

# **Networked Display**

Nicholas Mackie-Jones

CSCI 297 (CS190)  
Real-time Embedded Systems

Spring 2009

## **Abstract**

The purpose of this project was to create a network-enabled text display. A ZNeo Z16 microcontroller manages the communication between a WizNET NM7010B+ Ethernet networking module and the text display, a PalmIIIxe running PalmOrb software to emulate a Matrix Orbital LK204-25 4x20 Text LCD display. The displayed text is pulled from a coordinated webserver over the Internet, using the HTTP protocol, and reformatted to fit the constrained screen. Communication with the network module is done using SPI, and RS-232 is used for communicating with the PalmIIIxe. The ZNeo Z16 is mounted on the ZNeo Contest Kit board, which provides the assets to support these communications formats.

## **Status**

I had anticipated that the most intensive parts of the software, both in terms of development time and code quantity, would be in the handling of network transmission and reception. However, the W3150A+ proved far more capable than I had anticipated, handling all of the minutia of the transmit/receive that I had been concerned about. Instead, the network connection itself and the management and parsing of the data after it had been acquired turned out to be far more intensive than anticipated.

In order to keep the display current, it needs to periodically check in with the server to discover any changes. Since these check-ins occur frequently (on the order of seconds), I expected the TCP connection to be largely persistent, with only the occasional need to manage the disconnect/reconnect procedure. Instead, while the server would permit idle connections to persist for ten minutes or more, it would typically disconnect the client immediately following any completed transaction. This meant the disconnect/reconnect/transact cycle became the central, driving part of the program rather than a uncommon subroutine, which required a rethink of most of the program flow.

While the acquisition of the data proved simpler than expected, the management of the data once it was acquired proved trickier than anticipated. The plan had been for the text processing to be simplified by having a highly predictable format, restricting the program to handling code from a known server rather than dealing with the full range of the HTTP spec. However, the HTTP spec allows intermediate servers and proxies to alter and rewrite most aspects of the transactions, necessitating the code handle a wider variety of formats lest it fail when operating outside of the development intranet.

## Specification & Implementation

### Hardware

The hardware used in this project falls into three basic modules: the microcontroller & platform, the networking module, and the display module.



#### *Microcontroller Platform - ZNeo Contest Kit*

The microcontroller used was a ZNeo Z16F2811FI20SG, on a ZNeo Contest Kit board. The ZNeo connects to the display module through the "modem" DB9 onboard serial port (P2), using the onboard UART1 and associated RS-232 level-shifter (U18, a MAX3245CAI transceiver) to format the RS-232 communications. The ZNeo has two connections to the network module, in addition to supplying it with 3.3V ( $V_{dd}$ ) power and ground. The main connection uses the onboard SPI capabilities to communicate with the module's W3150A+ chip. The other connection uses 5 GPIO pins to monitor the link status lines which originate from the module's PNY chip.

#### *Networking Module - WizNET NM7010B+*

The aforementioned networking module was the WizNET NM7010B+. It provides a RJ-45 jack and supporting hardware for interfacing with an Ethernet network, and includes a hardware TCP/IP chip which can handle much of the networking management, up to and including all of the transport reliability layers.

This networking module has three primary sub-components of interest:

- ◆ W3150A+ Hardware TCP/IP chip
- ◆ RD1-125BAG1A MAG-JACK, the RJ-45 jack and transformer
- ◆ RTL8201CP, the PNY between the MAG-JACK and the W3150A+

The vast majority of the interaction with the NM7010B+ is with the W3150A+ chip, which contains most of the programmatic components of the module.. The RD1-125BAG1A and RTL8201CP operate successfully without need for interaction, though the MAG-JACK datasheet proved useful for identifying the wiring of its integrated LEDs. Although the link status information comes from the PNY, the NM7010B+ datasheet covers the wiring in sufficient detail that the RTL8201CP datasheet was not deemed necessary (which was fortunate, as it requires formal registration and interaction with a sales representative).

The NM7010B+ used was the version covered by the v1.1 datasheet, though the v1.3 datasheet was used for development without issue. The difference is a change in the PNY chip and associated resistors, which does not have any apparent impact on the usage of the module from the perspective of this project.

The NM7010B+ provides access to all three communication modes of the W3150A+: direct bus, indirect bus, and SPI. In direct bus mode, 15 pins form a 15-bit address line and 8 pins form an 8-bit data line, and data is latched in or out using conventional Chip Select, Read, and Write lines, each with their own pin. This mode also supports outputting an interrupt signal through a dedicated output pin.

In indirect bus mode, 2 pins form a 2-bit address line and 8 pins form an 8-bit data line, along with the Chip Select, Read, Write, and Interrupt lines of the direct bus mode. The 13 unused address pins must be pulled down, as indirect mode is activated by setting a mode register located at address 0x0000, which is reachable in direct mode (the default) only all 15 pins are pulled low. The two address lines are then used to indicate address high, address low, data, and data-auto-increment-address meanings of the data register. The auto-increment-address mode, if activated in the configuration register, will automatically increment the active address after each data read or data write latching, to streamline multibyte reads and multibyte writes.

The third mode, SPI, is a new feature of the "+" revisions of the NM7010B and W3150A hardware families, and is activated by pulling a dedicated SPI\_EN enabling pin high. The pin is internally pulled low by the NM7010B+, in order to provide backwards compatibility with boards designed for the NM7010B. Beyond this pin, only standard four SPI lines are required for all addressing and data purposes. The four SPI pins are a subset of the address pins, that does not overlap with the indirect bus pins.

Though SPI communicates in 1-byte increments, the W3150A+ uses 4-byte transactions as its unit of meaningful communication. The first byte is an opcode, either Read (0x0F) or Write (0xF0). The next two bytes form the address, with the high byte first and the low byte second. Here as elsewhere, the W3150A+ communicates in MSB-first formats. The fourth byte is the data byte, which conveys the payload in write operations and is ignored in read operations. Since each byte sent in SPI necessitates a byte be returned, the W3150A+ returns the byte index in the sequence. That is, for the opcode byte, it returns 0x00; for the address high byte, it returns 0x01; and for the address low byte, it returns 0x02. The fourth byte returns 0x03 for write operations, or the requested data for read operations.

Use of the W3150A+ involves substantial configuration in the form of setting registers. This naturally includes the four bytes of its IP, four bytes of its subnet, four bytes of its default gateway, and the six bytes of its MAC/Source Hardware address. The W3150A+ supports four sockets to enable four simultaneous connections. The 8 kilobytes of receive buffer and 8 kilobytes of transmit buffer must be split between the four sockets, allocating each 0, 1k, 2k, 4k, or 8k (the default is 2k to each). Once the sizes have been decided, the calculation of the offsets for the start and stop of each buffer is straightforward. Each socket has independent registers for communication protocol (such as TCP Client, TCP Server, UDP, etc), source port, destination IP, destination port, etc as necessary for the given protocol.

TCP connections proceed through a well-documented lifecycle. Essentially, it involves initialization, followed by the connection process, then the utilization of the established connection, and various forms of voluntary or involuntary disconnection, returning to either the pre-initialized, initialized, or connecting states. The utilization during the established stage is up to the application, which can monitor the receive and transmit buffers, read or write to them when they have available content or space, as appropriate. Commands, both in terms of managing the connection and managing the buffers (such as to indicate received data has been consumed, or that the data in the transmit buffer should be sent, etc), are issued by writing to the relevant command register.

Outside of the W3150A+, the other connection to the NM7010B+ was the status lines. The status lines coming from the PNY indicate the status of the physical connection. The five lines convey the detected presence of a network, the detection of 10BaseT, the detection of 100BaseT, detection of Full or Half Duplex operation, and collision detection. All five are pulled high internally by the NM7010B+, and assert low. In the case of the duplex line, low indicates Full Duplex while high indicates Half Duplex, and in the case of the collision line, low means a collision has been detected. The 10BaseT and 100BaseT lines also are used to indicate activity, by briefly unasserting when activity is detected on the line.

#### *Display Device - PalmOrb Emulation Software LK204-25*

The display device used was a Palm IIIxe running PalmOrb 1.0-release software in order to emulate a Matrix Orbital LK204-25 Character LCD module. The Palm IIIxe, using its default PalmOS v3.5.0, is attached to a standard Palm serial cradle, which is in turn attached to the ZNeo Contest Kit's modem DB9 port.

PalmOrb (<http://palmorb.sourceforge.net/>) is a free, open-source application released under the GNU GPL version 2. It does a near-complete emulation of the LK204-25, including its 4x20 display area, 5x5 input interface, and RS-232 command interface. The most substantial omissions are that the 6 GPIO pins only output to the display and the lack of I<sup>2</sup>C communication, in addition to not supporting a few of the display-related commands (such as enabling the backlight). These omissions were not relevant for this project, and thus were no detriment.

This project used the RS-232 interface, the 4x20 display area, and display-related commands supported by both PalmOrb and the real hardware. A real Matrix Orbital LK204-25 could be connected in place of the PalmOrb-running Palm without the need for any code or compilation changes. PalmOrb was largely used in its default configuration. The RS-232 format was 8 data bits, no parity bit, and one stop bit, at 19200 baud. Curiously, the only “Device” setting that worked was “Cradle – AUTO” - the “Serial/RS232” option yielded no successful communication, neither in testing or with the ZNeo. It is suspected that this option is for more advanced Palm-family devices with additional ports and connectors, and that the cradle, although operating as a Serial/RS232 connector, presents itself differently to applications running locally on the Palm.

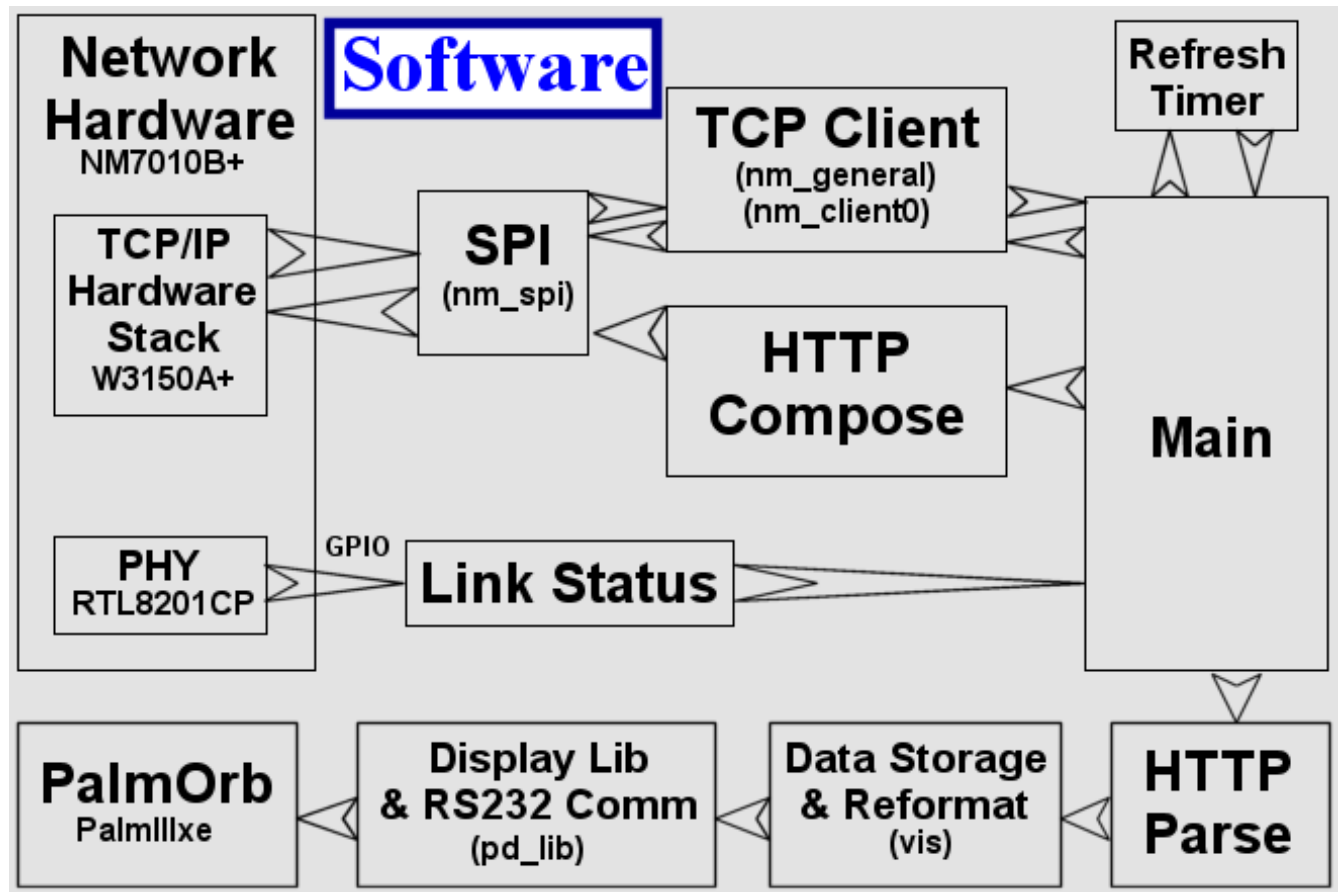
The Matrix Orbital Text LCD communication format is straightforward. Bytes other than 0xFE are display according to the internal character set. 0xFE is the command flag, indicating that the following

byte indicates a command rather than a character. Most commands only require one byte, though some require one or two parameters, which are sent after the command byte without additional 0xFE precursors. The primary commands used in this project are Clear Screen (0xFE 0x58) which clears the screen, Go Home (0xFE 0x48) which returns the cursor to the top-left corner of the display, and Set Cursor Position (0xFE 0x47 [row] [column]) which sets the cursor to the indicated row and column. The LK204-25 indices its rows and columns starting at 1 rather than 0, so the valid rows are 1,2,3,4, and similarly the valid columns are 1,2,3...18,19,20.

**Software**

The software design split the project into several modules/segments, though they fit under two broad categories. One side dealt with the networking portions, focusing on control of the network module and the sending of data. The other side dealt with the processing, storage, and display of received data. As mentioned earlier in the document, the maintenance of the network connection became central to the program, and so the main 'application' portion falls under the networking category rather than inbetween the two.

The networking portion was separated into SPI Abstraction, TCP Client, HTTP Composition, and Link Status segments, as well as the main application and refresh timer. The data and display portion was separated into HTTP Parsing, Data Storage & Reformatting, and Display & RS-232 Abstraction segments.



### *Networking*

The Net Module SPI library (`nm_spi`) was used to simplify the SPI communications, allowing the W3150A+'s 4-byte transactions to be issued with a single function call. This library was used by the TCP Client and HTTP Compose segments. The TCP Client provides methods for the setup of the networking module, connection creation and maintenance, and connection utilization (data transmission and receipt). In code, the TCP Client is split between module-wide configuration (Net Module General, `nm_general`) and tasks specific to the socket and mode used (Net Module TCP Client Socket 0, `nm_client0`). The HTTP Compose module (`HTTP_Compose`) constructs the outgoing HTTP requests, and sends them via the TCP Client data transmission functionality. This Compose module is largely an organizational creature, as its actual code is spread through several sourcefiles.

The Link Status library (`nm_linkstatus`) tracks the physical link status as indicated by the hardware's PNY module. This information is used to indicate a "no cable" state to the user, as well as providing visual feedback on the link state in debugging modes.

The main application is constantly monitoring the TCP connection state, using the TCP Client library to handle most of its actions. When not connected, it attempts to (re)connect to the server, progressing through the various connection stages in tandem with the network module. When a connection is lost, it attempts an orderly disconnect from our end as well. While connected, it checks for newly received data or data waiting to be sent. A periodic timer is used to trigger the generation and sending of HTTP Requests to acquire/update the data for display.

### *Data Display*

The HTTP responses the program receives need to be broken down into several parts. All but the first HTTP Request sent are conditional, based on the Last Modified time of the most recent page data received. Thus the HTTP Response Code must first be extracted, to establish if there is any meaningful data to parse, or if the response is merely informing us that there has been no change. Currently only these two codes are handled: 200, meaning the page's new version is present in the payload, and 304, meaning the content has not been changed since the indicated Last Modified timestamp. Assuming there is data, the new modified time must be extracted and recorded, and then the payload needs to be parsed to extract the data segments we wish to display.

HTTP uses two-byte newlines as its primary delimiter. The two bytes are 0x0D, 0x0A, aka a Carriage Return (CR) followed by a Line Feed (LF). They are used to separate each component of the Response header, including the status line and each header field in the Response, and two delimiters in a row indicates the boundary between the headers and the payload. The ordering of the header fields is arbitrary, and intermediate servers are permitted to rearrange most header fields. Intermediate servers are, in general, allowed to rewrite Response data into any other "equivalent" form. For example, since leading and trailing whitespace in header fields is typically considered ignorable by the HTTP spec, servers are free to add and remove it. Thus any parsing routine should be flexible enough to deal with these potential conditions. The parsing done by this project only accounts for some of the potential variation; examples such as unexpected whitespace are not handled in most cases by the current code.

Once extracted from the Response, the payload's data segments of interest are then passed to the Data Storage & Reformatting segment, internally referred to as "vis" for "Visualization module". This is where the raw data extracted from the HTTP Response payload is reformatted for display within the constrained environment of the 4x20 character screen, and then stored for later reference in case the display needs a full or partial refresh.

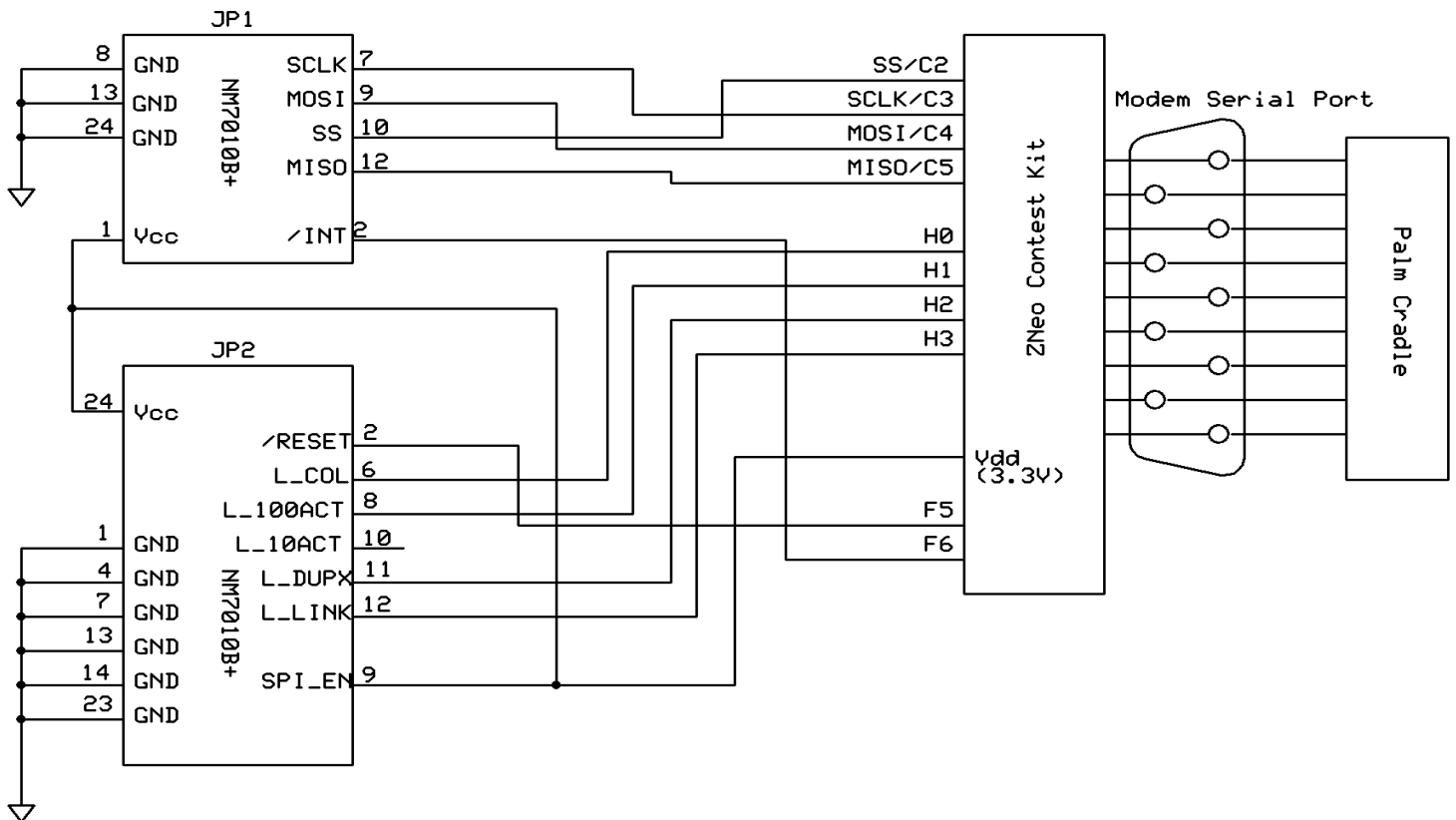
At this time, the reformatting takes two forms. First, the page title is centered using spaces as prefix padding, if the title is less than 20 characters long. Longer titles are simply truncated to 20 characters. Second, the document data is split across the two lines of the display currently used for content. If the data contains a linebreak, it attempts to split the two lines at that line break. If the data does not have a line break and is longer than one line, it attempts to wrap the one line across both (this is in part handled by the wrapping functionality of the LK204-25).

The formatted data is communicated to the display device using a Display Library, which provides a convenient text printing API to abstract the RS232 formatting and display commanding. The various line and partial line output functions use the cursor set position commands and loops with a variety of end conditions (character count, null, with/without padding) to output the source text. This is layered on top of a simple UART communications library, based on simple busywaiting.

## Construction

### *Physical Connections*

The pin assignments/connections between the NM7010B+ and ZNeo were as depicted below. The ZNeo to cradle connection is largely symbolic, intending to indicate only that the cradle connected through the DB9 modem serial port.



### *Milestones*

The development milestones chart proceeded in a very different order than had been anticipated in the proposal, but most of the milestones themselves remained unchanged

#### **Milestones chart**

##### Network Module

- Liveness and minimal functionality verifications via the 5 link status pins, and pings of the wire
- General IP configuration, verified with pings to the IP
- Outgoing TCP/IP connection configuration, verified by (contentless) connection to server while monitoring server's TCP connections
- Incoming TCP/IP connection configuration, verified by telnet and HTTP connection attempts
- HTTP Request construction, verified by server logs
- HTTP Response acceptance, verified by ZNeo memory inspection

##### Display Module

- Confirm display system (software & hardware) functionality, via LCD Smartie
- "Hello World"
- Raw data display
- Parsed data display
- Transition from mixed data/debug info to user-friendly data

The ability to verify the success of so many stages during development of the network module code was invaluable. It meant that incorrect interpretations of the documentation could be corrected early on, beneficial both for reducing deep-rooted bugs and for improving the interpretations of the documentation when developing later, more complex modules.

The LCD Smartie verification was an important stage of the display evaluation. In order to ensure that the PalmOrb-PalmOS-PalmIIIxe-Cradle hardware/software stack was properly functioning, it was first tested using a PC running LCD Driver software. In this case, the software used was LCD Smartie v5.4 (<http://lcdsmartie.sourceforge.net/>), a free and open-source program released under the GNU GPL version 2, and recommended by the PalmOrb documentation. These tests allowed for experimentation where only the Palm facet of the project was in question, without ambiguity over whether problems were due to project code or due to the Palm stack. This proved to be a valuable tool to resolve several issues in a much quicker fashion, such as the unexpected “Device” setting behavior mentioned earlier.

As awareness of the network state grew in importance (with the constant disconnect/reconnect cycles, and the realization that indicating the presence/absence of a network cable would be critical for practical use), it was determined that it would need to be part of the nominal display. However, the existing plans (for a 1-line title and 3-line body) consumed all of the available display space, and attempts at splitting the third body line between the connection status and content displays lead to more parsing and positioning complications than could be dealt with in the time available. In the end, the entire third body line (the bottom line of the display) was given over to connection status (as it had been for debugging purposes, although the content was cleaned up once it became part of the non-

debug display), which feels like a dramatic waste of space. Ultimately, the information should be able to be condensed into a handful of characters (with or without the use of custom characters), leaving the majority of the bottom line for either content or additional status information.

## Retrospective

Scoping was the focus of the central design decisions in this project, with many decisions that would be made differently with the knowledge gained afterward, going back as far as the initial project proposal composition and design. Originally, I was highly concerned about the complexity of the networking components, and I therefore scoped down most of the other facets of the project. As it turns out, the networking hardware proved to be far more capable than expected, and the W3150A+ chip handled all of the duties I was most concerned about and had anticipated spending the majority of the development time on. To compensate for this loss in complexity, I scoped up the degree of HTTP and HTML parsing I would aim for. However, text processing is one of my weak areas, and so the increased HTML parsing attempts burned time and slowed development to a crawl, and ultimately the HTML parsing code had to be scaled back and cut in order to get a reasonably reliable parsing capability.

In retrospect, I would have scoped this project much differently. There are several variations I could have pursued. The most straightforward would be to have used one of the lower-level modes of the W3150A+ chip, and done the TCP coding I had originally expected to do, ignoring the fact that the chip could do it for me. Alternatively, the input capabilities of the LK204-25 could have been included to provide an interactive display instead of the project's passive one. The 5x5 input array was discovered relatively late in development; otherwise it might have been included anyway, as an interactive display was the hobby goal that this project was derived from. The interaction could go in the direction of a “reader”, allowing the user to scroll, highlight, etc, or it could be used to reconfigure/command the network module, setting configuration options such as refresh rate, IP address, or source webpage, for example. Either avenue could have provided an ample opportunity for expandable complexity. In either case, I would retrospectively keep the HTML parsing target very minimal, and would approach the HTTP parsing in a different manner (essentially, convert the HTTP data at least partially to more familiar formats, and parse that, rather than attempting to think and parse in a purely HTTP-style environment).

One thing I learned several times over in this project is the way relatively simple systems can lie behind daunting specifications sheets. This was in several different flavors: With the W3150A+, the documentation covered the complex, low-level options alongside the higher-level ones, hiding the simplicity amid other unrelated features. With the HTTP specifications documents, the complexity came from their highly formalized terminology and formatting structure, invaluable for information density but obfuscating the very simple nature of the protocol under discussion. The Matrix Orbital documentation was filled with hardware information about various ways to power and connect to the module, while the communications portion was largely just tables of commands.

Though it makes a poor brochure bullet point, I learned a lot about the ZDS II Debugger doing this project. While I had used it to various degrees for the labs, in this project I was parsing long strings in an unfamiliar protocol (HTTP) using an unfamiliar source (NM7010B+) with an unfamiliar compiler

on a device with minimal character output ability. This forced me to use extensive amounts of memory inspection and code stepping (both through my own functions and through C/ZNeo library functions) to learn about the hardware behavior and the protocol characteristic, and to track down bugs in my text parsing routines. While I've used debuggers in other courses, and for the labs in this course, this project was the first time I truly **had** to use a debugger as a fundamental part of my development process, and it taught me a great deal about its use. Were this a real-world project, many of the unknowns could have been explored separately in a more familiar setting, such as writing a Java program to learn HTTP Request/Response details, and work in the constrained embedded environment would be done once these unknowns became understood. However, with the differing priorities of academia, exploring these topics directly on the embedded system provided a very useful experience with embedded development.