If the **con_est** function returns true then a connection has been established with a client and you can then call the **RX** and **TX** functions to exchange data with the client. Do not attempt to receive or transmit data before a connection has been established.

If the EtherX card will be a client then it will call the **connect** function after it called the **open** function. This function will tell the W5100 that it will be a client and that it should attempt to establish a TCP connection with a TCP server location at the IP address specified in the destination IP address field that we specified in the **open** function call. Once the **connect** function has been called you need to call the **con_est** function to determine if a connection has been made. Do not attempt to receive or transmit data before a connection has been established. The **con_est** function will return true when the connection has been established and will return false if a connection has not been established. Below is a snippet of code from the SimpleTCPClient that shows how to use **connect** and **con_est**. It can be found in the CodeExamples directory of the CD.

```
  'Initialize the W5100.
  eth.init(@gateway, @subnet, @ip, @mac)

  'Open the socket for TCP communication.
  eth.open(TCP, PORT, PORT, @dip)

  'Display message.
  term.pstring(string("Connecting...",13))

  'Connect to the PC
  eth.connect

  'Wait for connection to be established.
  repeat while eth.con_est == false

  'Display connected message.
  term.pstring(string("Connected to PC",13,13))
```

You can see that the process for setting up the W5100 as a TCP server and as a TCP client differ by only one line. You use **listen** for a TCP server and **connect** for TCP client. Other than that the process is the same. Whether you choose to implement your socket as a UDP, TCP client, or TCP server the next thing you need to do is to send and receive data. This is accomplished using the **RX** and **TX** functions which we will discuss next.

6.3.6 Transmit

Once the socket has been opened (and a connection established if you are a TCP socket type) then you can send and receive data. The **TX** function is called to transmit data and its code is shown below:

```
PUB TX (dataptr, size) | tptr, offset, startadd, a0, a1, counter, temp

  '' Call this function to send data via the W5100

  'Read the offset so we know what the starting address is
  'of the TX buffer.
  tptr := read($04,$24)         'Read the transmit pointer
  tptr := tptr << 8
  tptr += read($04,$25)
  offset := tptr & $7FF         'Socket 0 has 2K of buffer and that determines the mask here of
$7FF
  startadd := $4000 + offset    'Add the offset to the starting address of socket 0

  'Write the data to the W5100 internal memory buffer
  repeat counter from 0 to size-1
    a1 := startadd & $FF
    a0 := (startadd & $FF00) >> 8
    write(a0, a1, byte[dataptr])
    offset++
    offset := offset & $7FF     'Socket 0 has 2K of buffer and that determines the mask here of
$7FF
    startadd := $4000 + offset  'Add the offset to the starting address of socket 0
    dataptr++

  'Update the offset counter and write it back to the W5100
  tptr += counter
  temp := (tptr & $FF00) >> 8
  write($04,$24,temp)
  temp := tptr & $FF
  write($04,$25,temp)

  'Tell the W5100 to write the data
  write($04,$01,$20)
```

You can see that the function takes two arguments. The first is a pointer to a byte array that contains the data you want to send. The second argument specifies how many bytes you actually want to send. The driver writes the number of bytes you specify in the size argument from the address in main memory starting at the address pointed to by the dataptr argument to the internal buffer of the W5100. Once the bytes have been written to the internal buffer of the W5100 the EtherX driver instructs the W5100 to transmit that data. Keep in mind that the default size for the internal buffer for socket 0 (the socket that the EtherX driver operates on) is 2048 bytes. This means that you can’t write more than 2048 bytes worth of data at one time using the **TX** function.

NOTE	Got a problem being limited to 2048 bytes at a time? No problem. Check out the W5100 data sheet (it’s included on the CD) and read about how to configure the W5100. Specifically, you want to modify the size of the socket buffer. Learn what registers address you need to write to and what data you need to write to get the buffer size you want and write that value to the address you need using the **write** function. That’s what it’s there for!

Another gotcha besides the fact that by default you can’t transmit more than 2048 bytes at a time is that you need to be sure that you don’t specify the size argument to be any larger than your buffer that is pointed to by the dataptr argument. There is no automatic checking for array out of bounds.

The **TX** function abstracts some painful details of how the internal W5100 buffer handles itself. There is an internal counter within the W5100 that determine where it starts sending data from. I encourage you to hurt your head and learn how this process works. It will help you understand things like circular buffers and FIFOs.

6.3.7 Receive

The **TX** function is nice in that it operates the same no matter what your socket type is. Thus, whether you opened a socket as TCP or UDP, the **TX** function doesn’t care. It is used the same way no matter what. However, the **RX** function is called to receive data from the W5100 and it behaves differently depending on you socket type (UDP or TCP). The **RX** function is shown below:

```
PUB RX (dataptr, size, block) : ret_size | offset, startadd, a0, a1, counter, rdptr, temp, tempsize

  '' Call this function to receive data via the W5100.
  '' The function will return the number of bytes read.
  '' Set block == true if you want the function to block waiting for data.
  ''
  '' This function will return the actual number of bytes read.

  'Block waiting for the W5100 to tell us that there is data.
  'Technically we are reading the number of bytes that have
  'been received
  if(block == true)
    repeat
      temp := read($04,$26)
      temp := temp << 8
      temp += read($04,$27)
    while temp == 0             'Wait as long as we have received zero bytes

  'If the user wants a non-block RX call then we will return immediately
  'with a size of zero if there is no data to receive.
  else
    temp := read($04,$26)
    temp := temp << 8
    temp += read($04,$27)
    if(temp == 0)
      return 0

  'Compute the starting address
  rdptr := read($04,$28)        'Read the receive pointer
  rdptr := rdptr << 8
  rdptr += read($04,$29)
  offset := rdptr & $7FF        'Socket 0 has 2K of buffer and that determines mask here of $7FF
  startadd := $6000 + offset    'Add the offset to the starting address of socket 0

  'Determine how many bytes we need to read.
  tempsize := read_rsr
  if(tempsize > size)
    ret_size := size
  else
    ret_size := tempsize

  'Now we read the data from the W5100 and write it to the array
  'pointed to by dataptr.
  repeat counter from 0 to ret_size-1
    a1 := startadd & $FF
    a0 := (startadd & $FF00) >> 8
    byte[dataptr] := read(a0,a1)
    dataptr++
    offset++
    offset := offset & $7FF     'Socket 0 has 2K of buffer and that determines mask here of $7FF
    startadd := $6000 + offset  'Add the offset to the starting address of socket 0

  'Need to increment the rdptr by the number of bytes actually read
  rdptr += ret_size

  'Then write the value of the new pointer back
  temp := (rdptr & $FF00) >> 8
  write($04,$28,temp)
  temp := rdptr & $FF
  write($04,$29,temp)

  'Tell the W5100 that we have read the data
  write($04,$01,$40)

  'Issue the read command to the W5100 which just updates registers.
  'We will block waiting for the W5100 to finish.
  'This takes very little time but we will check it just to be sure.
  repeat
    temp := read($04,$01)
  while temp <> $00
```

As you can see this is the largest and most complex function in the EtherX driver. You can see why we saved it for last. Still, when you boil it down, all it really does is read some data from an internal buffer in the W5100 and write that data to main memory in the Hydra. This function takes in three arguments. The first, dataptr, is the memory address of the byte array that the EtherX driver will write the data that it reads from the W5100 to.

The next argument, size, stipulates the maximum number of bytes that the **RX** function should write to the memory pointed to by dataptr. For example, your receive buffer that you have allocated in the Hydra may only be 32 bytes in size and the W5100 may have received 40 bytes from the internet. In this case you want to be sure the EtherX driver doesn't write 40 bytes worth of data into a buffer that you have only allocated 32 bytes for. The next argument, block, we will skip for a second. The function will return the actual number of bytes written to the buffer pointed to by the dataptr argument. As an example let's again say you had a 32 byte receive buffer setup in the Hydra that was pointed by the dataptr argument and the W5100 had only 16 bytes worth of data that it had received. The EtherX driver will write 16 bytes to the buffer pointed to by the dataptr argument and return a value of 16. In our first example where the W5100 contained 40 bytes but only wrote 32 bytes to the buffer the return value would be 32.

The third argument in the function, block, is a Boolean variable that will instruct the **RX** function what to do if there is currently no data waiting for us in the W5100. If there is no data in the W5100 for us to read we can do one of two things. The first is to wait until there is data available and this is called blocking. The program will "block" which means it sits there and does nothing, waiting for the data to arrive. Set the block argument to true if you want the **RX** function to "block" and wait for data to arrive if there is no data immediately available in the W5100. The second thing you could do if there is no data waiting for us in the W5100 when called is to return immediately and somehow indicate to the application that there was no data available at the time. In our case we would return immediately and set the return value (which remember indicates how many bytes have been read from the W5100 into the Hydra main memory pointed to by the dataptr argument) to 0. To set the **RX** function to "non-blocking" set the block argument to false.

NOTE	Blocking or non-blocking? Which should you use? Remember that a blocking call will wait until there is data available which in theory could be a long time. Simply ask yourself what would happen to my application if it paused and waited around for too long. Would I miss some user input (i.e. Gamepad sample)? Would I be unable to update the video or the music? If a function paused for too long and it would cause a problem in your application then use non-blocking. If you application will never do anything other than look for data from the W5100 until it gets it then better to use blocking. The concepts of blocking and non-blocking are important when writing code for things like device drivers and operating systems.

Sometimes it's helpful to know before we call the **RX** function if there is data available. The EtherX driver has a spin function to read the receive size register called **read_rsr**. If this function returns a non-zero number then there is data in the W5100 for us to read. This **read_rsr** function is also handy in determining if we have read all the data out of the W5100. For example, if we make a call the **RX** function and specify that we should read and store 32 bytes from the W5100 to the main memory of the Hydra pointed to by the dataptr argument and the return value is 32 then we have a problem. We don't know if there are more bytes in the W5100 for us to read. Remember we specify a maximum amount of bytes to be written to the main memory in the Hydra. If we store that maximum amount of data there could still be more in the W5100 waiting for us to read it. But, how could we know? The answer is to call the **read_rsr** function. If after we call **RX** with a size argument of 32 and we get a return value of 32 and we call the **read_rsr** and it returns 0 then we know we got all the data. If it returns a non-zero value then we know there is still data left over in the internal W5100 buffer for us to read out.

The other thing we need to address here is the difference between a call to **RX** when we have a socket type of TCP vs. a socket type of UDP. Assuming there is already data in the W5100 internal buffer waiting for us to read, when our socket type is TCP and we read data from it we will read only the data that was received by the W5100. When we have a socket type of UDP every received packet of data will have an eight byte header appended to it by the W5100. This header consists of the IP address of the received packet (four bytes), the source port (two bytes), and the data size (two bytes). So, if you specified the socket type as UDP and were expecting to your PC to send you 32 bytes you would in fact read 40 bytes from the W5100 when you read the data from its internal buffer.

Just as with the transmit buffer of the W5100, the receive buffer of the W5100 can only store 2048 bytes (default value). Don't let the W5100 get too much data into it before you read or your data will be corrupt. Just as with the transmit buffer you can configure the W5100 to have a larger receive buffer using the **write** function provided by the EtherX driver.

6.3.8 Receive Size Register

The last application function is the **read_rsr** function which will read the receive size register. It will return the number of bytes that the W5100 has stored in its receive buffer for you to read.

If this function returns a non-zero number then there is data in the W5100 for us to read. This **read_rsr** function is also handy in determining if we have read all the data out of the W5100. For example, if we make a call the **RX** function and specify that we should read and store 32 bytes from the W5100 to the main memory of the Hydra pointed to by the dataptr argument and the return value is 32 then we have a problem. We don't know if there are more bytes in the W5100 for us to read. Remember we specify a maximum amount of bytes to be written to the main memory in the Hydra. If we store that maximum amount of data there could still be more in the W5100 waiting for us to read it. But, how could we know? The answer is to call the **read_rsr** function. If after we call **RX** with a size argument of 32 and we get a return value of 32 and we call the **read_rsr** and it returns 0 then we know we got all the data. If it returns a non-zero value then we know there is still data left over in the internal W5100 buffer for us to read out.

6.4 Putting all the Application Functions Together

This section is just meant to give you a skeleton of how and in what order you should use the application functions to use the EtherX card. Actually, the best way to understand how to use the driver is to check out the examples included in the CD in the CodeExamples directory. Section 7 will briefly cover these examples to help you out.

First, determine if you are going to change the default mode. For most applications leaving the default 10Mbps, full duplex mode should be fine. But, should you feel the need to change it, call the **mode** function.

What function you call depends on your socket type. If you are going to use socket type UDP then the order of operations goes like so:

```
objectname.start
objectname.init(@gway, @subnet, @ip, @mac)
objectname.open(0, sport, dport, @dest_ip)

‘Then call RX and TX function to read/write data.
```

If you are going to use socket type TCP and will operate as a TCP server then the order of operations would go like this:

```
object_name.start
object_name.init(@gway, @subnet, @ip, @mac)
object_name.open(1, sport, dport, @dest_ip)
object_name.listen
repeat while object_name.con_est == false

'Then call RX and TX function to read/write data
```

If you are going to use socket type TCP and will operate as a TCP client then the order of operations would go like this:

```
object_name.start
object_name.init(@gway, @subnet, @ip, @mac)
object_name.open(1, sport, dport, @dest_ip)
object_name.connect
repeat while object_name.con_est == false

'Then call RX and TX function to read/write data
```

Check out Section 7 and the code examples to see how this skeleton format fits into real applications.

7.0 The HYDRA EtherX Demos

All the demos discussed here are located on the CD under the "CodeExamples" directory. We'll discuss each of them here and what they are meant to demonstrate.

7.1 Ping Example

The Ping example is located on the CD_ROM in the following directory:

CD_ROM:\CodeExamples\Ping

Simply put, it allows the user to set the IP address of the W5100 using the gamepad so that you can attempt to ping the device. The purpose of this is to test that the EtherX card is working and that your PC can communicate with it. This is the program that you are instructed to load in the Quick Start portion of this manual.

The main top level file is called **Ping.spin** and pulls in a number of sub-objects directly or indirectly, they are:

W5100_drv_011.spin	- W5100 Driver
tv_terminal_010_finish.spin	- TV terminal from Parallax. Modified to include the finish function.
tv_drv_010.spin	- TV Driver from Parallax.
graphics_drv_010.spin	- Graphics Driver from Parallax.
gamepad_drv_001.spin	- Gamepad driver from Nurve.

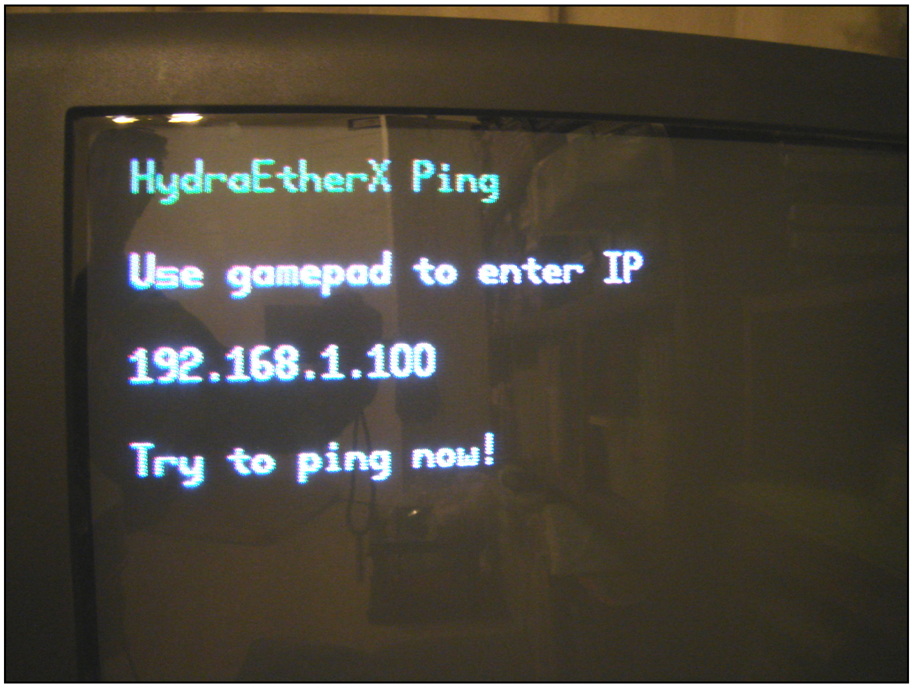
The program begins with a splash screen saying that it is waiting for the Ethernet to come up. It takes a couple of seconds for the PLLs in the W5100 to lock onto the Ethernet signal. After starting the driver the in the Hydra the Ping program will wait for 5 seconds for the Ethernet to lock. After that you will be given a screen to set the IP address of the W5100.

The field you are currently modifying is highlighted in red. Pressing up will increment the address of that field. Pressing down will decrement the address of that field. Use left and right to move to the next field that can be modified. When all the fields are have been modified to your pleasing then push any button.

At that point the Hydra will set the W5100 IP address to the value you specified with the gamepad. The Hydra will then display a message for you to now attempt to ping the EtherX card with your PC.

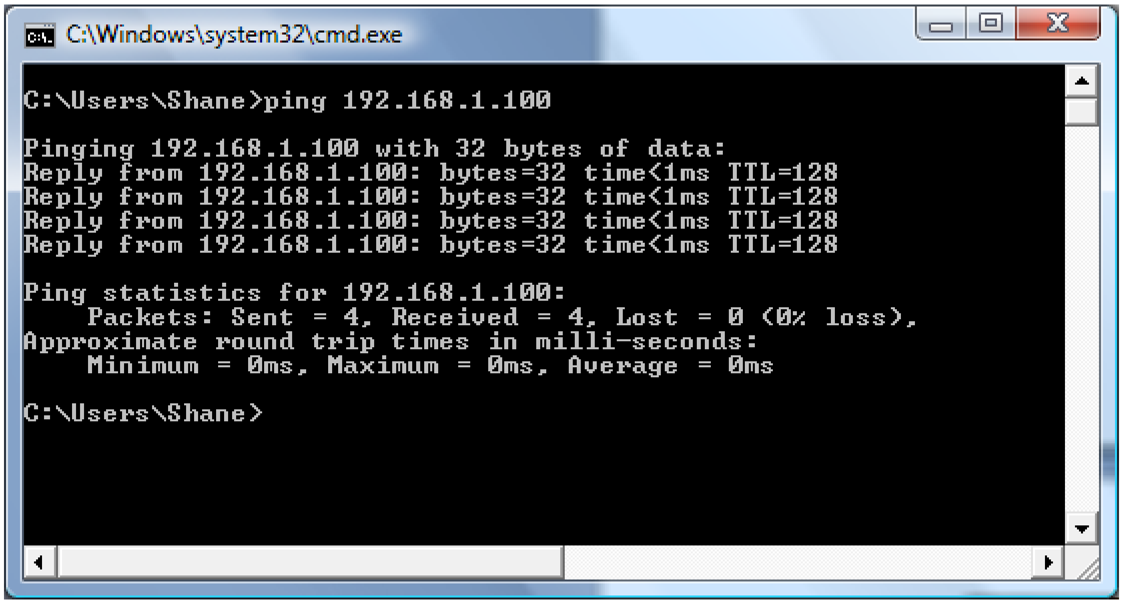
Figure 25.0 shows the TV terminal output of the Hydra after the Ping program has been run. Figure 26.0 shows the command prompt on the PC after pinging the Hydra.

Figure 25.0 – TV terminal output from the Hydra.



```
HydraEtherX Ping

Use gamepad to enter IP

192.168.1.100

Try to ping now!
```

Figure 26.0 – Command prompt after pinging the Hydra.



```
C:\Users\Shane>ping 192.168.1.100

Pinging 192.168.1.100 with 32 bytes of data:
Reply from 192.168.1.100: bytes=32 time<1ms TTL=128
Reply from 192.168.1.100: bytes=32 time<1ms TTL=128
Reply from 192.168.1.100: bytes=32 time<1ms TTL=128
Reply from 192.168.1.100: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.1.100:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\Users\Shane>
```

You see a demonstration of the Ping program running on YouTube. Check it out:

http://www.youtube.com/v/7SZKPr7Lbww

7.2 TCP Server Example

The TCP server example is actually two examples in one. It shows how to make the EtherX card a TCP server and how to write TCP client code for the PC in C. To run the example simply load in the TCP server example into the Hydra. The EtherX card will then be configured as a TCP server will listen for a connection from the PC. Next, you need to execute the C code which will establish a TCP connection with the EtherX card and send it text string. The Hydra will receive this string and display it on the TV. Once it receives the text string and displays it, the Hydra will send a text string to the PC. The C program will then receive this string and display it on the console.

The main directory for the TCP Server Example is located on the CD_ROM in the following directory:

CD_ROM:\CodeExamples\SimpleTCPServer\

Within that directory are two more directories "**C**" and "**Spin**". The "**C**" directory contains the C code for a simple TCP client. The "**Spin**" directory contains the Hydra spin code for a simple TCP server.

The main top level spin file in the "**Spin**" directory is **SimpleTCPServer.spin** pulls in a number of sub-objects directly or indirectly, they are:

W5100_drv_011.spin	- W5100 Driver
tv_terminal_010_.spin	- TV terminal from Parallax.
tv_drv_010.spin	- TV Driver from Parallax.
graphics_drv_010.spin	- Graphics Driver from Parallax.

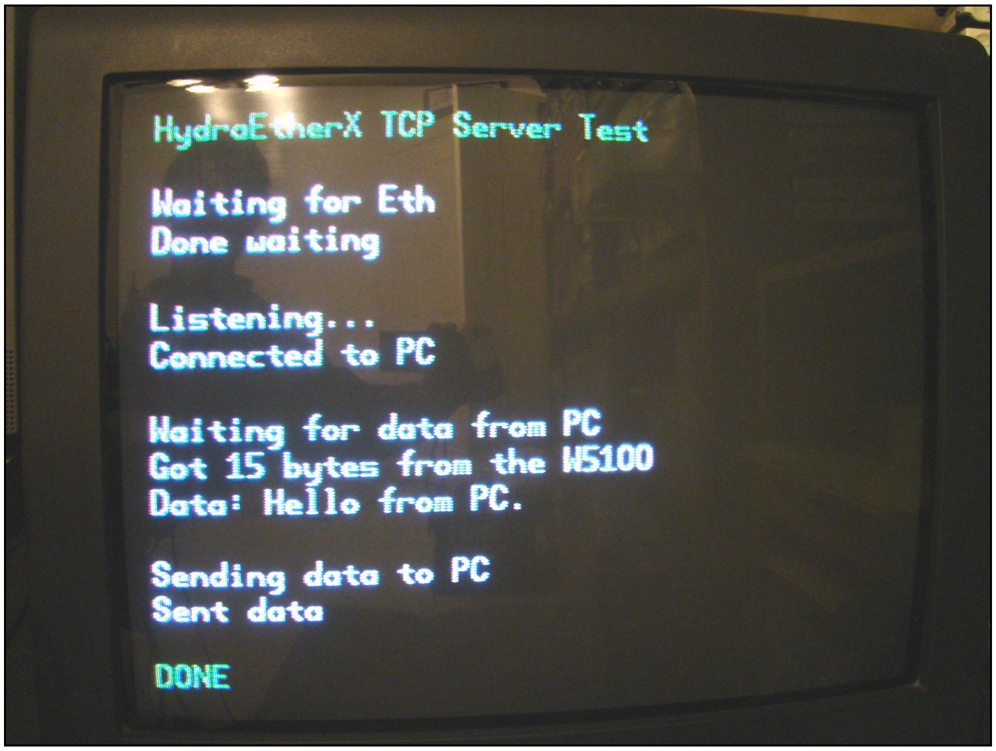
Load the program into the Hydra and you should see a message that is it listening for a connection. You then need to open a command prompt and execute the C code which is located at:

CD_ROM:\CodeExamples\SimpleTCPServer\C\lcc\tcpclient.exe

The program will immediately attempt to establish a connection with the EtherX card and attempt to send and receive data. The data that is received by the C program will be displayed on the command prompt. Similarly, the data that is received by the EtherX card will be displayed by the Hydra on the TV terminal. The data sent by both the C program and the Hydra are simple text strings that say "Hello."

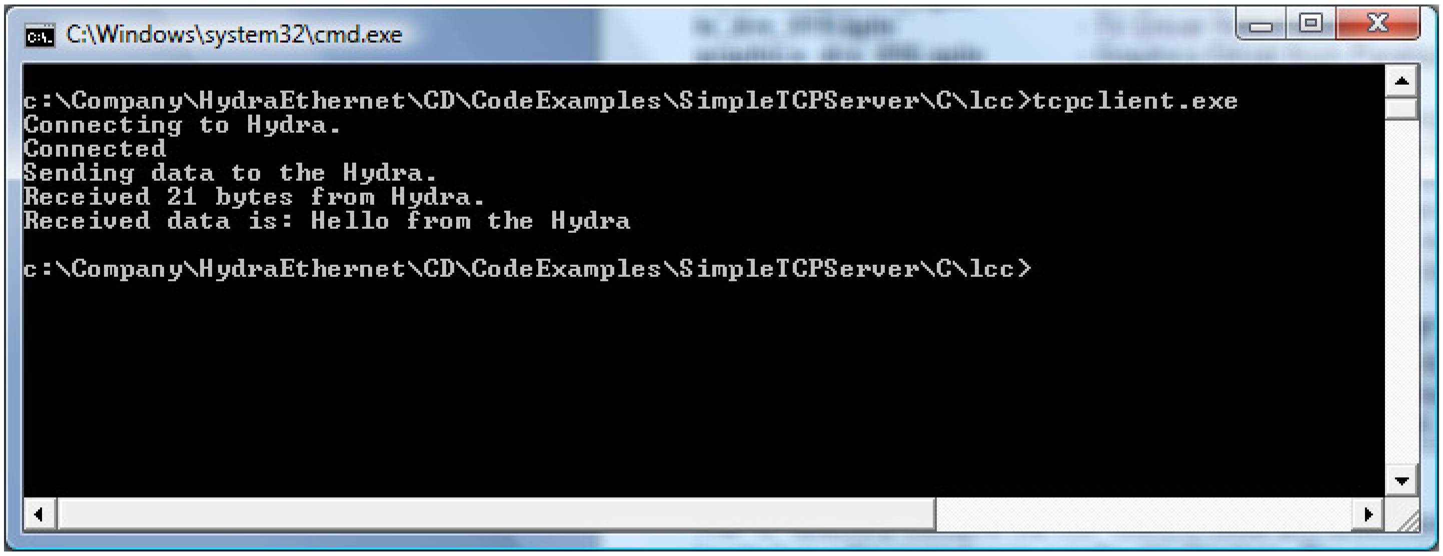
Figure 27.0 shows the TV terminal output of the Hydra after the simple TCP server program has been run. Figure 28.0 shows the command prompt on the PC after the TCP client program has been run.

Figure 27.0 – TV terminal output from the Hydra.



```
HydraEtherX TCP Server Test

Waiting for Eth
Done waiting

Listening...
Connected to PC

Waiting for data from PC
Got 15 bytes from the W5100
Data: Hello from PC.

Sending data to PC
Sent data

DONE
```

Figure 28.0 – Command prompt output.



```
c:\Company\HydraEthernet\CD\CodeExamples\SimpleTCPServer\C\lcc>tcpclient.exe
Connecting to Hydra.
Connected
Sending data to the Hydra.
Received 21 bytes from Hydra.
Received data is: Hello from the Hydra

c:\Company\HydraEthernet\CD\CodeExamples\SimpleTCPServer\C\lcc>
```

NOTE	The SimpleTCPServer program is initialized with IP address 192.168.1.100. If your sub-domain is different or if this IP address is already used on your network then you need to change this IP. Also, the HydraEtherX listens on port 120. If you have problems getting the PC and the EtherX card to communicate make sure that your router and firewall will allow traffic on this port.

The “**C**” directory contains the TCP client C code and executable that is meant to run on your PC. The C code itself is located on a file named **main.c** in the “**C**” directory. The executable is located in a subdirectory called “**lcc**”. You will notice additional files in these directories and they are generated by the compiler. I used the LCC compiler to the code. You can download this compiler at:

http://www.cs.virginia.edu/~lcc-win32/

It's a free C compiler that anyone can use for free. Try it out!

7.3 TCP Client Example

The TCP client example is actually two examples in one. It shows how to make the EtherX card a TCP client and how to write TCP server code for the PC in C. To run the example execute the C code which will begin by listening for a TCP client to attempt a connection. Then load in the TCP client code into the Hydra. The EtherX card will be configured as a TCP client and will try to connect to the TCP server on the PC. Once a connection is established the PC will send a text string to the EtherX card. The Hydra will display this string to the TV terminal and send a text string back to the PC server which will be waiting to receive data. When the PC receives this text string from the EtherX card it will display the message to the command prompt.

Start by executing the TCP server in a command prompt on your PC. The program is located at:

CD_ROM:\CodeExamples\SimpleTCPClient\C\lcc\tcpserver.exe

This program will then listen for a connection from the EtherX card.

Next load the TPC client code into the Hydra. The code is located on the CD_ROM at:

CD_ROM:\CodeExamples\SimpleTCPClient\Spin

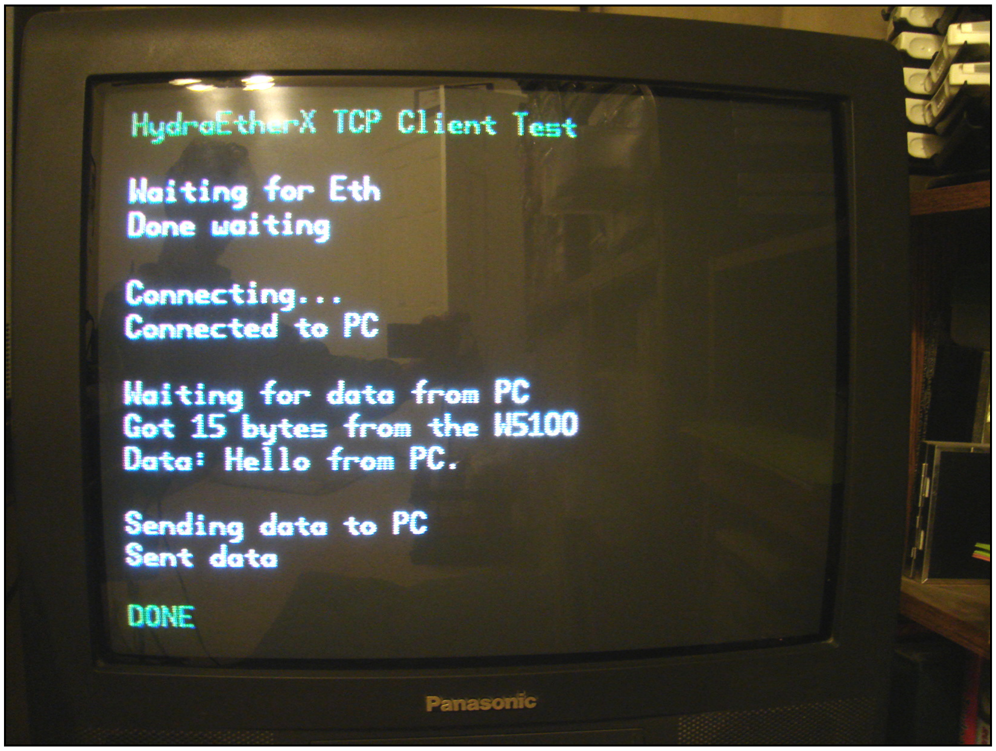
The main top level spin file in the “Spin” directory is **SimpleTCPClient.spin** pulls in a number of sub-objects directly or indirectly, they are:

W5100_drv_011.spin	- W5100 Driver
tv_terminal_010_.spin	- TV terminal from Parallax.
tv_drv_010.spin	- TV Driver from Parallax.
graphics_drv_010.spin	- Graphics Driver from Parallax.

The Hydra will then configure the W5100 as a TCP client and attempt a TCP connection with the PC. The data that is received by the C program will be displayed on the command prompt. Similarly, the data that is received by the EtherX card will be displayed by the Hydra on the TV terminal. The data sent by both the C program and the Hydra are simple text strings that say “Hello.”

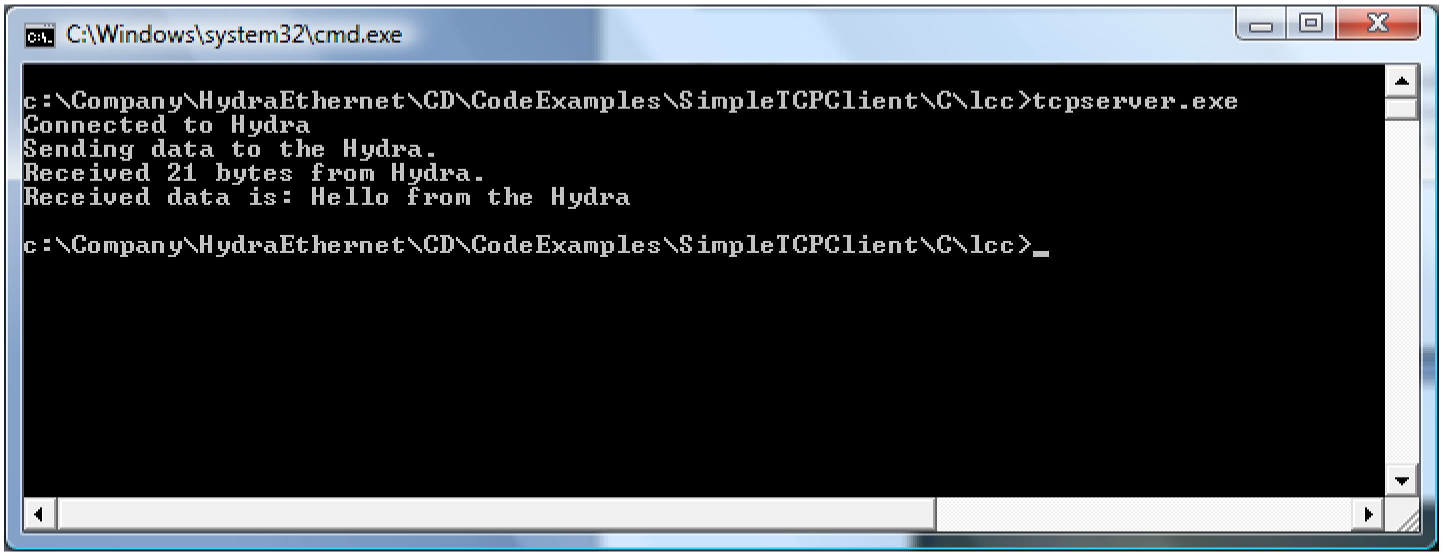
Figure 29.0 shows the TV terminal output of the Hydra after the simple TCP client program has been run. Figure 30.0 shows the command prompt on the PC after the TCP server program has been run.

Figure 29.0 - TV terminal output from the Hydra.



```
HydraEtherX TCP Client Test

Waiting for Eth
Done waiting

Connecting...
Connected to PC

Waiting for data from PC
Got 15 bytes from the W5100
Data: Hello from PC.

Sending data to PC
Sent data

DONE
```

Figure 30.0 – Command prompt output.



```
c:\Company\HydraEthernet\CD\CodeExamples\SimpleTCPClient\C\lcc>tcpserver.exe
Connected to Hydra
Sending data to the Hydra.
Received 21 bytes from Hydra.
Received data is: Hello from the Hydra

c:\Company\HydraEthernet\CD\CodeExamples\SimpleTCPClient\C\lcc>_
```

NOTE	The SimpleTCPClient program is initialized with IP address 192.168.1.100. If your sub-domain is different or if this IP address is already used on your network then you need to change this IP. The SimpleTCPClient assumes that the PC lives at 192.168.1.2. If your PC has a different IP address then this needs to be changed in the Spin code. Also, the HydraEtherX listens on port 120. If you have problems getting the PC and the EtherX card to communicate make sure that your router and firewall will allow traffic on this port.

The "**C**" directory contains the TCP server C code and executable that is meant to run on your PC. The C code itself is located on a file named **main.c** in the "**C**" directory. The executable is located in a subdirectory called "**lcc**". You will notice additional files in these directories and they are generated by the compiler. I used the LCC compiler to the code. You can download this compiler at:

http://www.cs.virginia.edu/~lcc-win32/

It's a free C compiler that anyone can use for free. Try it out!

7.4 UDP Example

Just as the TCP client and server were two examples in one so is the UDP example. It shows how to set up the EtherX card to use the UDP protocol as well as how to write UDP code for the PC in C. Remember that UDP is connectionless so there is no listen or connect call needed for either the PC or the EtherX card.

The PC will then listen for data from the EtherX card. Load the UDP program into the Hydra and the Hydra will configure the EtherX card to use the UDP protocol. The Hydra will then send a text string to the PC which the PC will display on the command prompt. When the PC gets the string and displays it, it will send a text string the EtherX card which the Hydra will display on the TV terminal.

Start by executing the UDP program in a command prompt on your PC. The program is located on the CD_ROM at:

CD_ROM:\CodeExamples\SimpleUDP\C\lcc\udp.exe

This program will then wait for data to arrive from the EtherX card.

Next load the UDP code into the Hydra. The code is located on the CD_ROM at:

CD_ROM:\CodeExamples\SimpleUDP\Spin

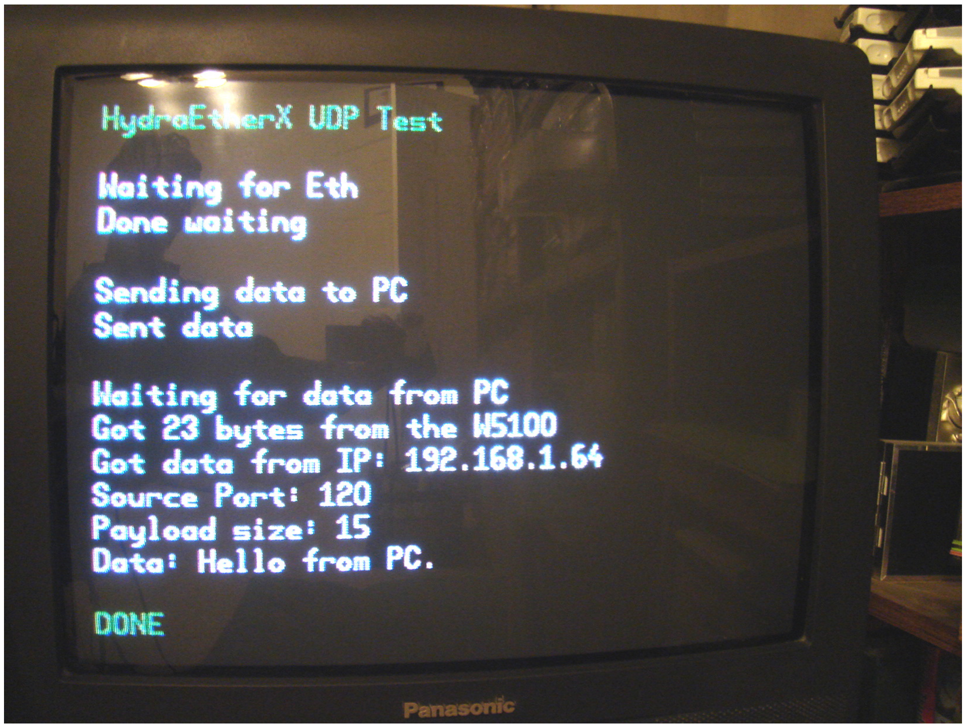
The main top level spin file in the "Spin" directory is **SimpleUDP.spin** pulls in a number of sub-objects directly or indirectly, they are:

W5100_drv_011.spin	- W5100 Driver
tv_terminal_010_.spin	- TV terminal from Parallax.
tv_drv_010.spin	- TV Driver from Parallax.
graphics_drv_010.spin	- Graphics Driver from Parallax.

The Hydra will then configure the W5100 to use the UDP protocol and will send a text string to the PC. The data that is received by the C program will be displayed on the command prompt. Similarly, the data that is received by the EtherX card will be displayed by the Hydra on the TV terminal. The data sent by both the C program and the Hydra are simple text strings that say "Hello."

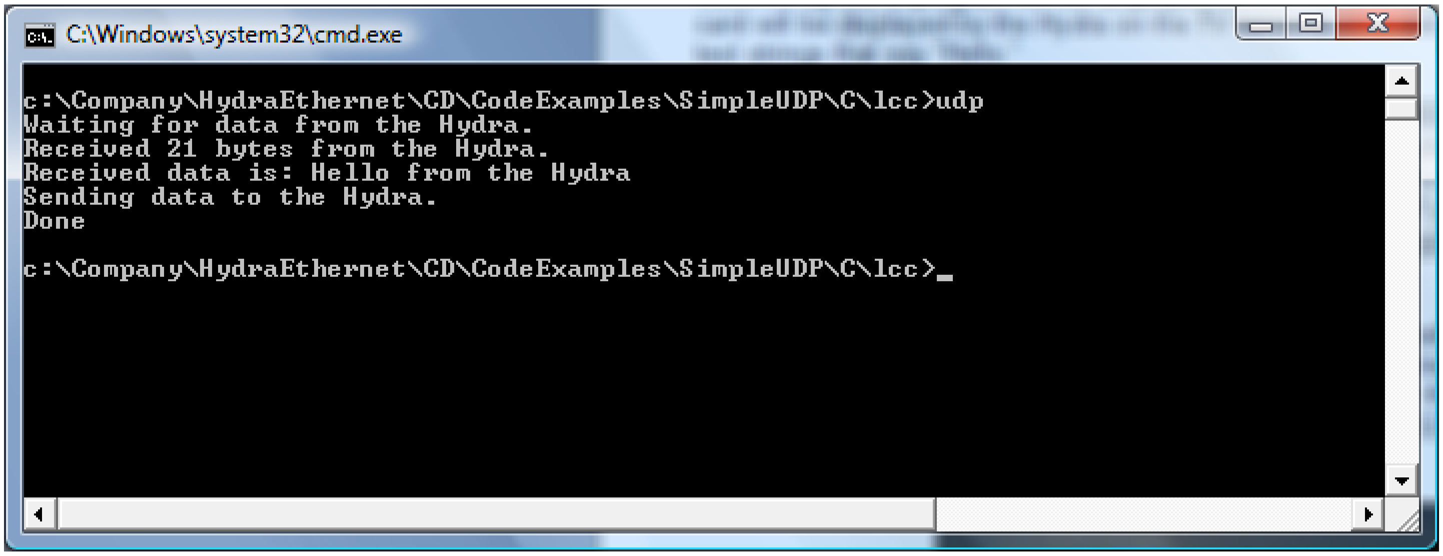
Figure 31.0 shows the TV terminal output of the Hydra after the simple UDP program has been run. Figure 32.0 shows the command prompt on the PC after the UDP program has been run.

Figure 31.0 - TV terminal output from the Hydra.



```
HydraEtherX UDP Test

Waiting for Eth
Done waiting

Sending data to PC
Sent data

Waiting for data from PC
Got 23 bytes from the W5100
Got data from IP: 192.168.1.64
Source Port: 120
Payload size: 15
Data: Hello from PC.

DONE
```

Figure 32.0 – Command prompt output.



```
c:\Company\HydraEthernet\CD\CodeExamples\SimpleUDP\C\lcc>udp
Waiting for data from the Hydra.
Received 21 bytes from the Hydra.
Received data is: Hello from the Hydra
Sending data to the Hydra.
Done

c:\Company\HydraEthernet\CD\CodeExamples\SimpleUDP\C\lcc>_
```

NOTE	The SimpleUDP program is initialized with IP address 192.168.1.100. If your sub-domain is different or if this IP address is already used on your network then you need to change this IP. The SimpleUDP assumes that the PC lives at 192.168.1.2. If your PC has a different IP address then this needs to be changed in the Spin code. Also, the HydraEtherX listens on port 120. If you have problems getting the PC and the EtherX card to communicate make sure that your router and firewall will allow traffic on this port.

The "**C**" directory contains the TCP server C code and executable that is meant to run on your PC. The C code itself is located on a file named main.c in the "**C**" directory. The executable is located in a subdirectory called "**lcc**". You will notice additional files in these directories and they are generated by the compiler. I used the LCC compiler to the code. You can download this compiler at:

http://www.cs.virginia.edu/~lcc-win32/

It's a free C compiler that anyone can use for free. Try it out!

7.5 Tic Tac Toe

The next example is an example of a TCP based game. For this example you will need two Hydras connected via the EtherX cards. You can see the Tic Tac Toe game running on a local network on two Hydras in this YouTube clip:

http://www.youtube.com/watch?v=EqzG31wzOIE

The game is currently setup such that the server and the client are determined at startup by the players. The game will then set the MAC and IP addresses based on whether that particular Hydra is a client or server. This is a handy way to write one piece of code that will setup the EtherX card based on whether the EtherX card will be a client or server.

It is possible to run the game on two Hydras that are not on the same network. Simply configure your router to forward all data for port 200 (port used by the Tic Tac Toe game) to the IP address of the EtherX card on the Hydra. This would allow any two players on the internet to play the game.

The point of this game is to show when it is appropriate to use TCP for games. Turn based games (not real time) are ideal for using TCP. Time is not critical in turn based games but the data is. Remember that TCP guarantees the delivery of data. For a game like Tic Tac Toe we don't care if our opponent's new move takes an extra second to reach our Hydra but we **must** receive the data. Data is only sent when the player makes a move. Thus, if we loose that data then we will never know that it is our turn.

7.6 Squish Kevin

Squish Kevin is an example of a real time UDP based game. For this example you will need two Hydras connected via the EtherX cards. The code is written so that the EtherX card is configured to a set MAC and IP address at start up. If you happen to have two Hydras be sure that the MAC and IP addresses are different for each one. To play this game over the internet configure your router to forward all data for port 200 (port used by the Squish Kevin game) to the IP address of the EtherX card which is specified in the spin code.

The rules of the game are very simple. You are a blue circle on the screen and your opponent is a red circle on the screen. The point is to move your circle over your opponent and hit any button in an attempt to "squish" him. If you succeed you get a point. First player to five points win. You can see Squish Kevin in action on two Hydras on a local network in this YouTube link:

http://www.youtube.com/watch?v=DNHr7VTmL8Q

The point of this game is to show when it is appropriate to use UDP for games. For Squish Kevin we don't care as much if we happen to loose a packet. There will be another packet along in another 1/16th of a second. Most likely the player will never notice. Unlike Tic Tac Toe which only sent data when the player moved, Squish Kevin constantly sends data about the players' position and status. The real important thing is not the reliability of the data arrival but the speed at which the data arrives. We want the data to reach the players as quick as possible. As soon as a player moves we want his opponent on other Hydra to instantaneously see it. UDP will appear faster in real time games because the internet devices won't halt all transmissions after packet losses. Remember that UDP has no way of knowing whether transmitted data has reached its destination properly. It just keeps sending hoping for the best.

7.7 Hydra Web Server

The point of the Hydra Web Server is simply so show off some of the cool things you can do with the Hydra and its EtherX card besides play games. This web server will send over some html to the web client and redirect it to show a picture of the EtherX card from the Avery Digital web site.

So, to set up a web server that is accessible by other computers on the internet, configure your router to forward all port 80 data to the IP address of the EtherX card. The IP address is specified in the spin code for the server. Be careful though. Some ISPs will not allow traffic on port 80. There is a simple fix for this however. Just have the Hydra Web Server listen on a different port and instruct the web browser to fetch the web page from that port. By default all web browsers attempt to fetch the web page from port 80 and nearly all allow you to specify a different port if you so wish. As an example, if you type www.somesite.com:8080 into Internet Explorer it would try to fetch a web page from somesite.com on port 8080.

See the Hydra Web Server in action by checking out this YouTube link:

http://www.youtube.com/watch?v=QupQJw0WIEE

8.0 Applications

So now that you have the EtherX card and are a little familiar with how to use it, what can you do with it. In actuality the list is endless but here are a few suggestions:

1. Create a streaming MP3 player. You PC can stream the MP3 file to the Hydra which will connect to a TV and not only play the MP3 on the TV but display the song title and artist as well.
2. Create a multi-level single player game in which each level is actually stored on the PC and when the player reaches the new level the Hydra pulls the data for the new level from the PC.
3. Create a very simple text based web browser.
4. Cool Games!!!

8.1 EtherX card without Hydra

I wanted to put a small amount of attention to the fact that it is possible to use the EtherX card without the Hydra. In actuality nothing about this card is exclusively Hydra specific. All one would need to do is get connector that interfaces the fingers on the end of the board to whatever you want. In fact, all the EtherX requires to work are 3.3V, ground, reset, and the SPI signals. The OPMODE pins are optional. They can be left floating as they are pulled to the default 10Mbps value by default.

In truth not even a connector is really needed. If you have some solder skills all you would need to do is solder wires to the fingers of the EtherX card that you need and then wire the EtherX card to breadboard is you so wish. Many microcontrollers have built in SPI hardware. With this in mind your favorite microcontroller can be on the internet in no time.

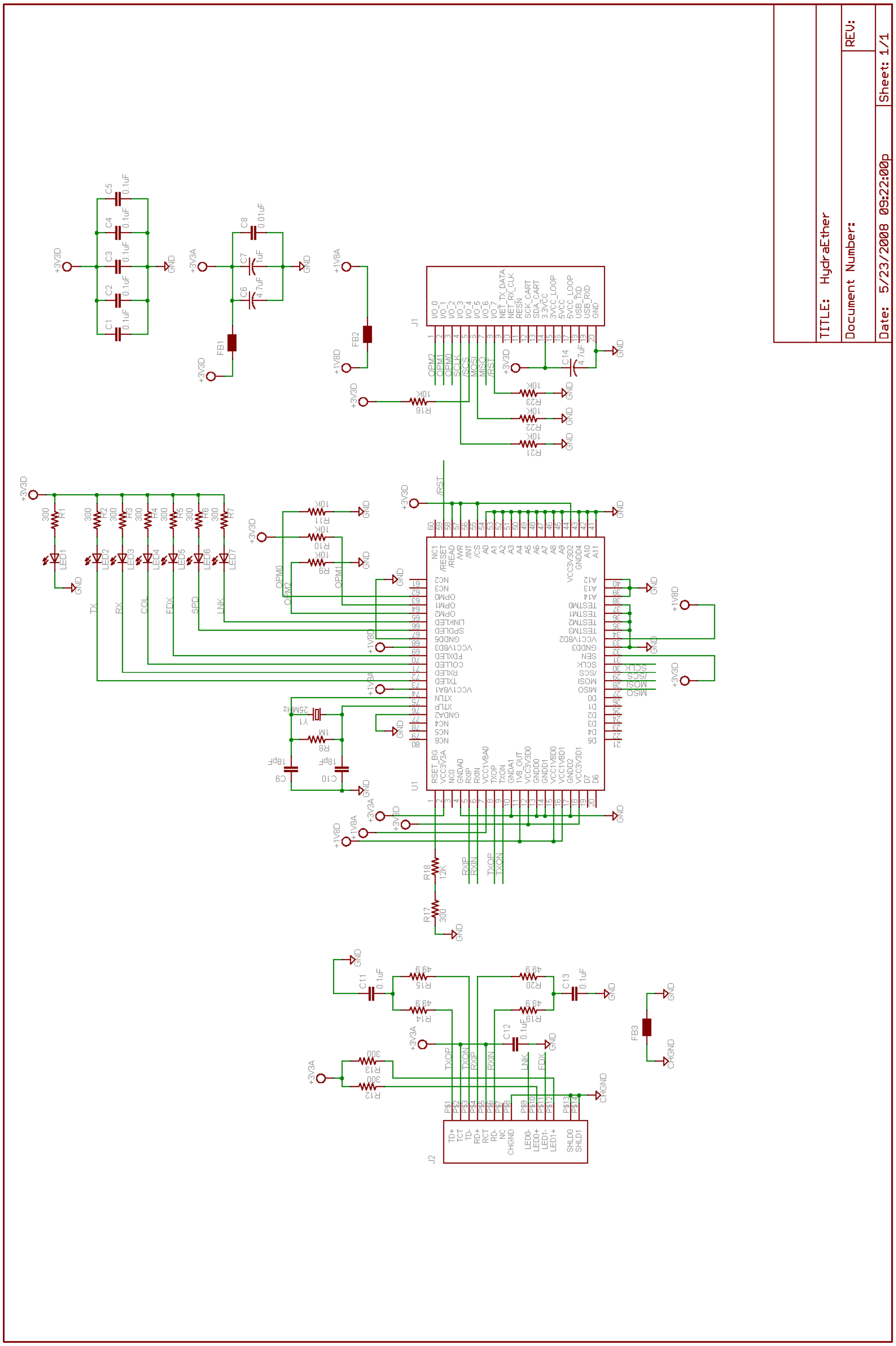
Appendices

The following are a short list of appendices that cover the HYDRA EtherX schematics, PCB, and driver source code listing.

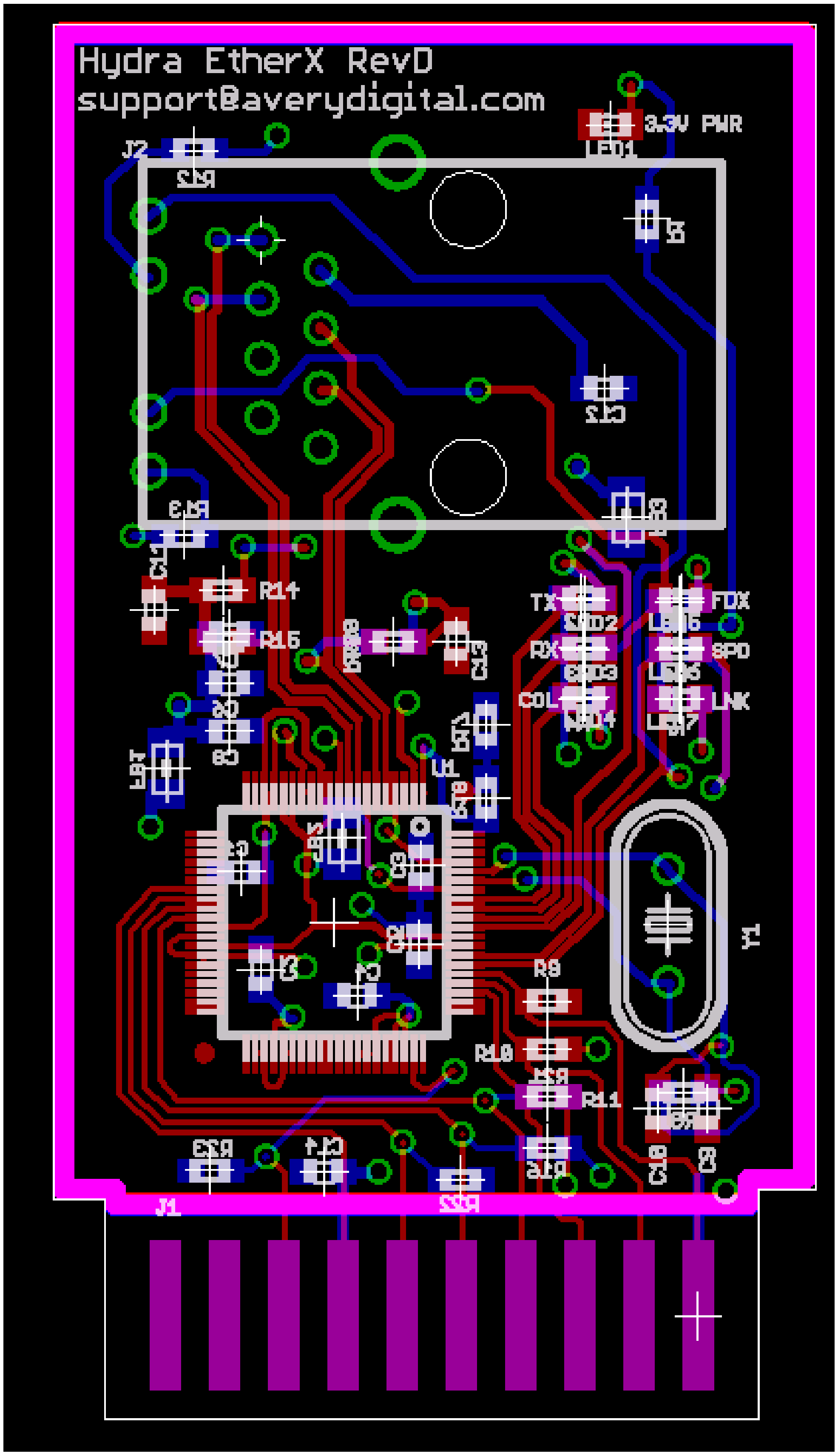
- Appendix A - HYDRA EtherX Schematic
- Appendix B - PCB Reference Layout
- Appendix C - API Driver Source Code Listing

A. HYDRA ETHERX Schematic

Revision D. (Next Page)


TITLE: HydraEther
Document Number:
REV:
Date: 5/23/2008 09:22:00p
Sheet: 1/1

B. PCB Reference Layout



Revision D. (Not actual size)

C. API Driver Source Code Listing

The following is a listing of the entire EtherX driver. Normally, we would never put a code dump like this in a printed book since it's a huge waste of paper, but since this is an eBook, it's nice to have the listing in the same document for reference.

```
{{///////////////////////////////////////////////////////////////////////

W5100 Ethernet Driver
Author: Shane Avery
LAST MODIFIED: 8.18.08
VERSION 1.1

Spin Functions:
        start : okay
                Starts the ASM in a new COG.
                returns: okay - determines if the cog started ok or not

        stop
                Stops driver - frees a cog
                args:    none
                returns: none

        init (gway, subnet, ip, mac)
                Inits some registers.
                args:    gway - 4 byte array which is the gateway IP
                         subnet - 4 byte array which is the subnet
                         ip - 4 byte array which is this device's IP
                         mac - 6 byte array that is the MAC address
                returns: none


        open (type, source_port, dest_port, dest_ip)
                Opens Socket0.
                args:    type - 0 for UDP or 1 for TCP
                         source_port - source port
                         dest_port - desination port
                         dest_ip - destination ip (4 byte array)
                returns: none

        close
                Closes the socket.
                args:    none
                returns: none

        listen
                When TCP server use listen to establish connection
                args:    none
                returns: none

        connect
                When TCP client use connect to establish connection
                args:    none
                returns: none

        TX (dataptr, size)
                Transmits data
                args:    dataptr - byte array to be transmitted
                         size - size of the array
                returns: none

        RX (dataptr, size, block) : ret_size
                Receives data
                args:    dataptr - byte array received data is written to
                         size - the max size of the byte array
                         block - set to true to block (wait forever) for data
                returns: ret_size - the actual size of data received

        read_rsr : ret_val
                Read the receive size register. A non-zero indicates RX data.
                args:    none
                returns: ret_val - the value of the receive size register
        con_est : ret_val
                Returns true if a connection has been established
                args:    none
                returns: ret_val - true if connection has been established

        read(a0, a1) : ret_val
                Read the byte at address a0, a1
                args:    a0 - First byte of the address
                         a1 - Second byte of the address
                returns: ret_val - the byte from address a0,a1

        write(a0, a1, dout)
                Write a byte to address a0,a1
                args:    a0 - First byte of the address
                         a1 - Second byte of the address
                         dout - Data byte to write
                returns: none

        mode(opmode)
                Set the opmode of the W5100
                args:    opmode - New value of the opmode (valid from 0-7)
                returns: none

API Instructions:
        TCP Server:
                object_name.start
                object_name.init(@gway, @subnet, @ip, @mac)
                object_name.open(1, sport, dport, @dest_ip)
                object_name.listen
                repeat while object_name.con_est == false

                'Then call RX and TX function to read/write data

        TCP Client:
                object_name.start
                object_name.init(@gway, @subnet, @ip, @mac)
                object_name.open(1, sport, dport, @dest_ip)
                object_name.connect
                repeat while object_name.con_est == false

                'Then call RX and TX function to read/write data

         UDP:
                object_name.start
                object_name.init(@gway, @subnet, @ip, @mac)
                object_name.open(0, sport, dport, @dest_ip)

                'Then call RX and TX function to read/write data

                '*******************************************************
                'NOTE: UDP packet reads include the UDP header
                '*******************************************************

Performance measurements:
Size:
        Default of 2048 byte buffer for the socket. Thus, never TX more than 2K bytes.
        Never let the RX size be more than 2K bytes.

Speed performance (TCP w/ 256 byte payload):
        Write speed 7.5KB/sec (payload data)
        Read speed 7.8KB/sec (payload data)
        SPI write assembly takes 13.6us to write one byte
        SPI write assembly + spin functions take 58us (ideal)
        SPI read assembly take 14.1us to read one byte
        SPI read assembly + spin functions take 56us (ideal)

Storage:
        ASM portion is 396 bytes
        Entire driver is 326 longs with 8 longs variable

Detailed Change Log
--------------------
v1.1 (8.18.08)
- Removed read_sr function (only currently used to determine connection status)
- Added the conn_est function

v1.0 (5.14.08)
- Changed the pins to reflect the new PCB design.
- Driver can now select the W5100 mode.

v0.9 (4.22.08)
- Start pub function now has no args and takes the device out of
  reset.
- Added a new pub function called init which does include the
  ip, mac, gway, and subnet args.

v0.8 (4.1.08)
- Changed the start pub function to include the ip, mac, gway, and
  subnet args.

v0.7 (3.18.08)
- Improved the RX function
- More detail in the comment header

v0.6 (3.16.08)
- Beta release

//////////////////////////////////////////////////////////////////////}}

'/////////////////////////////////////////////////////////////////////
' CONSTANTS SECTION ///////////////////////////////////////////////////
'/////////////////////////////////////////////////////////////////////
CON

  _clkmode = xtal1 + pll8x
  _xinfreq = 10_000_000 + 0000

  'Constants for UDP for TCP port
  UDP = 0
  TCP = 1

  'Constants used to make the code more readable
  SPI_RD = 1
  SPI_WR = 2
  SPI_DONE = 0

VAR

  long  SPIRW, add0, add1, dataout, datain
  long  socket_type
  long  cogon, cog

OBJ


PUB start : okay

  '' This is the public start function. It starts
  '' a new cog at the assembly entry point

  'Start the SPI code in a new COG
  stop
  okay := cogon := (cog := cognew(@entry, @SPIRW)) > 0

PUB stop

'' Stops driver - frees a cog

  if cogon~
    cogstop(cog)

PUB init (gway, subnet, ip, mac)

  'Init the registers
  'Gateway
  write($00,$01,byte[gway])
  write($00,$02,byte[gway+1])
  write($00,$03,byte[gway+2])
  write($00,$04,byte[gway+3])

  'Subnet
  write($00,$05,byte[subnet])
  write($00,$06,byte[subnet+1])
  write($00,$07,byte[subnet+2])
  write($00,$08,byte[subnet+3])

  'MAC
  write($00,$09,byte[mac])
  write($00,$0a,byte[mac+1])
  write($00,$0b,byte[mac+2])
  write($00,$0c,byte[mac+3])
  write($00,$0d,byte[mac+4])
  write($00,$0e,byte[mac+5])

  'IP
  write($00,$0f,byte[ip])
  write($00,$10,byte[ip+1])
  write($00,$11,byte[ip+2])
  write($00,$12,byte[ip+3])

PUB open (type, source_port, dest_port, dest_ip) | temp, temp2

  '' This function will open either a TCP or UDP socket based on
  '' the type argument. The source_port and dest_port are the
  '' source and destination port numbers. The last argument
  '' is meant to be a four byte array that stores the destination
  '' IP. We could have just made this a long arg but this
  '' format is more human friendly.

  'Set the socket type so that other function know if we are
  'TCP or UDP
  socket_type := type

  'Configure socket for UDP for TCP (TCP is the default)
  if(socket_type == UDP)
    write($04,$00,2)
  else
    write($04,$00,1)

  'Configure source port
  temp := (source_port >> 8) & $FF
  temp2 := source_port & $FF
  write($04,$04,temp)
  write($04,$05,temp2)

  'Configure dest port
  temp := (dest_port >> 8) & $FF
  temp2 := dest_port & $FF
  write($04,$10,temp)
  write($04,$11,temp2)

  'Dest IP
  write($04,$0c,byte[dest_ip])
  write($04,$0d,byte[dest_ip+1])
  write($04,$0e,byte[dest_ip+2])
  write($04,$0f,byte[dest_ip+3])

  'Open socket
  write($04,$01,1)

PUB close

  '' Close the socket.

  write($04,$01,$10)

PUB listen

  '' This function will initiate a TCP listen. Call this function
  '' for a TCP server to listen for a connection

  if(socket_type == TCP)
    write($04,$01,2)

PUB connect

  '' When TCP this function will establish a connection with a server.

  if(socket_type == TCP)
    write($04,$01,4)

PUB TX (dataptr, size) | tptr, offset, startadd, a0, a1, counter, temp

  '' Call this function to send data via the W5100

  'Read the offset so we know what the starting address is
  'of the TX buffer.
  tptr := read($04,$24)         'Read the transmit pointer
  tptr := tptr << 8
  tptr += read($04,$25)
  offset := tptr & $7FF         'Socket 0 has 2K of buffer and that determines mask here of $7FF
  startadd := $4000 + offset    'Add the offset to the starting address of socket 0

  'Write the data to the W5100 internal memory buffer
  repeat counter from 0 to size-1
    a1 := startadd & $FF
    a0 := (startadd & $FF00) >> 8
    write(a0, a1, byte[dataptr])
    offset++
    offset := offset & $7FF     'Socket 0 has 2K of buffer and that determines mask here of $7FF
    startadd := $4000 + offset  'Add the offset to the starting address of socket 0
    dataptr++

  'Update the offset counter and write it back to the W5100
  tptr += counter
  temp := (tptr & $FF00) >> 8
  write($04,$24,temp)
  temp := tptr & $FF
  write($04,$25,temp)

  'Tell the W5100 to write the data
  write($04,$01,$20)

PUB RX (dataptr, size, block) : ret_size | offset, startadd, a0, a1, counter, rdptr, temp, tempsize

  '' Call this function to receive data via the W5100.
  '' The function will return the number of bytes read.
  '' Set block == true if you want the function to block waiting for data.
  ''
  '' This function will return the actual number of bytes read.

  'Block waiting for the W5100 to tell us that there is data.
  'Technically we are reading the number of bytes that have
  'been received
  if(block == true)
    repeat
      temp := read($04,$26)
      temp := temp << 8
      temp += read($04,$27)
    while temp == 0             'Wait as long as we have received zero bytes

  'If the user wants a non-block RX call then we will return immediately
  'with a size of zero if there is no data to receive.
  else
    temp := read($04,$26)
    temp := temp << 8
    temp += read($04,$27)
    if(temp == 0)
      return 0

  'Compute the starting address
  rdptr := read($04,$28)        'Read the receive pointer
  rdptr := rdptr << 8
  rdptr += read($04,$29)
  offset := rdptr & $7FF        'Socket 0 has 2K of buffer and that determines mask here of $7FF
  startadd := $6000 + offset    'Add the offset to the starting address of socket 0

  'Determine how many bytes we need to read.
  tempsize := read_rsr
  if(tempsize > size)
    ret_size := size
  else
    ret_size := tempsize

  'Now we read the data from the W5100 and write it to the array
  'pointed to by dataptr.
  repeat counter from 0 to ret_size-1
    a1 := startadd & $FF
    a0 := (startadd & $FF00) >> 8
    byte[dataptr] := read(a0,a1)
    dataptr++
    offset++
    offset := offset & $7FF     'Socket 0 has 2K of buffer and that determines mask here of $7FF
    startadd := $6000 + offset  'Add the offset to the starting address of socket 0
  'Need to increment the rdptr by the number of bytes actually read
  rdptr += ret_size

  'Then write the value of the new pointer back
  temp := (rdptr & $FF00) >> 8
  write($04,$28,temp)
  temp := rdptr & $FF
  write($04,$29,temp)

  'Tell the W5100 that we have read the data
  write($04,$01,$40)

  'Issue the read command to the W5100 which just updates registers.
  'We will block waiting for the W5100 to finish.
  'This takes very little time but we will check it just to be sure.
  repeat
    temp := read($04,$01)
  while temp <> $00

PUB read_rsr : ret_val

  '' Read the receive size register.
  '' This will return the value in the receive size register in the W5100.
  '' A non-zero value indicates that there is data to be read.

  ret_val := read($04,$26)
  ret_val := ret_val << 8
  ret_val += read($04,$27)

PUB con_est : ret_val

  '' Call this function to determine if a TCP connection has been
  '' established. The function will return true if a connection
  '' has been established.

  if(read($04,$03) == $17)
    ret_val := true
  else
    ret_val := false

PUB read(a0, a1) : ret_val

  '' Call this function to read a byte from the W5100 via SPI.
  '' The arguments are two bytes that contains address byte 0
  '' and address byte 1. See W5100 data sheet for what registers
  '' these actually address. The function will return the byte
  '' that came via SPI from the W5100.

  'Read data from the address specified in the arguments
  add0 := a0                    'Set the arguments to the global
  add1 := a1                    'variables values
  SPIRW := SPI_RD               'Set SPIRW to the read value

  'Wait until the driver clears SPIRW. This indicates that it is done.
  repeat until SPIRW == SPI_DONE

  'The driver wrote the result to datain. Thus, that is the value
  'that we will return
  ret_val := datain

PUB write(a0, a1, dout)

  '' Call this function to write a byte from the W5100 via SPI.
  '' The arguments are three bytes that contains address byte 0,
  '' address byte 1 and the data we wish to write. See W5100
  '' data sheet for what registers these actually address.
  '' Note that the datain global variable will be overwritten
  '' during this process.

  'Write data to the address specified in the arguments
  add0 := a0                    'Set the arguments to the global
  add1 := a1                    'variables values
  dataout := dout
  SPIRW := SPI_WR               'Set SPIRW to the write value

  'Wait until the driver clears SPIRW. This indicates that it is done.
  repeat until SPIRW == SPI_DONE

PUB mode(opmode) | temp
```

```
  '' Call this function with the value wanted for opmode before
  '' the start function is called. If the start function has
  '' already been called then this function will do nothing.

  if (cogon == 0)                 'Be sure driver not already running
    DIRA |= $70000                'Set pins 16-18 to output
    temp := opmode << 16          'Shift our arg 16 bits left
    OUTA &= $FFF8FFFF             'Clear all outputs on pins 16-18
    OUTA |= opmode                'Set only required pins on pins 16-18

DAT
'******************************************************
'* Assembly for reading/writing via SPI to the W5100 *
'******************************************************
        org

entry   'Entry point for this driver. Start a new COG here.
        'Setup the IO and create masks.
        mov             t1,#0                   'Always make t1 equal to 0

        mov             ssm,#$10                'Create the mask for ss
        shl             ssm,#16

        mov             clkm,#$8                'Create the mask for clk
        shl             clkm,#16

        mov             doutm,#$20              'Create the mask for dout
        shl             doutm,#16

        mov             dinm,#$40               'Create the mask for din
        shl             dinm,#16

        mov             rstm,#$80               'Create the mask for reset
        shl             rstm,#16

        mov             t2,outa                 'Copy the outa register to t2
        or              t2,ssm                  'When ss goes output we want it high
        andn            t2,rstm                 'When reset goes output we want it low
        andn            t2,clkm                 'When clk goes output we want it low
        andn            t2,doutm                'When dout goes output we want it low
        mov             outa,t2                 'Write the modified register back to outa

        mov             t2,dira                 'Copy the dira register to t2
        or              t2,ssm                  'Set outputs
        or              t2,clkm
        or              t2,doutm
        or              t2,rstm
        mov             dira,t2                 'Write the modified value back to dira

        mov             t2,outa                 'Take the W5100 out of reset
        or              t2,rstm
        mov             outa,t2



SPILoop 'Main loop. Waits for SPIRW to be non-zero.
        'Reads arguments from main memory. Begins to set up MOSI pin.
        rdlong          t2,par                  'Grab the SPIRW value from main memory
        tjz             t2,#SPILoop             'If this value is zero then we are done

        mov             tpar,par                'Get a temporary copy of the parameter pointer
        add             tpar,#4                 'Add one to the pointer to get us to point to add0
        rdlong          ta0,tpar                'Get address0 from main memory
        add             tpar,#4                 'Add one to the pointer to get us to point to add1
        rdlong          ta1,tpar                'Get address1 from main memory
        add             tpar,#4                 'Add one to the pointer to get us to dataout
        rdlong          tdout,tpar              'Get dataout from main memory

        djnz            t2,#sbranch7            'Check to see if we should read or write
        andn            outa,doutm              'If we read then set dout to $0F
        jmp             #firstnibble

sbranch7
        or              outa,doutm              'If we write then set dout for $F0

firstnibble
        andn            outa,ssm                'Drop the SS bit
        mov             t2,#4
```

```
sloop1  'Here we will send the first byte. It is $0F for a read and $F0 for a write.
        or              outa,clkm               'Set clock high
        andn            outa,clkm               'Clear clock low
        djnz            t2,#sloop1              'Repeat this four times (first nibble is $0 or $F)

        rdlong          t2,par                  'Grab the SPIRW value from main memory
        djnz            t2,#sbranch8            'Determine what the second nibble should be
        or              outa,doutm              'Set the MOSI if we are a read operation
        jmp             #secnibble

sbranch8
        andn            outa,doutm              'Clear the MOSI if we are a write operation

secnibble
        mov             t2,#4                   'Reset the loop counter

sloop2
        or              outa,clkm               'Set clock high
        andn            outa,clkm               'Clear clock low
        djnz            t2,#sloop2              'Repeat this four times

        'Send the second byte which is the first byte of the address (address 0)
        mov             t2,#8                   'Init the loop counter
sloop3
        mov             t3,#$80                 'We will look at the MSb every time
        and             t3,ta0                  'AND the address0 value with $80
        tjz             t3,#sbranch1            'Test to see if the result is zero
        or              outa,doutm              'If not then we set the MOSI bit
        jmp             #sbranch2
sbranch1
        andn            outa,doutm              'If it is zero then we clear the MOSI bit
sbranch2
        shl             ta0,#1                  'Shift the register left one
        or              outa,clkm               'Set the clock
        andn            outa,clkm               'Clear the clock
        djnz            t2,#sloop3              'Do this eight times

        'Send the third byte which is the second byte of the address (address 1)
        mov             t2,#8                   'Init the loop counter
sloop4
        mov             t3,#$80                 'We will look at the MSb every time
        and             t3,ta1                  'AND the address0 value with $80
        tjz             t3,#sbranch3            'Test to see if the result is zero
        or              outa,doutm              'If not then we set the MOSI bit
        jmp             #sbranch4
sbranch3
        andn            outa,doutm              'If it is zero then we clear the MOSI bit
sbranch4
        shl             ta1,#1                  'Shift the register left one
        or              outa,clkm               'Set the clock
        andn            outa,clkm               'Clear the clock
        djnz            t2,#sloop4              'Do this eight times

        'The fourth byte is special. To fit this driver in under 512 bytes the read
        'and the write operations both happen here. So, even during a read operation
        'we will write the value in the dataout global variable. This is ok because
        'the W5100 will happily ignore it (as it knows we are reading). Also, when
        'performing a write operation the datain global variable will be overwritten.
        'The W5100 data sheet claims that in this case the value will always be $03.
        mov             t2,#8                   'Init the loop counter
        mov             tdin,#0                 'Clear the tdin variable
sloop5                                          'Here we write whatever value is in dataout
        mov             t3,#$80                 'We will look at the MSb every time
        and             t3,tdout                'AND the address0 value with $80
        tjz             t3,#sbranch5            'Test to see if the result is zero
        or              outa,doutm              'If not then we set the MOSI bit
        jmp             #sbranch6
sbranch5
        andn            outa,doutm              'If it is zero then we clear the MOSI bit
sbranch6
        shl             tdout,#1                'Shift the register left one
        mov             t3,dinm                 'Start shifting in data from MISO.
        or              outa,clkm               'Set the clock
        shl             tdin,#1                 'First shift tdin left. Set/clear the new LSb.
        and             t3,ina                  'Determine if P16 is set/clear
        tjz             t3,#sbranch9
        or              tdin,#1                 'If it is set then set the LSb of tdin.
sbranch9
```

```
        andn            outa,clkm               'Clear the clock
        djnz            t2,#sloop5              'Repeat this eight times

        or              outa,ssm                'Set SS. This ends the data transaction
        andn            outa,doutm              'Clear MOSI pin.

        mov             tpar,par                'Move the tdin value to the global datain memory
location
        add             tpar,#16
        wrlong          tdin,tpar

        wrlong          t1,par                  'Clear the SPIRW variable to indicate we are done
        jmp             #SPILoop                'Jump back to the main loop

        'To here is 396 bytes (includes the reserves at the bottom)

tpar    res   1         'temp variable for the par global memory pointer
t1      res   1         'temp variable 1 which is always zero
t2      res   1         'temp variable 2
t3      res   1         'temp variable 3
ta0     res   1         'temp address0
ta1     res   1         'temp address1
tdin    res   1         'temp datain
tdout   res   1         'temp dataout
ssm     res   1         'SS pin mask
clkm    res   1         'CLK pin mask
doutm   res   1         'MOSI pin mask
dinm    res   1         'MISO pin mask
rstm    res   1         'RST pin mask
```

NOTES