ECE 3710: µ-Controllers – Final Project
Hardware Ethernet Filter

John Call
Landon Wilcox

1. Introduction

Our project is to program a microcontroller that can intercept traffic from a common source and filter it before sending out network requests to the world. To simplify the environment, the device will connect a “client” computer connected to a “server” router. Additionally the device will only filter HTTP requests to bing.com.

2. Scope

This document contains: customer requirements, required parts for operation, detailed descriptions of the filtration methods developed, methods of testing implemented in the design process, alternatives that could be considered in similar designs, hardware and software diagrams, and source code developed for the project.

Part of the design of this project included displaying text on an LCD. Font binaries were used in the calculations for rendering text. These binaries are not included in the document, but can be found on https://github.com/Assimilater/ECE3710/tree/master/Shared/fonts. Additionally, datasheets or other information concerning components used can be found on the wiki page for this project: https://spaces.usu.edu/display/ece3710/Hardware+Ethernet+Filter.

3. Design Overview

3.1. Requirements:

a. The device will have a touchscreen interface to disable/enable internet filtering
b. The device will have an LCD display indicating the status of the filter, as well as a count of how many requests have been blocked
c. The device will allow all traffic except for requests to bing.com, and the internet will function normally when the filter is disabled (slowdown experienced because of processing on the M4 is acceptable). There is no required method of determining a filter hit.
d. The device will send a reset packet on a filter hit. Optionally, the device may send a web page indicating the original request was blocked.

3.2. Dependencies:

a. 1 Tiva-C TM4C123G Launchpad
b. 1 ER-TFTM032 LCD/Touchscreen Module
c. 2 Wiz550io modules
d. An external 3.3V source
e. A router with internet access
f. A laptop/desktop with an Ethernet port
g. Two (non-crossover) Ethernet cables
h. Wireshark (software) may prove useful

3.3. Theory of operation:

To reduce the amount of packet recalculation, the chips used by the microcontroller are given a ghost configuration. In order to obtain the necessary information for this, the client computer will connect to the router normally and the MAC/Physical address, IP Address, Subnet, and Default gateway will be obtained via ipconfig (Windows)/ifconfig (Unix) in the command window and hard-coded into the router-connected Wiz550io chip. Wireshark will be used to determine the MAC address of the default gateway, which will be hard-coded into the client-connected Wiz550io chip. If the router is accessible, MAC addresses are often printed on the router.

After this information is obtained the client and router are connected to their respective chips. The internet filter is enabled by default, and can be toggled by the user pressing a button on the touchscreen/LCD. All packets transmitted from the router are automatically passed on to the client. All packet filtering is done when the client makes a request before passing it on to the router. This reduces overall complexity, and prevents the internet from being bottlenecked by the significantly greater number of incoming packets when compared to outgoing packets.

3.4. Design Alternatives:

Filtering based on incoming packets: Not all data coming in is explicitly requested by the user. Additionally, other nefarious individuals may be intercepting traffic outside of our network before it reaches the outer internet. Thus, there might be benefit in filtering incoming traffic as well. For the purposes of this alpha model, we sought to reduce code complexity and traffic bottlenecks. Thus we chose the design philosophy of only filtering on outgoing traffic.

Using one SSI module for all peripheral devices: Electromagnetism forced us to use a separate module for the touchscreen. We found that when the microcontroller sourced a clock shared among 3 devices the electromagnetic interference would affect the data frames; for the Wiz550io chips especially. Even with our current configuration, care must be taken that the clock wire is sufficiently far away, physically, from the MISO/MOSI wires. There should be ways to overcome this; however, we found any combination of pull-down/pull-up resistors on MOSI, MISO, and the clock was insufficient.

Using interrupts instead of a busy loop: In our original design, there was only one SSI module used. To use interrupts we would have had to make use of the SSI module atomic for the 3 interrupts. This poses a problem when a higher-priority interrupt is triggered after a lower-priority interrupt begins an SPI frame. The higher-priority interrupt would be stalled waiting for the lower-priority interrupt to release the SSI module. To avoid this complexity, we elected to check the interrupt pins in a busy loop.

Other methods of filtering: The chosen method of filtration, by the conclusion of our project, was matching the destination IP address with the known IP address of bing. We also implemented the prevention of DNS-lookups associated with bing. The primary downside of this is that it does not meet the customer requirement of being able to be disabled. When the user disables the filter, the client will be able to perform the DNS lookup, and the IP address will be cached. If this happens, after the filter is re-enabled it will not prevent connections with bing.

We also implemented a deep packet search for bing. This looks at every byte in the packet. The primary downside of this is that it is especially slow, as the runtime complexity increases to O(n). Additionally not all packets associated with a request for bing contain the domain name, leaving sockets that will need to be reset.

We began looking into implementing a domain-based search that would scan HTTP request packets for bing. This would, similarly, require being able to reset the connection as a TCP socket would already be established by the time this happens. We simply ran out of time to finish implementing this method of filtration.

Sending a reset packet: The primary reasons we did not complete this requirement is time constraints in completing the project, and the amount of learning still needed. Given an extra few days we surely could have determined an acceptable implementation of this.

From our understanding, the best way to do this would be to calculate a reset packet to be forwarded to the client. This would prevent it from sending additional packets. If domain-based filtering was in use, a couple packets establishing a TCP socket connection with the remote server would have already passed the filter and a reset packet would need to be sent to the server. The packet sent to reset the server could likely be the original packet sent by the client that triggered the filter with the reset bit in the TCP header set to 1.

Redirecting to a full webpage: This would be a really nice way to inform the user the page was actually blocked, rather than invoking suspicion that the internet might not be working. However, implementing this would require significantly more research and development time, as well as more significant packet recalculation that might increase the internet latency associated with using the microcontroller. In implementing this, consideration must also be taken that not all HTTP requests are complete web pages, and replying with an HTML document may not be appropriate in all circumstances.

More Filter Options via the Touchscreen: Since we implemented a few filtration methods, and a couple more could be considered, drawing a series of additional buttons on the touchscreen to toggle the filter mode might have been a good idea. Similarly, a few methods were considered for what to do when a filter hit occurs; which could have been given control over from the touchscreen if implemented. The primary reason for this not appearing in our design, is time constraints.

4. Design Details

4.1 Connecting the LCD

a. To maintain simplicity of the driver code for the LCD, we decided to connect the 8-pin data bus to strictly one GPIO port: namely, port B.

b. Using Port B meant we had to avoid using PD0, and PD1 due to their hard-wired connection on the Launchpad. Also, PD4 and PD5 are used on the Launchpad for USB connections. This left 4 pins available on port D, a perfect number for the remaining required pins to operate the LCD: CSX, D/CX, WRX, RDX. For the exact wiring, see the schematics in Figure 1; found in **Appendix A**.

4.2 Connecting SPI

a. SPI is used to get coordinates from the touchscreen controller, and for communicating with the Wiz550io chips. The Wiz550io chips support SPI Freescale modes 0 and 3 (SPH = 0, SPO = 0 and SPH = 1, SPO = 1). The touchscreen supports Freescale mode 0 and Microwire. We configured our SSI modules use Freescale mode 0, since they both support it.

b. The SSI module on the M4 makes it convenient when generating the frames, but the FSS generated by this module does not serve as a chip select for Freescale SPI. When writing our SPI driver, we thus had it manually drive a chip select low at the beginning of the frame and bring it high again at the end of the frame.

c. Originally, we intended to use SSI0 on Port A for all three devices. However, once we added the Wiz550io chips we found that EM interference became a real issue in communications. We are still unsure of the full reasons for this, but we were able to avoid this problem by moving the touchscreen's interface to SSI1 on port F.

d. After this, we discovered that when we tried to recombine single wires for SPI into a bus (which we had scattered all over the place to reduce EM as a possibility in SPI communication errors) EM interference returned. It seems care must still be taken (after putting the touchscreen on a separate module) that the clock wire is not too close to the MOSI/MISO wires.

4.3 MISC Connections

- **a.** Our general philosophy in assigning pins was to group like functionality together.
- **b.** Fun fact: we are tight on GPIO pins for this project. Only PF4, PC6, and PC7 are free.
- **c.** The pins on port A that are not associated with SSI0 are used as reset signals for the TFT and Wiz550io chips.
- **d.** Port C is used for ready signals from the Wiz550io chips. The chip brings this pin high after its internal MCU is done getting the W5500 to a factory state.
- **e.** The first half of Port E is used for SPI Chip Select signals. The second half is used for interrupt pins that indicate there is data to send over SPI.

4.4 Coding (code found in Appendices D-F)

a. Program Logic

Refer to the flowchart Figure 3 (shown in **Appendix B**). Also refer to the **Design Alternatives** section for a synopsis on the decision to use a busy while loop to check for interrupts instead of implementing hardware interrupts.

The touchscreen handler does however use timed interrupts. This way, touch screen depress bouncing is smoothed out and the data sampled from touch screen coordinates is smoother, and covers a longer timed sample window. In order to make this function, a boolean variable *atomic_touch* is flagged volatile and the code continues in a busy wait for that variable which will change in the SysTick handler after the user has depressed.

The SysTick handler calls the LCD_GetXY with the mode set to sample as long as the interrupt is still low. After it's done it requests the average from the LCD driver and toggles the status if the coordinate is inside the boundaries of the button.

The Server and Client handlers both call the *NET_READDATA* function, which prepares a global array of packets for the handlers to iterate over. The server handler will always forward every packet. The client handler will forward every packet unless the filter is enabled and the selected filter method returns a hit. If it does, it will report the block and move on.

b. SPI Driver

Since we have multiple devices that use SPI the same way it made sense to have a common driver. After we split it up we had to modify the driver to support any SSI module. This results in the typing of the pointers passed being a little messy, but our structs handle that for us.

Additionally, when coding the enet driver, we found it useful to be able to send two separate data blocks so the enet driver could handle the generation of the address and control phase; where external code could focus on the data it was interested in. In order to facilitate this we broke up

SPI_Transfer into 3 segments that the enet driver could invoke in the order suited for that transaction.

c. LCD Driver

To reduce the number of function calls between writes, and increase refresh rates on the LCD, the driver function *LCD_WriteData* was written to accept several bytes of data which it would iteratively write to port B. To take advantage of this feature, LCD initialization codes (provided on the USU course wiki) were compressed into array constants.

To streamline writing the same set of bytes repeatedly (used when filling a square with a color), the driver function *LCD_WriteBlock* was written. We discovered that if this function was called too quickly after *LCD_SetColumn* and *LCD_SetPage* the LCD chip would not have time to finish the configuration, and some bytes would not be written to the appropriate location in the LCD memory. To avoid this, the function *LCD_FillRegion* was written, which puts in a busy wait (to avoid complications of having to use microcontroller configuration outside of the LCD driver).

The driver function *LCD_GetXY* handles all SPI transaction needs for sampling where the user is pressing on the touchscreen. It accepts an enumerated mode, and writes data out to the second parameter. It has 3 modes: reset count, collect sample, and average all samples collected.

d. Fonts

Binary fonts were provided on the USU course wiki. We used a couple different fonts in our design, so it was useful to have a common interface for them. To do this we had to modify the fonts a little bit. Some fonts had extended ascii, and others only had the characters from space to ~. So we deleted all font data outside this range.

To handle the different sizes their dimensions are hardcoded into font structures along with a pointer to the base of the array they are stored in. The *font_get* and *LCD_WriteText* driver functions are the only ones that have to deal the logic of this sizing.

The *font_get* function transforms a string, passed by user code, into an array of pointers to the beginning of each character in the font binary relating to each character in the string. This is iterated over by *LCD_WriteText*, which also handles calculating the dimensions of the region required to write this text, and printing out the pixels in the order natural to the LCD.

e. Ethernet Driver

The *NET_SPI* function is the primary data handler for the enet driver. It can be used outside of the driver, but user code will mostly likely refer to the *NET_READDATA* and *NET_WRITEPACKET* functions (details to follow). The *NET_SPI* function prepares the address and control phases of the SPI transaction described in the datasheet for the W5500. It then forwards the data (passed as a parameter) to the SPI driver. For details on how the frame is broken into these two pieces, refer to the SPI driver section **4.4.2**.

The *NET_Init* function initializes the Ethernet chips for a ghost configuration. The configuration for this, currently, has to be hardcoded (at the beginning of enet.c). This way, the filter does not need to recalculate packets based on different IP configuration or physical addresses. Refer to the **Theory of Operation** section for details on how this configuration is determined.

For both Ethernet chips, Socket 0 is configured to receive raw mac packets (by setting them for MACRAW mode) and then is opened. MacRaw mode allows the chip to see packets being transmitted that are not directed towards them. This is essential in order to allow the client to communicate with other servers on the internet. Using regular TCP, or UDP socket configurations available to the Wiz550io will only open if the user's end destination is the chip itself.

Not much documentation about MacRaw mode exists in the data sheet. We found some documentation for the W5300 to give us a baseline of understanding; but there are some differences. Through trial and error, we discovered the packets are received in the format of two bytes indicating the packet size (including the same two bytes), followed by the raw Ethernet type II frame (without the CRC checksum). In other words, the MAC Header and the payload.

When writing in MacRaw mode, it seems you must write one packet at a time, giving a Send command between each packet. You write both the MAC header and the payload, but not the two bytes indicating the size of the frame. Our driver functions, *NET_READDATA* and *NET_WRITEPACKET*, were written under these considerations.

NET_READDATA also calls a helper function to parse through all the data received from the chips into packet blocks (using the two bytes that indicate the size), and assigning pointers to important parts of the packet: the source and destination mac addresses, packet type, and beginning of packet contents.

f. Filters

We developed and finished testing three different methods of filtering data. The brute force method (and the slowest) is a deep packet inspection method. This method looks through every byte in the packet's payload, searching for the phrase 'bing'.

The second method blocks DNS requests for bing.com. This is significantly faster, and pretty effective at preventing connections to bing. However, one of the customer requirements was the ability to toggle the filter's enable status. When disabling the filter, the client has the opportunity to send a DNS request for bing. After the first successful DNS query, the IP of bing will be cached on most client computers, and future DNS queries will be unnecessary. Once the filter is enabled again, the client will be able to connect to bing using the cached IP address.

The third method, which is used in our finished product, is to look at the destination IP address. The IP address of bing is known, and can be checked. The disadvantage of this method is, in many circumstances, one server will host multiple websites; selecting which codebase is accessed by HTTP headers indicating the domain name. Thus our filter may block more than is intended. This doesn't appear to be a problem with bing, as Microsoft.com has a different host and we can't think of any other web services that would be hosted at that IP address.

We began coding other logic we could filter based on, but ran out of time to finish developing and rigorously testing these methods. For more details, refer to the **Design Alternatives** section.

5. Testing

5.1 Requirements

a. When we pressed button on the touchscreen to disable the filtering, bing would load. When we pressed the button again, bing would no longer load.

b. We observed the number change on the LCD as packets were blocked. We also observed text and color change on the display to indicate if the filter was enabled or disabled.

c. We accessed various web sites, including google and youtube. Our ability to access content from these sites was independent of whether the filter was enabled or disabled.

d. This requirement was not met due to lack of time caused by hitting unforeseen bugs in the wiring of the SPI module and unknowingly using a crossover Ethernet cable.

5.2 Other Testing

a. We were able to verify the SPI frames were generated correctly for both the TFT and the Wiz550io chips by use of the logic analyzer. Refer to the screenshots in **Appendix C**.

b. In development, we noticed the microcontroller would hard fault after several minutes of usage. After some minor code changes to add extra steps to avoid stepping out of array boundaries, these hard faults became less frequent. As of now, the device has serviced internet for several hours without hard faulting or resetting.

c. The nature of this project is easy to test in an overall sense. Most errors would result in a web page not being able to load, and the windows network icon showing a warning symbol. Thus the final test was using the internet on the device. Our laptop has been able to access the internet without resetting the microcontroller for several hours now.

d. For fun, we ran a speed test to observe the ramifications of using our microcontroller. The download speed at USU is normally above 640 Mbps and a network latency of 2ms (as tested on speedtest.net). With the microcontroller in the middle, the speed dropped to between 0.39 Mbps and 0.43 Mbps with a network latency of 6ms.

6. Conclusion

The hardware filter proved effective in preventing undesired content being delivered to the client. Currently, there is a significant impact on internet speed. This could possibly be improved upon by increasing the SPI transfer speeds, and researching into sending more than one packet at a time with the Wiz550io, or finding a faster chip. Our framework could be extended to cover more filtering methods described in the design alternatives section, as well as filtering more content than simply bing.com. This could have application in parental controls, ad blocking, and internet security.

Appendix A – Schematics

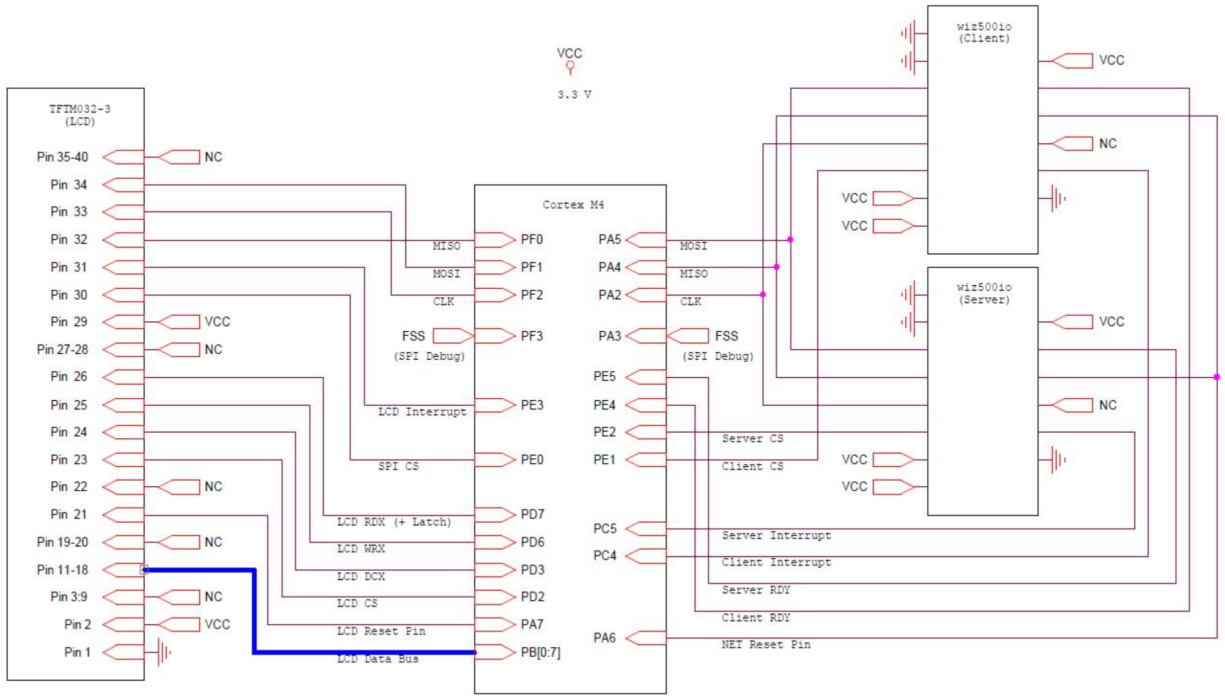

Figure 1 - Top-Level Wiring Diagram

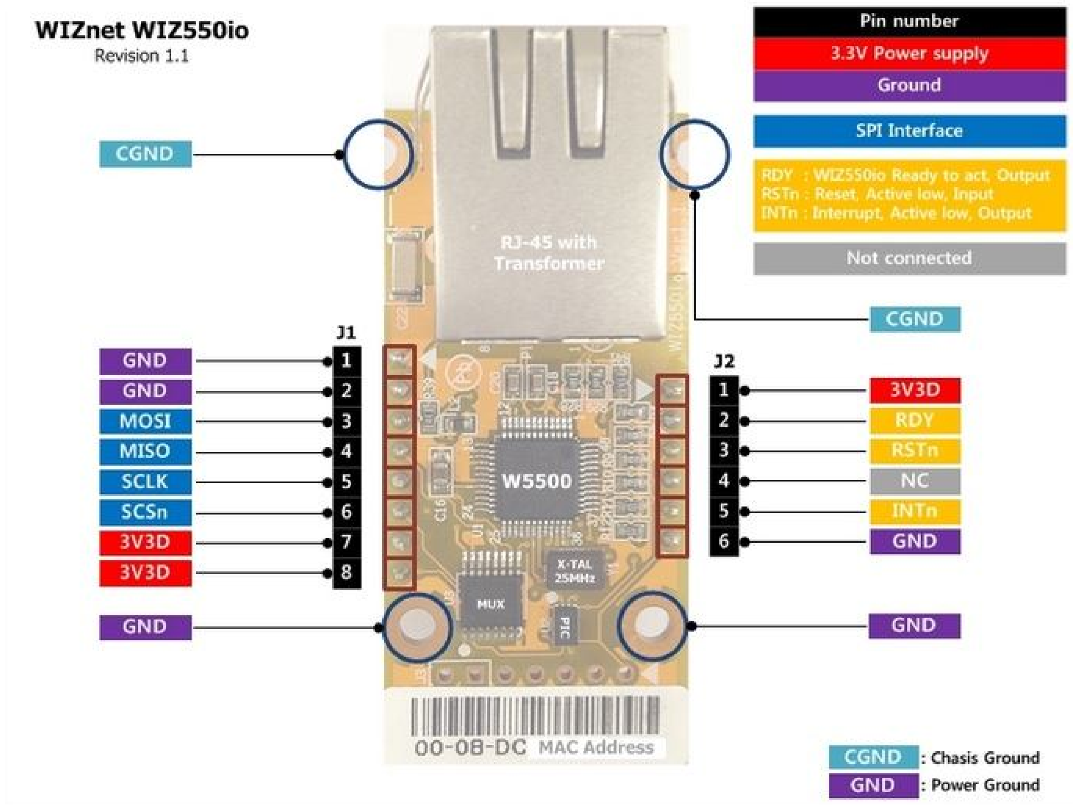

Figure 2 – Wiz500io Pin Diagram (Source: wizwiki.net)

Appendix B – Code Flow

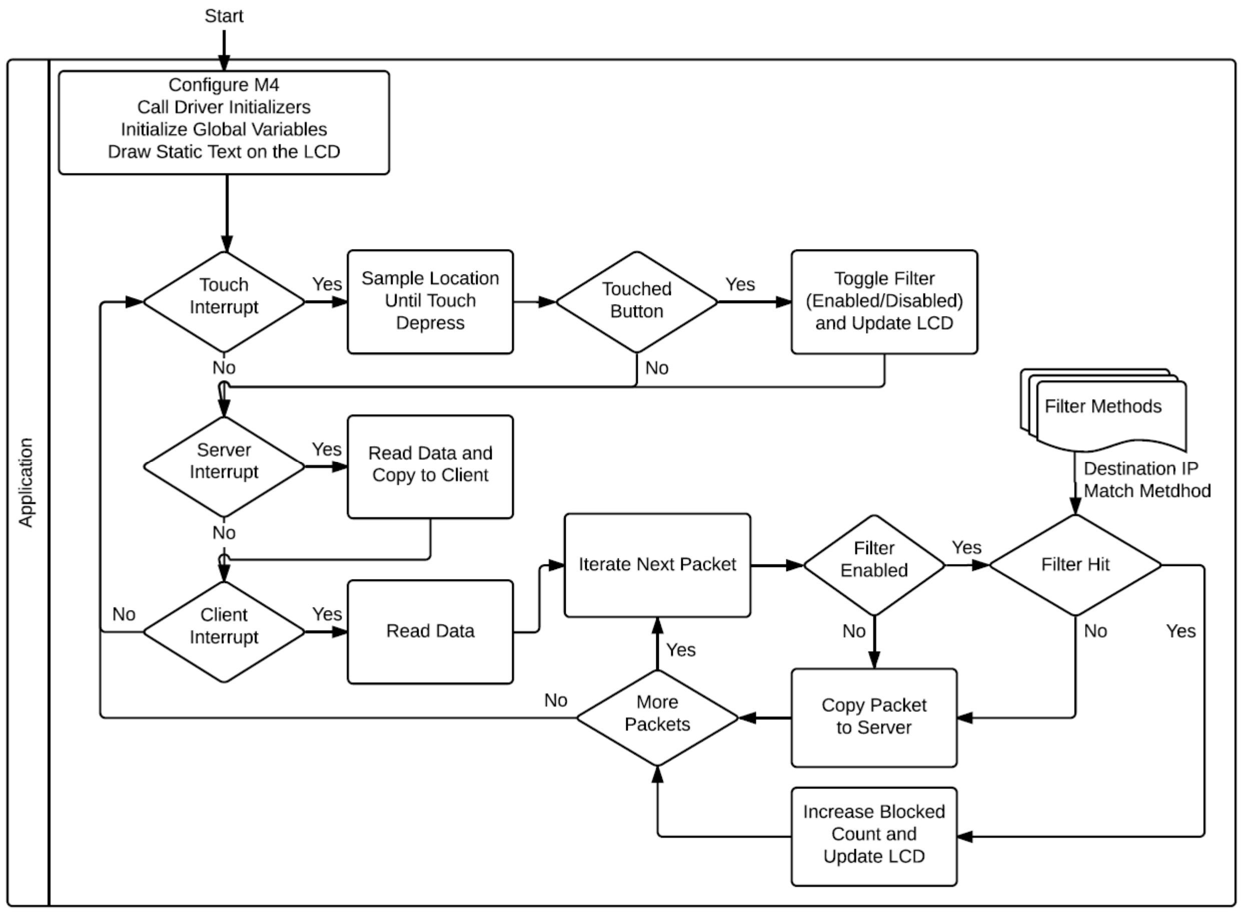

Figure 3 - Top-Level Software Model

Appendix C – Oscilloscope Tests

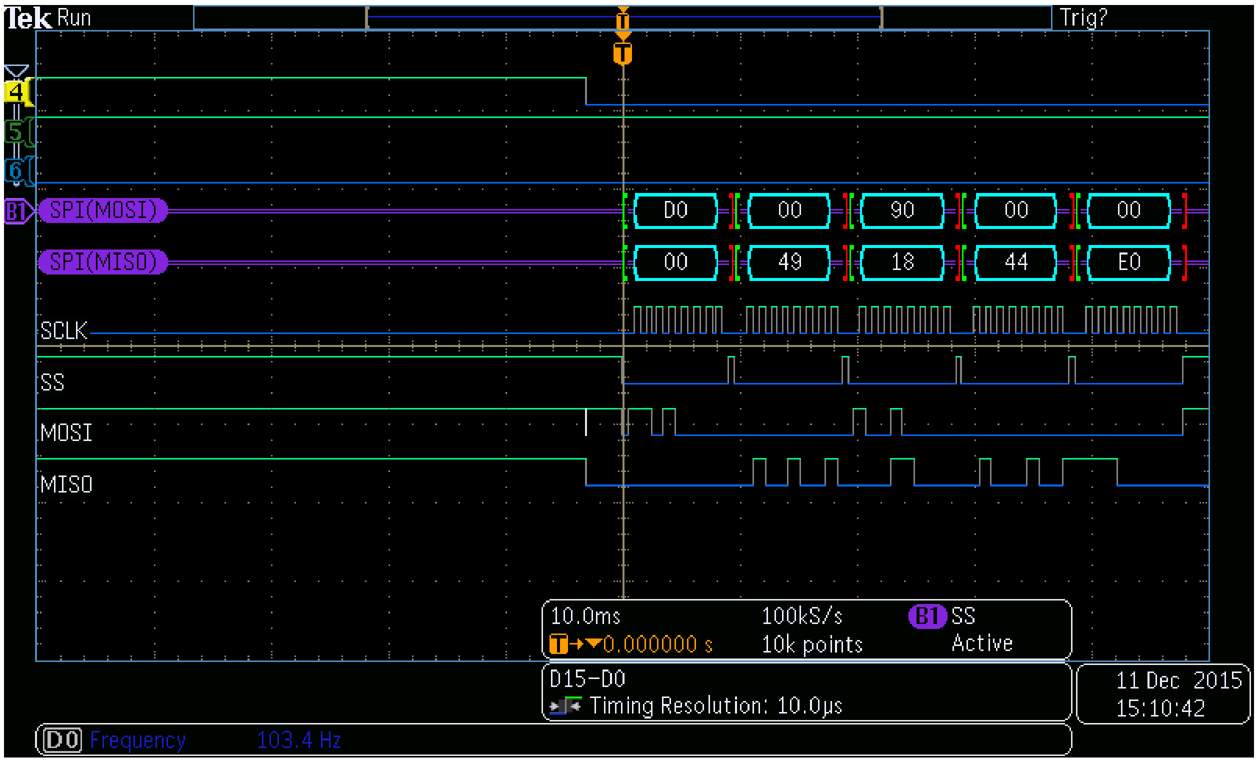

Figure 4 - Working SPI Frame for the TFT Touchscreen

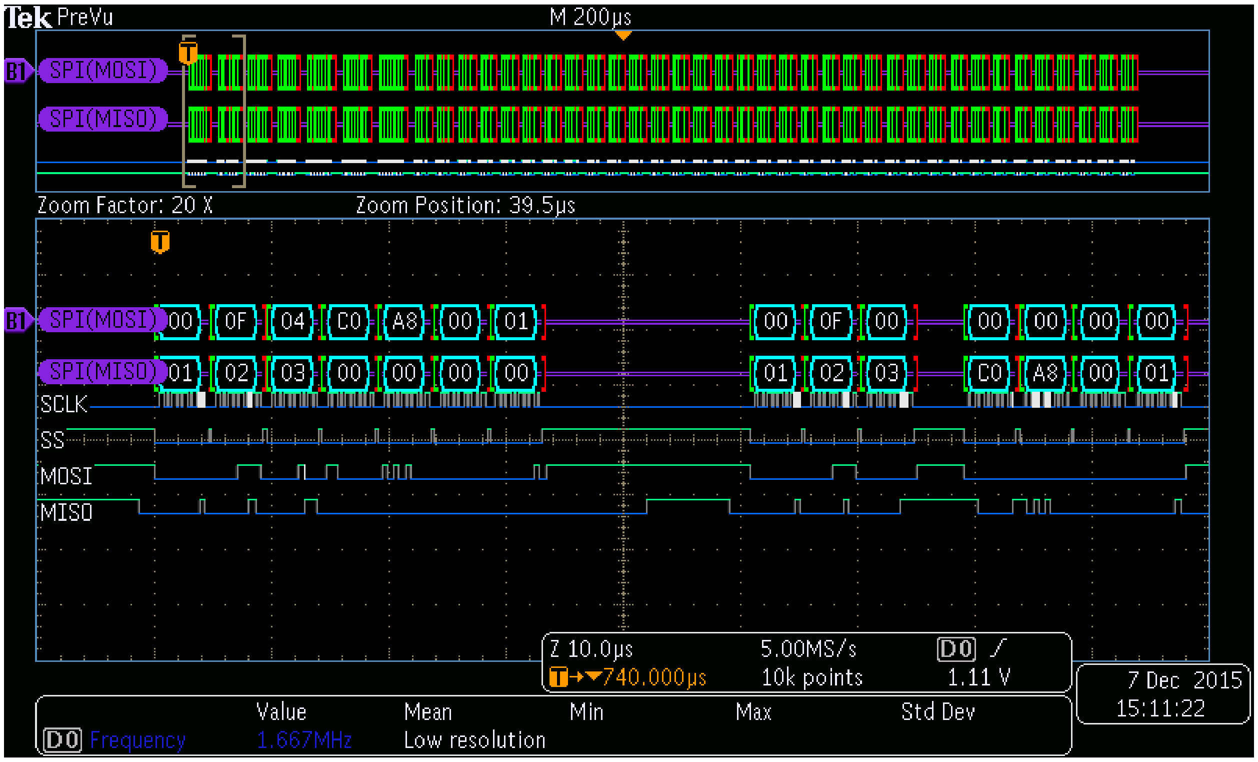
Tek PreVu
M 200μs
SPI(MOSI)
SPI(MISO)
Zoom Factor: 20 X
Zoom Position: 39.5μs
SPI(MOSI) 00 0F 04 C0 A8 00 01 00 0F 00 00 00 00 00
SPI(MISO) 01 02 03 00 00 00 00 01 02 03 C0 A8 00 01
SCLK
SS
MOSI
MISO
Z 10.0μs
740.000μs
5.00MS/s
10k points
D0 1.11 V
7 Dec 2015 15:11:22
Value Mean Min Max Std Dev
D0 Frequency 1.667MHz Low resolution

Figure 5 - Writing an IP (192.168.0.1) to the Wiz550io and reading back the same value with valid SPI frames

Appendix D – Code: Primary Application Logic

main.c:

```c
#include "../Shared/Controller.h"
#include "../Shared/LCD.h"
#include "enet.h"
#include "filter.h"

//-----------------------------------------------------------------------------------------+
// Pre-calculated constants for the dimensions of our boxes                                |
//-----------------------------------------------------------------------------------------+
const short LENGTH_OUTER = 80;
const short LENGTH_INNER = 70;

const short ROW_OUTER_Y0 = 25;
const short ROW_INNER_Y0 = 30;
const short COL_OUTER_Y0 = 80;
const short COL_INNER_Y0 = 85;

const short ROW_OUTER_YF = ROW_OUTER_Y0 + LENGTH_OUTER;
const short ROW_INNER_YF = ROW_INNER_Y0 + LENGTH_INNER;
const short COL_OUTER_YF = COL_OUTER_Y0 + LENGTH_OUTER;
const short COL_INNER_YF = COL_INNER_Y0 + LENGTH_INNER;

//-----------------------------------------------------------------------------------------+
// Status variables                                                                        |
//-----------------------------------------------------------------------------------------+
typedef enum { FILTER_DISABLED = 0, FILTER_ENABLED = 1 } State;
State enable = FILTER_DISABLED;

uint block = 0;
char blocked_s[8] = "0000000";
char status_e[17] = "Status:  Enabled";
char status_d[17] = "Status: Disabled";
TextRegion blocked_r;
TextRegion status_r;

//-----------------------------------------------------------------------------------------+
// Helper functions to easily manage the display after initialization                      |
//-----------------------------------------------------------------------------------------+
void toggleStatus() {
    static Region button_r = {
        COL_INNER_Y0, COL_INNER_YF,
        ROW_INNER_Y0, ROW_INNER_YF
    };

    if (enable) {
        enable = FILTER_DISABLED;
        status_r.Text = status_d;
        button_r.Color = LCD_COLOR_RED;
        status_r.Color = LCD_COLOR_RED;
    } else {
        enable = FILTER_ENABLED;
        status_r.Text = status_e;
        button_r.Color = LCD_COLOR_GREEN;
        status_r.Color = LCD_COLOR_GREEN;
    }
    LCD_FillRegion(button_r);
    LCD_WriteText(status_r);
}

void reportBlock() {
    uint i = 7, t = ++block, mod;
    while (i > 0) {
        mod = t % 10;
        blocked_s[--i] = '0' + mod;
        t /= 10;
    }
    LCD_WriteText(blocked_r);
}

//-----------------------------------------------------------------------------------------+
// Define Interrupt Signals                                                                |
//-----------------------------------------------------------------------------------------+
#define INT_TOUCH           BAND_GPIO_PE3
#define INT_NET_SERVER      BAND_GPIO_PE4
#define INT_NET_CLIENT      BAND_GPIO_PE5

//-----------------------------------------------------------------------------------------+
// Time based sampling of touchscreen with atomic SPI blocker                            |
//--------------------------------------------------------------------------------------+
volatile bool atomic_touch = false;
void SysTick_Handler() {
    static coord data;
    SysTick->CTRL = 0x0; // Disable Systick interrups
    if (!INT_TOUCH) { // User is pressing down, collect sample
        LCD_GetXY(TOUCH_POLL, &data);
        SysTick->CTRL = 0x3; // Enable SysTick interrupts
    } else { // User let go, get the average from samples
        if (LCD_GetXY(TOUCH_GET, &data)) { // Won't enter if-statement if no samples
            if ((COL_OUTER_Y0 < data.col) && (data.col < COL_OUTER_YF) &&
                (ROW_OUTER_Y0 < data.page) && (data.page < ROW_OUTER_YF)) {
                toggleStatus();
            }
        }

        // Signal that touch interaction is over
        atomic_touch = false;
    }
}

//--------------------------------------------------------------------------------------+
// Uses a timer to sample; so data is averaged over a longer time and touch is debounced |
// This function is blocking, so as to give priority to user input on the touchscreen    |
// Because of interrupts, this acts like a multi-threaded portion of the application     |
//--------------------------------------------------------------------------------------+
void Touch_Handler() {
    atomic_touch = true;
    SysTick->CTRL = 0x3; // Enable SysTick interrupts
    while (atomic_touch); // Wait for touch interaction to finish
}

//--------------------------------------------------------------------------------------+
// Data is always forwarded from the server                                              |
//--------------------------------------------------------------------------------------+
void NET_SERVER_Handler() {
    uint i;
    byte Int = NET_GetInterrupt(NET_CHIP_SERVER);
    if (Int & NET_INT_RECV) {
        NET_ClearInterrupt(NET_CHIP_SERVER, NET_INT_RECV);
        NET_READDATA(NET_CHIP_SERVER);
        for (i = 0; i < NET_Packets; ++i) {
            NET_WRITEPACKET(NET_CHIP_CLIENT, i);
        }
    }
}

//--------------------------------------------------------------------------------------+
// Forward data from the client if disabled or the website passes the filter             |
//--------------------------------------------------------------------------------------+
void NET_CLIENT_Handler() {
    uint i;
    byte Int = NET_GetInterrupt(NET_CHIP_CLIENT);
    if (Int & NET_INT_RECV) {
        NET_ClearInterrupt(NET_CHIP_CLIENT, NET_INT_RECV);
        NET_READDATA(NET_CHIP_CLIENT);
        for (i = 0; i < NET_Packets; ++i) {
            if (enable && Filter_IP(i)) { // Can switch out filter method here
                reportBlock();
                //Filter_Reset(i);
                //NET_WRITEPACKET(NET_CHIP_SERVER, i);
            } else {
                NET_WRITEPACKET(NET_CHIP_SERVER, i);
            }
        }
    }
}

// Forward declarations used in main
void m4config(void);
void disp_init(void);

//--------------------------------------------------------------------------------------+
// void main() - Our Program Logic:                                                      |
// Real interrupts create the need for atomic use of SPI. To avoid the complications     |
// associated with doing this, our program is a busy wait on those interrupt pins.       |
// All interrupts are active low for our application                                     |
//--------------------------------------------------------------------------------------+
int main() {
    m4config();
    LCD_Init();
    disp_init();
    NET_Init();

    while (1) {
        if (!INT_TOUCH) {
            Touch_Handler();
        }
        if (!INT_NET_SERVER) {
            NET_SERVER_Handler();
        }
        if (!INT_NET_CLIENT) {
            NET_CLIENT_Handler();
        }
    }
}

//------------------------------------------------------------------------------------------+
// Initializing the display (LCD) for our application                                      |
//------------------------------------------------------------------------------------------+
void disp_init() {
    Region button_r = {
        COL_OUTER_Y0, COL_OUTER_YF,
        ROW_OUTER_Y0, ROW_OUTER_YF,
        LCD_COLOR_BLUE
    };

    // Fill in the outer box
    LCD_FillRegion(button_r);

    // Write the static text
    blocked_r.y = 105;
    blocked_r.x = ROW_OUTER_YF + 52;
    blocked_r.Font = &fonts()->Big;
    blocked_r.Text = "Packets";
    blocked_r.BackColor = LCD_COLOR_BLACK;
    blocked_r.Color = LCD_COLOR_YELLOW;
    LCD_WriteText(blocked_r);

    blocked_r.Text = "Blocked";
    blocked_r.y -= 20;
    LCD_WriteText(blocked_r);

    // Setup for the dynamic text
    blocked_r.y = 125;
    blocked_r.x = ROW_OUTER_YF + 25;
    blocked_r.Font = &fonts()->Ubuntu;
    blocked_r.Text = blocked_s;

    // Write initial value for dynamic text
    LCD_WriteText(blocked_r);

    // Setup status text (also dynamic)
    status_r.x = 25;
    status_r.y = 175;
    status_r.Font = &fonts()->Big;
    status_r.BackColor = LCD_COLOR_BLACK;

    // Start the filter as enabled
    toggleStatus();
}

//------------------------------------------------------------------------------------------+
// Configuring the registers on the m4 for our application                                 |
//------------------------------------------------------------------------------------------+
void m4config() {
    // Enable clocks
    SYSCTL->RCGCGPIO = 0x3F; // 11 1111 => f, e, d, c, b, a
    SYSCTL->RCGCSSI = 0x3; // SSI0, SSI1
    GPIO.PortD->LOCK.word = GPIO_UNLOCK; // PD7 needs to be unlocked
    GPIO.PortF->LOCK.word = GPIO_UNLOCK; // PF0 needs to be unlocked

    // PA[0:1] => Unavailable
    // PA[2:5] => SPI
    GPIO.PortA->DEN.byte[0] = 0xFC;
    GPIO.PortA->AFSEL.byte[0] = 0x3C;
    GPIO.PortA->PDR.bit5 = 1; // Tx
    GPIO.PortA->DIR.bit6 = 1; // External NET Reset (Shared)
    GPIO.PortA->DIR.bit7 = 1; // External LCD Reset

    // PB[0:7] => LCD Data Bus
    GPIO.PortB->DEN.byte[0] = 0xFF;
    GPIO.PortB->DIR.byte[0] = 0xFF;

    // PC[0:3] => Unavailable
    // PC[4:5] => NET ready signals
    // PC[6:7] => NC
    GPIO.PortC->DEN.word |= 0xF0;
    GPIO.PortC->DIR.word &= 0x0F;

    // PD[0:1] is shared with PB[6:7] (for reasons beyond my comprehension)
    // PD[2:3,6:7] => LCD communication signals
    // PD[4:5] => Unavailable
    GPIO.PortD->CR.byte[0] = 0xFF;
    GPIO.PortD->DEN.byte[0] = 0xFF;
    GPIO.PortD->DIR.byte[0] = 0xFF;

    // PE[0:2] => CS Pins (output)
    // PE[3:5] => Interrupt Pins for SPI request (input)
    // PE[6:7] => Nonexistent
    GPIO.PortE->DEN.byte[0] = 0xFF;
    GPIO.PortE->DIR.byte[0] = 0x07;

    // SPI CS default state high
    GPIO.PortE->DATA.bit0 = 1;
    GPIO.PortE->DATA.bit1 = 1;
    GPIO.PortE->DATA.bit2 = 1;

    // PA[0:1] => Unavailable
    // PA[2:5] => SPI
    GPIO.PortF->CR.byte[0] = 0xF;
    GPIO.PortF->DEN.byte[0] = 0xF;
    GPIO.PortF->AFSEL.byte[0] = 0xF;
    GPIO.PortF->PCTL.half[0] = 0x2222;
    GPIO.PortF->PDR.bit1 = 1; // Tx

    // Configure SSI0 Freescale (CR0: SPH = 0, SPO = 0, FRF = 0)
    SSI0->CR1 = 0; // Disable
    SSI0->CC = 0x5; // Use PIOSC for the clock
    SSI0->CPSR = 0x2; // Clock divisor = 2 (the minimum/fastest)
    SSI0->CR0 = 0x307; // SCR = 3 (divisor, DSS = 7 (8-bit data)
    SSI0->CR1 |= 0x2; // Enable

    // Configure SSI1 Freescale (CR0: SPH = 0, SPO = 0, FRF = 0)
    SSI1->CR1 = 0; // Disable
    SSI1->CC = 0x5; // Use PIOSC for the clock
    SSI1->CPSR = 0x2; // Clock divisor = 2 (the minimum/fastest)
    SSI1->CR0 = 0x307; // SCR = 3 (divisor), DSS = 7 (8-bit data)
    SSI1->CR1 |= 0x2; // Enable

    // Configure Systick
    SysTick->LOAD = 16000; // 1ms
}
```

filter.h:

```
#ifndef __FILTER_H__
#define __FILTER_H__
#include "../Shared/embedded_t.h"

void Filter_Reset(uint);
bool Filter_IP(uint);
bool Filter_DNS(uint);
bool Filter_Deep(uint);


#endif
```

filter.c:

```
#include "filter.h"
#include "enet.h"

//---------------------------------------------------------------------------------------+
// Untested method for sending reset packets                                             |
//---------------------------------------------------------------------------------------+
void Filter_Reset(uint packet) {
    uint checksum;
    if (NET_Packet[packet].Payload[9] == 0x6) { // 0x6 signifies TCP
        NET_Packet[packet].Payload[33] |= 0x4; // Flag byte. 0x4 is the reset bit
        checksum = (NET_Packet[packet].Payload[36] << 8) | NET_Packet[packet].Payload[37];
        checksum += 0x4;
        checksum += checksum >> 4;
        NET_Packet[packet].Payload[36] = (checksum >> 2) & 0xFF;
        NET_Packet[packet].Payload[37] = checksum & 0xFF;
    }
}

//------------------------------------------------------------------------------------+
// Untested method for filtering on http requests (usually happen after sockets open)    |
// Note: We should send reset packets when using this method to play nice with servers   |
//------------------------------------------------------------------------------------+
bool Filter_HTTP(uint packet) {
    //uint i = 0; uint16 port_src, port_dst;
    if (NET_Packet[packet].Type == 0x0800) { // IPv4
        if (NET_Packet[packet].Payload[9] == 0x06) { // TCP protocol byte in IPv4 header
            //port_src = (NET_Packet[packet].Payload[20] << 8) | NET_Packet[packet].Payload[21];
            //port_dst = (NET_Packet[packet].Payload[22] << 8) | NET_Packet[packet].Payload[23];
            // Could check for port 80 or port 443
            // Could check for NET_Packet[packet].Payload[40] == "GET/ HTTP/1.1\0x0d0a"
        }

    }

    return false;
}

const byte IP_BING[4] = {204, 79, 197, 200};
//------------------------------------------------------------------------------------+
// Validate all IPv4 packets. Block packets with the same ip address as bing             |
// Note: This method may block undesired domains hosted on the same server               |
// (This doesn't appear to be a problem with bing.com, but it may for other domains)     |
//------------------------------------------------------------------------------------+
bool Filter_IP(uint packet) {
    if (NET_Packet[packet].Type == 0x0800) { // IPv4
        // Beginning of Destination in IPv4 header is index 16
        if (NET_Packet[packet].Payload[16] == IP_BING[0]) {
            if (NET_Packet[packet].Payload[17] == IP_BING[1]) {
                if (NET_Packet[packet].Payload[18] == IP_BING[2]) {
                    if (NET_Packet[packet].Payload[19] == IP_BING[3]) {
                        return true;
                    }
                }
            }
        }
    }

    return false;
}

//------------------------------------------------------------------------------------+
// Validate only DNS packets. Check for requests for bing's IP address                   |
// Note: If the client computer has the ip cached, this won't block a page load          |
//------------------------------------------------------------------------------------+
bool Filter_DNS(uint packet) {
    uint i;
    if (NET_Packet[packet].Type == 0x0800) { // IPv4
        if (NET_Packet[packet].Payload[9] == 0x11) { // UDP protocol byte in IPv4 header
            i = 28; // Skip the IPv4 Header and the UDP header
            // no need to check for 'b' in 'bing.com' in last 7 bytes.
            // Also, DNS Query: the last 4 bytes signify the class and type
            while(i < NET_Packet[packet].PayloadSize - 11) {
                if (NET_Packet[packet].Payload[i] == 'b') {
                    if (NET_Packet[packet].Payload[++i] == 'i') {
                        if (NET_Packet[packet].Payload[++i] == 'n') {
                            if (NET_Packet[packet].Payload[++i] == 'g') {
                                return true;
                            }
                        }
                    }
                }
            };
        }
    }

    return false;
}

//------------------------------------------------------------------------------------+
// Validate all packets. Indiscriminately search for 'bing' anywhere in the packet       |
// Note: Any filtering will reduce overall internet speed; but this is the slowest       |
//------------------------------------------------------------------------------------+
bool Filter_Deep(uint packet) {
    uint i = 0;

    while(i < NET_Packet[packet].PayloadSize - 5) {
        if (NET_Packet[packet].Payload[i] == 'b') {
            if (NET_Packet[packet].Payload[++i] == 'i') {
                if (NET_Packet[packet].Payload[++i] == 'n') {
                    if (NET_Packet[packet].Payload[++i] == 'g') {
                        return true;
                    }
                }
            }
        }
    };

    return false;
}
```

Appendix E – Code: M4 Controller Access Logic

embedded_t.h:

```
#ifndef __EMBEDDED_T_H__
#define __EMBEDDED_T_H__
#pragma anon_unions

//-----------------------------------------------------------------------------------+
// Type definitions that should be used for better, more descriptive code            |
//-----------------------------------------------------------------------------------+
typedef enum { false = 0, true = !false } bool;
typedef unsigned char byte;
typedef unsigned int uint32;
typedef unsigned short uint16;
typedef uint32 uint;
typedef byte uint8;

#endif
```

GPIO.h:

```
#ifndef _GPIO_H_
#define _GPIO_H_
#include "embedded_t.h"
#include "TM4C123GH6PM.h"

//access register via bit, byte, halfword, and word
typedef union {
    struct {
        volatile unsigned
            bit0:1,  bit1:1,  bit2:1,  bit3:1,  bit4:1,  bit5:1,  bit6:1,  bit7:1,
            bit8:1,  bit9:1,  bit10:1, bit11:1, bit12:1, bit13:1, bit14:1, bit15:1,
            bit16:1, bit17:1, bit18:1, bit19:1, bit20:1, bit21:1, bit22:1, bit23:1,
            bit24:1, bit25:1, bit26:1, bit27:1, bit28:1, bit29:1, bit30:1, bit31:1;
    };
    struct {
        volatile unsigned
            nibble0:4, nibble1:4,
            nibble2:4, nibble3:4,
            nibble4:4, nibble5:4,
            nibble6:4, nibble7:4;
    };
    volatile byte byte[4];
    volatile uint16 half[2];
    volatile uint word;
} REG;

typedef struct {
    volatile byte DATA_[1019]; //TI bit addresses
    REG DATA; //0x3FC
    REG DIR; //0x400
    REG IS; //0x404
    REG IBE; //0x408
    REG IEV; //0x40C
    REG IM; //0x410
    REG RIS; //0x414
    REG MIS; //0x418
    REG ICR; //0x41C
    REG AFSEL; //0x420
    volatile byte RES[220]; //reserved
    REG DR2R; //0x500
    REG DR4R; //0x504
    REG DR8R; //0x508
    REG ODR; //0x50C
    REG PUR; //0x510
    REG PDR; //0x514
    REG SLR; //0x518
    REG DEN; //0x51C
    REG LOCK; //0x520
    REG CR; //0x524
    REG AMSEL; // 0x528
    REG PCTL; // 0x52C
    REG ADCCTL; // 0x530
} GPIOPort;

typedef struct {
    GPIOPort* PortA;
    GPIOPort* PortB;
    GPIOPort* PortC;
    GPIOPort* PortD;
    GPIOPort* PortE;
    GPIOPort* PortF;
} GPIOBoard;

const static GPIOBoard GPIO = {
    (GPIOPort*) GPIOA_BASE,
    (GPIOPort*) GPIOB_BASE,
    (GPIOPort*) GPIOC_BASE,
    (GPIOPort*) GPIOD_BASE,
    (GPIOPort*) GPIOE_BASE,
    (GPIOPort*) GPIOF_BASE
};

#endif
```

Controller.h:

```
#ifndef __CONTROLLER_H__
#define __CONTROLLER_H__
#include "GPIO.h"

typedef volatile unsigned int* M4_DMA;

//-----------------------------------------------------------------------------------------+
//-----------------------------------------------------------------------------------------+
// SECTION: UART                                                                           |
//-----------------------------------------------------------------------------------------+
//-----------------------------------------------------------------------------------------+

//-----------------------------------------------------------------------------------------+
// UART registers (UART0)                                                                  |
//-----------------------------------------------------------------------------------------+
#define UART0_DR_R              (*((M4_DMA)0x4000C000))
#define UART0_RSR_R             (*((M4_DMA)0x4000C004))
#define UART0_ECR_R             (*((M4_DMA)0x4000C004))
#define UART0_FR_R              (*((M4_DMA)0x4000C018))
#define UART0_ILPR_R            (*((M4_DMA)0x4000C020))
#define UART0_IBRD_R            (*((M4_DMA)0x4000C024))
#define UART0_FBRD_R            (*((M4_DMA)0x4000C028))
#define UART0_LCRH_R            (*((M4_DMA)0x4000C02C))
#define UART0_CTL_R             (*((M4_DMA)0x4000C030))
#define UART0_IFLS_R            (*((M4_DMA)0x4000C034))
#define UART0_IM_R              (*((M4_DMA)0x4000C038))
#define UART0_RIS_R             (*((M4_DMA)0x4000C03C))
#define UART0_MIS_R             (*((M4_DMA)0x4000C040))
#define UART0_ICR_R             (*((M4_DMA)0x4000C044))

//-----------------------------------------------------------------------------------------+
// UART registers (UART1)                                                                  |
//-----------------------------------------------------------------------------------------+
#define UART1_DR_R              (*((M4_DMA)0x4000D000))
#define UART1_RSR_R             (*((M4_DMA)0x4000D004))
#define UART1_ECR_R             (*((M4_DMA)0x4000D004))
#define UART1_FR_R              (*((M4_DMA)0x4000D018))
#define UART1_ILPR_R            (*((M4_DMA)0x4000D020))
#define UART1_IBRD_R            (*((M4_DMA)0x4000D024))
#define UART1_FBRD_R            (*((M4_DMA)0x4000D028))
#define UART1_LCRH_R            (*((M4_DMA)0x4000D02C))
#define UART1_CTL_R             (*((M4_DMA)0x4000D030))
#define UART1_IFLS_R            (*((M4_DMA)0x4000D034))
#define UART1_IM_R              (*((M4_DMA)0x4000D038))
#define UART1_RIS_R             (*((M4_DMA)0x4000D03C))
#define UART1_MIS_R             (*((M4_DMA)0x4000D040))
#define UART1_ICR_R             (*((M4_DMA)0x4000D044))
#define UART1_CC_R              (*((M4_DMA)0x4000DFC8))

//-----------------------------------------------------------------------------------------+
// UART registers (UART2)                                                                  |
//-----------------------------------------------------------------------------------------+
#define UART2_DR_R              (*((M4_DMA)0x4000E000))
#define UART2_RSR_R             (*((M4_DMA)0x4000E004))
#define UART2_ECR_R             (*((M4_DMA)0x4000E004))
#define UART2_FR_R              (*((M4_DMA)0x4000E018))
#define UART2_ILPR_R            (*((M4_DMA)0x4000E020))
#define UART2_IBRD_R            (*((M4_DMA)0x4000E024))
#define UART2_FBRD_R            (*((M4_DMA)0x4000E028))
#define UART2_LCRH_R            (*((M4_DMA)0x4000E02C))
#define UART2_CTL_R             (*((M4_DMA)0x4000E030))
#define UART2_IFLS_R            (*((M4_DMA)0x4000E034))
```

```
#define UART2_IM_R              (*((M4_DMA)0x4000E038))
#define UART2_RIS_R             (*((M4_DMA)0x4000E03C))
#define UART2_MIS_R             (*((M4_DMA)0x4000E040))
#define UART2_ICR_R             (*((M4_DMA)0x4000E044))
#define UART2_CC_R              (*((M4_DMA)0x4000EFC8))

//-------------------------------------------------------------------------------------+
//-------------------------------------------------------------------------------------+
// SECTION: Timers                                                                     |
//-------------------------------------------------------------------------------------+
//-------------------------------------------------------------------------------------+

//-------------------------------------------------------------------------------------+
// Timer registers (TIMER0)                                                            |
//-------------------------------------------------------------------------------------+
#define TIMER0_CFG_R            (*((M4_DMA)0x40030000))
#define TIMER0_TAMR_R           (*((M4_DMA)0x40030004))
#define TIMER0_TBMR_R           (*((M4_DMA)0x40030008))
#define TIMER0_CTL_R            (*((M4_DMA)0x4003000C))
#define TIMER0_IMR_R            (*((M4_DMA)0x40030018))
#define TIMER0_RIS_R            (*((M4_DMA)0x4003001C))
#define TIMER0_MIS_R            (*((M4_DMA)0x40030020))
#define TIMER0_ICR_R            (*((M4_DMA)0x40030024))
#define TIMER0_TAILR_R          (*((M4_DMA)0x40030028))
#define TIMER0_TBILR_R          (*((M4_DMA)0x4003002C))
#define TIMER0_TAMATCHR_R       (*((M4_DMA)0x40030030))
#define TIMER0_TBMATCHR_R       (*((M4_DMA)0x40030034))
#define TIMER0_TAPR_R           (*((M4_DMA)0x40030038))
#define TIMER0_TBPR_R           (*((M4_DMA)0x4003003C))
#define TIMER0_TAPMR_R          (*((M4_DMA)0x40030040))
#define TIMER0_TBPMR_R          (*((M4_DMA)0x40030044))
#define TIMER0_TAR_R            (*((M4_DMA)0x40030048))
#define TIMER0_TBR_R            (*((M4_DMA)0x4003004C))

//-------------------------------------------------------------------------------------+
// Timer registers (TIMER1)                                                            |
//-------------------------------------------------------------------------------------+
#define TIMER1_CFG_R            (*((M4_DMA)0x40031000))
#define TIMER1_TAMR_R           (*((M4_DMA)0x40031004))
#define TIMER1_TBMR_R           (*((M4_DMA)0x40031008))
#define TIMER1_CTL_R            (*((M4_DMA)0x4003100C))
#define TIMER1_IMR_R            (*((M4_DMA)0x40031018))
#define TIMER1_RIS_R            (*((M4_DMA)0x4003101C))
#define TIMER1_MIS_R            (*((M4_DMA)0x40031020))
#define TIMER1_ICR_R            (*((M4_DMA)0x40031024))
#define TIMER1_TAILR_R          (*((M4_DMA)0x40031028))
#define TIMER1_TBILR_R          (*((M4_DMA)0x4003102C))
#define TIMER1_TAMATCHR_R       (*((M4_DMA)0x40031030))
#define TIMER1_TBMATCHR_R       (*((M4_DMA)0x40031034))
#define TIMER1_TAPR_R           (*((M4_DMA)0x40031038))
#define TIMER1_TBPR_R           (*((M4_DMA)0x4003103C))
#define TIMER1_TAPMR_R          (*((M4_DMA)0x40031040))
#define TIMER1_TBPMR_R          (*((M4_DMA)0x40031044))
#define TIMER1_TAR_R            (*((M4_DMA)0x40031048))
#define TIMER1_TBR_R            (*((M4_DMA)0x4003104C))

//-------------------------------------------------------------------------------------+
// Timer registers (TIMER2)                                                            |
//-------------------------------------------------------------------------------------+
#define TIMER2_CFG_R            (*((M4_DMA)0x40032000))
#define TIMER2_TAMR_R           (*((M4_DMA)0x40032004))
#define TIMER2_TBMR_R           (*((M4_DMA)0x40032008))
#define TIMER2_CTL_R            (*((M4_DMA)0x4003200C))
#define TIMER2_IMR_R            (*((M4_DMA)0x40032018))
#define TIMER2_RIS_R            (*((M4_DMA)0x4003201C))
#define TIMER2_MIS_R            (*((M4_DMA)0x40032020))
#define TIMER2_ICR_R            (*((M4_DMA)0x40032024))
#define TIMER2_TAILR_R          (*((M4_DMA)0x40032028))
#define TIMER2_TBILR_R          (*((M4_DMA)0x4003202C))
#define TIMER2_TAMATCHR_R       (*((M4_DMA)0x40032030))
#define TIMER2_TBMATCHR_R       (*((M4_DMA)0x40032034))
#define TIMER2_TAPR_R           (*((M4_DMA)0x40032038))
#define TIMER2_TBPR_R           (*((M4_DMA)0x4003203C))
#define TIMER2_TAPMR_R          (*((M4_DMA)0x40032040))
#define TIMER2_TBPMR_R          (*((M4_DMA)0x40032044))
#define TIMER2_TAR_R            (*((M4_DMA)0x40032048))
#define TIMER2_TBR_R            (*((M4_DMA)0x4003204C))

//-------------------------------------------------------------------------------------+
// Timer registers (TIMER3)                                                            |
//-------------------------------------------------------------------------------------+
#define TIMER3_CFG_R            (*((M4_DMA)0x40033000))
#define TIMER3_TAMR_R           (*((M4_DMA)0x40033004))
#define TIMER3_TBMR_R           (*((M4_DMA)0x40033008))
#define TIMER3_CTL_R            (*((M4_DMA)0x4003300C))
#define TIMER3_IMR_R            (*((M4_DMA)0x40033018))
#define TIMER3_RIS_R            (*((M4_DMA)0x4003301C))
#define TIMER3_MIS_R            (*((M4_DMA)0x40033020))
#define TIMER3_ICR_R            (*((M4_DMA)0x40033024))
#define TIMER3_TAILR_R          (*((M4_DMA)0x40033028))
#define TIMER3_TBILR_R          (*((M4_DMA)0x4003302C))
#define TIMER3_TAMATCHR_R       (*((M4_DMA)0x40033030))
#define TIMER3_TBMATCHR_R       (*((M4_DMA)0x40033034))
#define TIMER3_TAPR_R           (*((M4_DMA)0x40033038))
#define TIMER3_TBPR_R           (*((M4_DMA)0x4003303C))
#define TIMER3_TAPMR_R          (*((M4_DMA)0x40033040))
#define TIMER3_TBPMR_R          (*((M4_DMA)0x40033044))
#define TIMER3_TAR_R            (*((M4_DMA)0x40033048))
#define TIMER3_TBR_R            (*((M4_DMA)0x4003304C))

//-------------------------------------------------------------------------------------+
//-------------------------------------------------------------------------------------+
// SECTION: NVIC                                                                       |
//-------------------------------------------------------------------------------------+
//-------------------------------------------------------------------------------------+

//-------------------------------------------------------------------------------------+
// NVIC registers                                                                      |
//-------------------------------------------------------------------------------------+
#define NVIC_INT_TYPE_R         (*((M4_DMA)0xE000E004))
#define NVIC_ST_CTRL_R          (*((M4_DMA)0xE000E010))
#define NVIC_ST_RELOAD_R        (*((M4_DMA)0xE000E014))
#define NVIC_ST_CURRENT_R       (*((M4_DMA)0xE000E018))
#define NVIC_ST_CAL_R           (*((M4_DMA)0xE000E01C))
#define NVIC_EN0_R              (*((M4_DMA)0xE000E100))
#define NVIC_EN1_R              (*((M4_DMA)0xE000E104))
#define NVIC_DIS0_R             (*((M4_DMA)0xE000E180))
#define NVIC_DIS1_R             (*((M4_DMA)0xE000E184))
#define NVIC_PEND0_R            (*((M4_DMA)0xE000E200))
#define NVIC_PEND1_R            (*((M4_DMA)0xE000E204))
#define NVIC_UNPEND0_R          (*((M4_DMA)0xE000E280))
#define NVIC_UNPEND1_R          (*((M4_DMA)0xE000E284))
#define NVIC_ACTIVE0_R          (*((M4_DMA)0xE000E300))
#define NVIC_ACTIVE1_R          (*((M4_DMA)0xE000E304))
#define NVIC_PRI0_R             (*((M4_DMA)0xE000E400))
#define NVIC_PRI1_R             (*((M4_DMA)0xE000E404))
#define NVIC_PRI2_R             (*((M4_DMA)0xE000E408))
#define NVIC_PRI3_R             (*((M4_DMA)0xE000E40C))
#define NVIC_PRI4_R             (*((M4_DMA)0xE000E410))
#define NVIC_PRI5_R             (*((M4_DMA)0xE000E414))
#define NVIC_PRI6_R             (*((M4_DMA)0xE000E418))
#define NVIC_PRI7_R             (*((M4_DMA)0xE000E41C))
#define NVIC_PRI8_R             (*((M4_DMA)0xE000E420))
#define NVIC_PRI9_R             (*((M4_DMA)0xE000E424))
#define NVIC_PRI10_R            (*((M4_DMA)0xE000E428))
#define NVIC_CPUID_R            (*((M4_DMA)0xE000ED00))
#define NVIC_INT_CTRL_R         (*((M4_DMA)0xE000ED04))
#define NVIC_VTABLE_R           (*((M4_DMA)0xE000ED08))
#define NVIC_APINT_R            (*((M4_DMA)0xE000ED0C))
#define NVIC_SYS_CTRL_R         (*((M4_DMA)0xE000ED10))
#define NVIC_CFG_CTRL_R         (*((M4_DMA)0xE000ED14))
#define NVIC_SYS_PRI1_R         (*((M4_DMA)0xE000ED18))
#define NVIC_SYS_PRI2_R         (*((M4_DMA)0xE000ED1C))
#define NVIC_SYS_PRI3_R         (*((M4_DMA)0xE000ED20))
#define NVIC_SYS_HND_CTRL_R     (*((M4_DMA)0xE000ED24))
#define NVIC_FAULT_STAT_R       (*((M4_DMA)0xE000ED28))
#define NVIC_HFAULT_STAT_R      (*((M4_DMA)0xE000ED2C))
#define NVIC_DEBUG_STAT_R       (*((M4_DMA)0xE000ED30))
#define NVIC_MM_ADDR_R          (*((M4_DMA)0xE000ED34))
#define NVIC_FAULT_ADDR_R       (*((M4_DMA)0xE000ED38))
#define NVIC_MPU_TYPE_R         (*((M4_DMA)0xE000ED90))
#define NVIC_MPU_CTRL_R         (*((M4_DMA)0xE000ED94))
#define NVIC_MPU_NUMBER_R       (*((M4_DMA)0xE000ED98))
#define NVIC_MPU_BASE_R         (*((M4_DMA)0xE000ED9C))
#define NVIC_MPU_ATTR_R         (*((M4_DMA)0xE000EDA0))
#define NVIC_DBG_CTRL_R         (*((M4_DMA)0xE000EDF0))
#define NVIC_DBG_XFER_R         (*((M4_DMA)0xE000EDF4))
#define NVIC_DBG_DATA_R         (*((M4_DMA)0xE000EDF8))
#define NVIC_DBG_INT_R          (*((M4_DMA)0xE000EDFC))
#define NVIC_SW_TRIG_R          (*((M4_DMA)0xE000EF00))

//-------------------------------------------------------------------------------------+
//-----------------------------------------------------------------------------------------+
// SECTION: GPIO                                                                           |
//-----------------------------------------------------------------------------------------+
//-----------------------------------------------------------------------------------------+
#define GPIO_UNLOCK             0x4C4F434B

//-----------------------------------------------------------------------------------------+
// GPIO registers (PORTA)                                                                  |
//-----------------------------------------------------------------------------------------+
#define GPIO_PORTA_DATA_R       (*((M4_DMA)0x400043FC))
#define GPIO_PORTA_DIR_R        (*((M4_DMA)0x40004400))
#define GPIO_PORTA_IS_R         (*((M4_DMA)0x40004404))
#define GPIO_PORTA_IBE_R        (*((M4_DMA)0x40004408))
#define GPIO_PORTA_IEV_R        (*((M4_DMA)0x4000440C))
#define GPIO_PORTA_IM_R         (*((M4_DMA)0x40004410))
#define GPIO_PORTA_RIS_R        (*((M4_DMA)0x40004414))
#define GPIO_PORTA_MIS_R        (*((M4_DMA)0x40004418))
#define GPIO_PORTA_ICR_R        (*((M4_DMA)0x4000441C))
#define GPIO_PORTA_AFSEL_R      (*((M4_DMA)0x40004420))
#define GPIO_PORTA_DR2R_R       (*((M4_DMA)0x40004500))
#define GPIO_PORTA_DR4R_R       (*((M4_DMA)0x40004504))
#define GPIO_PORTA_DR8R_R       (*((M4_DMA)0x40004508))
#define GPIO_PORTA_ODR_R        (*((M4_DMA)0x4000450C))
#define GPIO_PORTA_PUR_R        (*((M4_DMA)0x40004510))
#define GPIO_PORTA_PDR_R        (*((M4_DMA)0x40004514))
#define GPIO_PORTA_SLR_R        (*((M4_DMA)0x40004518))
#define GPIO_PORTA_DEN_R        (*((M4_DMA)0x4000451C))
#define GPIO_PORTA_LOCK_R       (*((M4_DMA)0x40004520))
#define GPIO_PORTA_CR_R         (*((M4_DMA)0x40004524))
#define GPIO_PORTA_PCTL_R       (*((M4_DMA)0x4000452C))

//-----------------------------------------------------------------------------------------+
// GPIO registers (PORTB)                                                                  |
//-----------------------------------------------------------------------------------------+
#define GPIO_PORTB_DATA_R       (*((M4_DMA)0x400053FC))
#define GPIO_PORTB_DIR_R        (*((M4_DMA)0x40005400))
#define GPIO_PORTB_IS_R         (*((M4_DMA)0x40005404))
#define GPIO_PORTB_IBE_R        (*((M4_DMA)0x40005408))
#define GPIO_PORTB_IEV_R        (*((M4_DMA)0x4000540C))
#define GPIO_PORTB_IM_R         (*((M4_DMA)0x40005410))
#define GPIO_PORTB_RIS_R        (*((M4_DMA)0x40005414))
#define GPIO_PORTB_MIS_R        (*((M4_DMA)0x40005418))
#define GPIO_PORTB_ICR_R        (*((M4_DMA)0x4000541C))
#define GPIO_PORTB_AFSEL_R      (*((M4_DMA)0x40005420))
#define GPIO_PORTB_DR2R_R       (*((M4_DMA)0x40005500))
#define GPIO_PORTB_DR4R_R       (*((M4_DMA)0x40005504))
#define GPIO_PORTB_DR8R_R       (*((M4_DMA)0x40005508))
#define GPIO_PORTB_ODR_R        (*((M4_DMA)0x4000550C))
#define GPIO_PORTB_PUR_R        (*((M4_DMA)0x40005510))
#define GPIO_PORTB_PDR_R        (*((M4_DMA)0x40005514))
#define GPIO_PORTB_SLR_R        (*((M4_DMA)0x40005518))
#define GPIO_PORTB_DEN_R        (*((M4_DMA)0x4000551C))
#define GPIO_PORTB_LOCK_R       (*((M4_DMA)0x40005520))
#define GPIO_PORTB_CR_R         (*((M4_DMA)0x40005524))
#define GPIO_PORTB_PCTL_R       (*((M4_DMA)0x4000552C))

//-----------------------------------------------------------------------------------------+
// GPIO registers (PORTC)                                                                  |
//-----------------------------------------------------------------------------------------+
#define GPIO_PORTC_DATA_R       (*((M4_DMA)0x400063FC))
#define GPIO_PORTC_DIR_R        (*((M4_DMA)0x40006400))
#define GPIO_PORTC_IS_R         (*((M4_DMA)0x40006404))
#define GPIO_PORTC_IBE_R        (*((M4_DMA)0x40006408))
#define GPIO_PORTC_IEV_R        (*((M4_DMA)0x4000640C))
#define GPIO_PORTC_IM_R         (*((M4_DMA)0x40006410))
#define GPIO_PORTC_RIS_R        (*((M4_DMA)0x40006414))
#define GPIO_PORTC_MIS_R        (*((M4_DMA)0x40006418))
#define GPIO_PORTC_ICR_R        (*((M4_DMA)0x4000641C))
#define GPIO_PORTC_AFSEL_R      (*((M4_DMA)0x40006420))
#define GPIO_PORTC_DR2R_R       (*((M4_DMA)0x40006500))
#define GPIO_PORTC_DR4R_R       (*((M4_DMA)0x40006504))
#define GPIO_PORTC_DR8R_R       (*((M4_DMA)0x40006508))
#define GPIO_PORTC_ODR_R        (*((M4_DMA)0x4000650C))
#define GPIO_PORTC_PUR_R        (*((M4_DMA)0x40006510))
#define GPIO_PORTC_PDR_R        (*((M4_DMA)0x40006514))
#define GPIO_PORTC_SLR_R        (*((M4_DMA)0x40006518))
#define GPIO_PORTC_DEN_R        (*((M4_DMA)0x4000651C))
#define GPIO_PORTC_LOCK_R       (*((M4_DMA)0x40006520))
#define GPIO_PORTC_CR_R         (*((M4_DMA)0x40006524))
#define GPIO_PORTC_PCTL_R       (*((M4_DMA)0x4000652C))
```

```
//---------------------------------------------------------------------------------------+
// GPIO registers (PORTD)                                                                |
//---------------------------------------------------------------------------------------+
#define GPIO_PORTD_DATA_R       (*((M4_DMA)0x400073FC))
#define GPIO_PORTD_DIR_R        (*((M4_DMA)0x40007400))
#define GPIO_PORTD_IS_R         (*((M4_DMA)0x40007404))
#define GPIO_PORTD_IBE_R        (*((M4_DMA)0x40007408))
#define GPIO_PORTD_IEV_R        (*((M4_DMA)0x4000740C))
#define GPIO_PORTD_IM_R         (*((M4_DMA)0x40007410))
#define GPIO_PORTD_RIS_R        (*((M4_DMA)0x40007414))
#define GPIO_PORTD_MIS_R        (*((M4_DMA)0x40007418))
#define GPIO_PORTD_ICR_R        (*((M4_DMA)0x4000741C))
#define GPIO_PORTD_AFSEL_R      (*((M4_DMA)0x40007420))
#define GPIO_PORTD_DR2R_R       (*((M4_DMA)0x40007500))
#define GPIO_PORTD_DR4R_R       (*((M4_DMA)0x40007504))
#define GPIO_PORTD_DR8R_R       (*((M4_DMA)0x40007508))
#define GPIO_PORTD_ODR_R        (*((M4_DMA)0x4000750C))
#define GPIO_PORTD_PUR_R        (*((M4_DMA)0x40007510))
#define GPIO_PORTD_PDR_R        (*((M4_DMA)0x40007514))
#define GPIO_PORTD_SLR_R        (*((M4_DMA)0x40007518))
#define GPIO_PORTD_DEN_R        (*((M4_DMA)0x4000751C))
#define GPIO_PORTD_LOCK_R       (*((M4_DMA)0x40007520))
#define GPIO_PORTD_CR_R         (*((M4_DMA)0x40007524))
#define GPIO_PORTD_PCTL_R       (*((M4_DMA)0x4000752C))

//---------------------------------------------------------------------------------------+
// GPIO registers (PORTE)                                                                |
//---------------------------------------------------------------------------------------+
#define GPIO_PORTE_DATA_R       (*((M4_DMA)0x400243FC))
#define GPIO_PORTE_DIR_R        (*((M4_DMA)0x40024400))
#define GPIO_PORTE_IS_R         (*((M4_DMA)0x40024404))
#define GPIO_PORTE_IBE_R        (*((M4_DMA)0x40024408))
#define GPIO_PORTE_IEV_R        (*((M4_DMA)0x4002440C))
#define GPIO_PORTE_IM_R         (*((M4_DMA)0x40024410))
#define GPIO_PORTE_RIS_R        (*((M4_DMA)0x40024414))
#define GPIO_PORTE_MIS_R        (*((M4_DMA)0x40024418))
#define GPIO_PORTE_ICR_R        (*((M4_DMA)0x4002441C))
#define GPIO_PORTE_AFSEL_R      (*((M4_DMA)0x40024420))
#define GPIO_PORTE_DR2R_R       (*((M4_DMA)0x40024500))
#define GPIO_PORTE_DR4R_R       (*((M4_DMA)0x40024504))
#define GPIO_PORTE_DR8R_R       (*((M4_DMA)0x40024508))
#define GPIO_PORTE_ODR_R        (*((M4_DMA)0x4002450C))
#define GPIO_PORTE_PUR_R        (*((M4_DMA)0x40024510))
#define GPIO_PORTE_PDR_R        (*((M4_DMA)0x40024514))
#define GPIO_PORTE_SLR_R        (*((M4_DMA)0x40024518))
#define GPIO_PORTE_DEN_R        (*((M4_DMA)0x4002451C))
#define GPIO_PORTE_LOCK_R       (*((M4_DMA)0x40024520))
#define GPIO_PORTE_CR_R         (*((M4_DMA)0x40024524))
#define GPIO_PORTE_PCTL_R       (*((M4_DMA)0x4002452C))

//---------------------------------------------------------------------------------------+
// GPIO registers (PORTF)                                                                |
//---------------------------------------------------------------------------------------+
#define GPIO_PORTF_DATA_R       (*((M4_DMA)0x400253FC))
#define GPIO_PORTF_DIR_R        (*((M4_DMA)0x40025400))
#define GPIO_PORTF_IS_R         (*((M4_DMA)0x40025404))
#define GPIO_PORTF_IBE_R        (*((M4_DMA)0x40025408))
#define GPIO_PORTF_IEV_R        (*((M4_DMA)0x4002540C))
#define GPIO_PORTF_IM_R         (*((M4_DMA)0x40025410))
#define GPIO_PORTF_RIS_R        (*((M4_DMA)0x40025414))
#define GPIO_PORTF_MIS_R        (*((M4_DMA)0x40025418))
#define GPIO_PORTF_ICR_R        (*((M4_DMA)0x4002541C))
#define GPIO_PORTF_AFSEL_R      (*((M4_DMA)0x40025420))
#define GPIO_PORTF_DR2R_R       (*((M4_DMA)0x40025500))
#define GPIO_PORTF_DR4R_R       (*((M4_DMA)0x40025504))
#define GPIO_PORTF_DR8R_R       (*((M4_DMA)0x40025508))
#define GPIO_PORTF_ODR_R        (*((M4_DMA)0x4002550C))
#define GPIO_PORTF_PUR_R        (*((M4_DMA)0x40025510))
#define GPIO_PORTF_PDR_R        (*((M4_DMA)0x40025514))
#define GPIO_PORTF_SLR_R        (*((M4_DMA)0x40025518))
#define GPIO_PORTF_DEN_R        (*((M4_DMA)0x4002551C))
#define GPIO_PORTF_LOCK_R       (*((M4_DMA)0x40025520))
#define GPIO_PORTF_CR_R         (*((M4_DMA)0x40025524))
#define GPIO_PORTF_PCTL_R       (*((M4_DMA)0x4002552C))

//---------------------------------------------------------------------------------------+
// GPIO Bit-Banded Addresses                                                             |
//---------------------------------------------------------------------------------------+
// PA: 0x42000000 + 32*0x43FC = 0x42087F80
#define BAND_GPIO_PA2           (*((M4_DMA)0x42087F88))
#define BAND_GPIO_PA3           (*((M4_DMA)0x42087F8C))
```

```c
#define BAND_GPIO_PA4           (*((M4_DMA)0x42087F90))
#define BAND_GPIO_PA5           (*((M4_DMA)0x42087F94))
#define BAND_GPIO_PA6           (*((M4_DMA)0x42087F98))
#define BAND_GPIO_PA7           (*((M4_DMA)0x42087F9C))

// PB: 0x42000000 + 32*0x53FC = 0x420A7F80
#define BAND_GPIO_PB0           (*((M4_DMA)0x420A7F80))
#define BAND_GPIO_PB1           (*((M4_DMA)0x420A7F84))
#define BAND_GPIO_PB2           (*((M4_DMA)0x420A7F88))
#define BAND_GPIO_PB3           (*((M4_DMA)0x420A7F8C))
#define BAND_GPIO_PB4           (*((M4_DMA)0x420A7F90))
#define BAND_GPIO_PB5           (*((M4_DMA)0x420A7F94))
#define BAND_GPIO_PB6           (*((M4_DMA)0x420A7F98))
#define BAND_GPIO_PB7           (*((M4_DMA)0x420A7F9C))

// PC: 0x42000000 + 32*0x63FC = 0x420C7F80
#define BAND_GPIO_PC4           (*((M4_DMA)0x420C7F90))
#define BAND_GPIO_PC5           (*((M4_DMA)0x420C7F94))
#define BAND_GPIO_PC6           (*((M4_DMA)0x420C7F98))
#define BAND_GPIO_PC7           (*((M4_DMA)0x420C7F9C))

// PD: 0x42000000 + 32*0x73FC = 0x420E7F80
#define BAND_GPIO_PD2           (*((M4_DMA)0x420E7F88))
#define BAND_GPIO_PD3           (*((M4_DMA)0x420E7F8C))
#define BAND_GPIO_PD6           (*((M4_DMA)0x420E7F98))
#define BAND_GPIO_PD7           (*((M4_DMA)0x420E7F9C))

// PE: 0x42000000 + 32*0x243FC = 0x42487F80
#define BAND_GPIO_PE0           (*((M4_DMA)0x42487F80))
#define BAND_GPIO_PE1           (*((M4_DMA)0x42487F84))
#define BAND_GPIO_PE2           (*((M4_DMA)0x42487F88))
#define BAND_GPIO_PE3           (*((M4_DMA)0x42487F8C))
#define BAND_GPIO_PE4           (*((M4_DMA)0x42487F90))
#define BAND_GPIO_PE5           (*((M4_DMA)0x42487F94))

// PF: 0x42000000 + 32*0x253FC = 0x424A7F80
#define BAND_GPIO_PF0           (*((M4_DMA)0x424A7F80))
#define BAND_GPIO_PF1           (*((M4_DMA)0x424A7F84))
#define BAND_GPIO_PF2           (*((M4_DMA)0x424A7F88))
#define BAND_GPIO_PF3           (*((M4_DMA)0x424A7F8C))
#define BAND_GPIO_PF4           (*((M4_DMA)0x424A7F90))

//---------------------------------------------------------------------------------------+
//---------------------------------------------------------------------------------------+
// SECTION: MISC                                                                         |
//---------------------------------------------------------------------------------------+
//---------------------------------------------------------------------------------------+

//---------------------------------------------------------------------------------------+
// Bit-Banded Addresses For On-Board Connected Features                                  |
//---------------------------------------------------------------------------------------+
#define BOARD_LED_RED           BAND_GPIO_PF1
#define BOARD_LED_BLUE          BAND_GPIO_PF2
#define BOARD_LED_GREEN         BAND_GPIO_PF3
#define BOARD_BTN_SW1           BAND_GPIO_PF4 // Left button
#define BOARD_BTN_SW2           BAND_GPIO_PF0 // Right button

#endif
```

Appendix F – Code: SPI Driver

SPI.h:

```c
#ifndef __SPI_H__
#define __SPI_H__
#include "Controller.h"

typedef struct {
    M4_DMA CS;
    SSI0_Type* SSI;
} SPI_Route;
typedef struct {
    byte* MOSI;
    byte* MISO;
    uint N;
} SPI_Bus;
typedef struct {
    SPI_Route route;
    SPI_Bus bus;
} SPI_Frame;

void SPI_Transfer(SPI_Frame*);
void SPI_Begin(SPI_Route*);
void SPI_Block(SPI_Route*, SPI_Bus*);
void SPI_End(SPI_Route*);

#endif
```

SPI.c:

```c
#include "SPI.h"

//-------------------------------------------------------------------------------------+
// Forward declaration of functions that actually handle the SPI FIFO buffers          |
//-------------------------------------------------------------------------------------+
void SPI_ReadWrite(SPI_Route*, SPI_Bus*);
void SPI_Read(SPI_Route*, SPI_Bus*);
byte SPI_Write(SPI_Route*, SPI_Bus*);

//-------------------------------------------------------------------------------------+
// Small helpers that are used often when working with SPI                             |
//-------------------------------------------------------------------------------------+
void SPI_Wait(SSI0_Type* SSI) {
    while (!(SSI->SR & 0x1)); // Wait for TFE = 1
    while (SSI->SR & 0x10); // Wait for BSY == 0
}
byte SPI_Clear(SSI0_Type* SSI) {
    byte read;
    SPI_Wait(SSI);
    while (SSI->SR & 0x4) {
        read = SSI->DR;
    }

    // return value to get rid of compiler warning (#lazy)
    return read;
}

//-------------------------------------------------------------------------------------+
// Publicly accessible functions                                                       |
//-------------------------------------------------------------------------------------+
void SPI_Transfer(SPI_Frame* frame) {
    SPI_Begin(&frame->route);
    SPI_Block(&frame->route, &frame->bus);
    SPI_End(&frame->route);
}
void SPI_Begin(SPI_Route* route) {
    SPI_Clear(route->SSI);
    *route->CS = 0;
}
void SPI_Block(SPI_Route* route, SPI_Bus* bus) {
    if (!bus->MISO) {
        if (!bus->MOSI) { return; }
        SPI_Write(route, bus);
    } else if (!bus->MOSI) {
        SPI_Read(route, bus);
    } else {
        SPI_ReadWrite(route, bus);
    }
```

```c
}
void SPI_End(SPI_Route* route) {
    *route->CS = 1;
}

//-----------------------------------------------------------------------------------------+
// ReadWrite is the standard, read and write handle null pointer cases                    |
//-----------------------------------------------------------------------------------------+
void SPI_ReadWrite(SPI_Route* route, SPI_Bus* bus) {
    uint i, j, r;

    // Sending in groups of 8 (SPI FIFO Buffer Size)
    i = 0;
    for (r = bus->N; r > 8; r -= 8) {
        for (j = 0; j < 8; ++j) {
            route->SSI->DR = bus->MOSI[i + j];
        }
        SPI_Wait(route->SSI);
        for (j = 0; j < 8; ++j) {
            bus->MISO[i + j] = route->SSI->DR;
        }
        i += 8;
    }

    // Send all remaining groups
    for (j = 0; j < r; ++j) {
        route->SSI->DR = bus->MOSI[i + j];
    }
    SPI_Wait(route->SSI);
    for (j = 0; j < r; ++j) {
        bus->MISO[i + j] = route->SSI->DR;
    }
}
void SPI_Read(SPI_Route* route, SPI_Bus* bus) {
    uint i, j, r;
    byte dummy = 0;

    // Sending in groups of 8 (SPI FIFO Buffer Size)
    i = 0;
    for (r = bus->N; r > 8; r -= 8) {
        for (j = 0; j < 8; ++j) {
            route->SSI->DR = dummy;
        }
        SPI_Wait(route->SSI);
        for (j = 0; j < 8; ++j) {
            bus->MISO[i + j] = route->SSI->DR;
        }
        i += 8;
    }

    // Send all remaining groups
    for (j = 0; j < r; ++j) {
        route->SSI->DR = dummy;
    }
    SPI_Wait(route->SSI);
    for (j = 0; j < r; ++j) {
        bus->MISO[i + j] = route->SSI->DR;
    }
}
byte SPI_Write(SPI_Route* route, SPI_Bus* bus) {
    uint i, j, r;
    byte dummy;

    // Sending in groups of 8 (SPI FIFO Buffer Size)
    i = 0;
    for (r = bus->N; r > 8; r -= 8) {
        for (j = 0; j < 8; ++j) {
            route->SSI->DR = bus->MOSI[i + j];
        }
        SPI_Wait(route->SSI);
        for (j = 0; j < 8; ++j) {
            dummy = route->SSI->DR;
        }
        i += 8;
    }

    // Send all remaining groups
    for (j = 0; j < r; ++j) {
        route->SSI->DR = bus->MOSI[i + j];
    }
    SPI_Wait(route->SSI);
    for (j = 0; j < r; ++j) {
        dummy = route->SSI->DR;
    }

    // return value to get rid of compiler warning (#lazy)
    return dummy;
}
```

Appendix G – Code: LCD/Touchscreen Driver With Fonts

fonts.h:

```c
#ifndef __FONTS_H__
#define __FONTS_H__
#include "embedded_t.h"

typedef struct {
    byte* root;
    byte width, height;
} font;

typedef struct {
    uint n;
    byte* s[25];
} text;

typedef struct {
    font Ubuntu;
    font Terminus;
    font Small;
    font Big;
    font _8x8;
    font Basic_8x8;
} FontList;

const FontList* fonts(void);
text font_get(const font*, const char*);

#endif
```

Fonts were obtained from the USU ECE 3710 class wiki; slight alterations were made. For individual font headers see: https://github.com/Assimilater/ECE3710/tree/master/Shared/fonts

fonts.c:

```c
#include "fonts.h"
#include "fonts/Ubuntu.h"
#include "fonts/TerminusFont.h"
#include "fonts/SmallFont.h"
#include "fonts/font8x8_basic.h"
#include "fonts/font8x8.h"
#include "fonts/BigFont.h"
#include <string.h> // strlen
//#include <stdlib.h> // malloc

//---------------------------------------------------------------------------------------+
// My implementation of the singleton for accessing fonts                                |
//---------------------------------------------------------------------------------------+
const FontList* fonts() {
    static FontList list;
    static byte init = 0;
    if (!init) {
        // Ubuntu
        list.Ubuntu.root = (byte*) Ubuntu;
        list.Ubuntu.width = 3; // (24 x 32) / 8 bit char
        list.Ubuntu.height = 32;

        // Terminus
        list.Terminus.root = (byte*) TerminusFont;
        list.Terminus.width = 1; // (8 x 12) / 8 bit char
        list.Terminus.height = 12;

        // Small
        list.Small.root = (byte*) SmallFont;
        list.Small.width = 1; // (8 x 12) / 8 bit char
        list.Small.height = 12;

        // Big
        list.Big.root = (byte*) BigFont;
        list.Big.width = 2; // (16 x 16) / 8 bit char
        list.Big.height = 16;

        // 8x8
        list.Basic_8x8.root = (byte*) font8x8_basic;
        list.Basic_8x8.width = 1; // (8 x 8) / 8 bit char
        list.Basic_8x8.height = 8;
        // 8x8_basic
        list._8x8.root = (byte*) font8x8;
        list._8x8.width = 1; // (8 x 8) / 8 bit char
        list._8x8.height = 8;

        init = 1;
    }

    return &list;
}

//-----------------------------------------------------------------------------------------+
// Transform c-string into array of pointers to font characters                           |
//-----------------------------------------------------------------------------------------+
text font_get(const font* font, const char* val) {
    int i;
    char c;
    text d;

    // Rather than use dynamic memory, max out the size to 25
    //d.s = malloc(sizeof(byte*) * d.n);
    d.n = strlen(val);
    if (d.n > 25) { d.n = 25; }
    for (i = 0; i < d.n; ++i) {
        c = val[i];
        c = (c < ' ' || c > '~')
            ? 0 // use space for unknown characters (ie extended ascii)
            : c - ' ';

        d.s[i] = font->root + (c * font->width * font->height);
    }
    return d;
}
```

LCD.h:

```
#ifndef __TFTM032_H__
#define __TFTM032_H__
#include "fonts.h"

//-----------------------------------------------------------------------------------------+
// Color Codes For The LCD                                                                 |
//-----------------------------------------------------------------------------------------+
#define SIZE_COLOR 2
extern const byte LCD_COLOR_WHITE[SIZE_COLOR];
extern const byte LCD_COLOR_BLACK[SIZE_COLOR];
extern const byte LCD_COLOR_GREY[SIZE_COLOR];
extern const byte LCD_COLOR_BLUE[SIZE_COLOR];
extern const byte LCD_COLOR_RED[SIZE_COLOR];
extern const byte LCD_COLOR_MAGENTA[SIZE_COLOR];
extern const byte LCD_COLOR_GREEN[SIZE_COLOR];
extern const byte LCD_COLOR_CYAN[SIZE_COLOR];
extern const byte LCD_COLOR_YELLOW[SIZE_COLOR];

//-----------------------------------------------------------------------------------------+
// Communication Signals for the LCD                                                       |
//-----------------------------------------------------------------------------------------+
#define LCD_CSX BAND_GPIO_PD2
#define LCD_DCX BAND_GPIO_PD3
#define LCD_WRX BAND_GPIO_PD6
#define LCD_RDX BAND_GPIO_PD7

#define LCD_RST BAND_GPIO_PA7
#define TP_CSX BAND_GPIO_PE0

//-----------------------------------------------------------------------------------------+
// LCD Global Constants                                                                    |
//-----------------------------------------------------------------------------------------+
static const unsigned short LCD_COLS = 240;
static const unsigned short LCD_ROWS = 320;
static const unsigned int LCD_AREA = LCD_COLS * LCD_ROWS;

typedef struct {
    unsigned short ColumnStart, ColumnEnd, PageStart, PageEnd;
    const byte* Color;
} Region;

typedef struct {
    unsigned short x, y;
    const font* Font;
    const byte* BackColor;
    const byte* Color;
    char* Text;
} TextRegion;

typedef struct {
    unsigned short col, page;
} coord;

typedef enum {
    TOUCH_RESET,
    TOUCH_POLL,
    TOUCH_GET,
} SAMPLE_MODE;

//-----------------------------------------------------------------------------------+
// Driver Functions                                                                  |
//-----------------------------------------------------------------------------------+
bool LCD_GetXY(SAMPLE_MODE, coord*);
void LCD_WaitChip(void);
void LCD_WriteCmd(const byte);
void LCD_WriteData(const byte*, const int);
void LCD_WriteBlock(const byte*, const int, const int);
void LCD_WriteText(const TextRegion);

void LCD_SetColumn(const unsigned short, const unsigned short);
void LCD_SetPage(const unsigned short, const unsigned short);
void LCD_FillRegion(const Region);

void LCD_Init(void);

#endif
```

LCD.c:

```
#include "../Shared/Controller.h"
#include "LCD.h"
#include "SPI.h"

//-----------------------------------------------------------------------------------+
// Color Codes For The LCD                                                           |
//-----------------------------------------------------------------------------------+
const byte LCD_COLOR_WHITE      [SIZE_COLOR] = {0xFF, 0xFF};
const byte LCD_COLOR_BLACK      [SIZE_COLOR] = {0x00, 0x01};
const byte LCD_COLOR_GREY       [SIZE_COLOR] = {0xF7, 0xDE};
const byte LCD_COLOR_BLUE       [SIZE_COLOR] = {0x00, 0x1F};
const byte LCD_COLOR_RED        [SIZE_COLOR] = {0xF8, 0x00};
const byte LCD_COLOR_MAGENTA    [SIZE_COLOR] = {0xF8, 0x1F};
const byte LCD_COLOR_GREEN      [SIZE_COLOR] = {0x07, 0xE0};
const byte LCD_COLOR_CYAN       [SIZE_COLOR] = {0x7F, 0xFF};
const byte LCD_COLOR_YELLOW     [SIZE_COLOR] = {0xFF, 0xE0};

//-----------------------------------------------------------------------------------+
// Experimentally determined codes given by the TFT at the outermost coordinates     |
//-----------------------------------------------------------------------------------+
#define TOUCH_MAX_COL (uint16)0xEDF;
#define TOUCH_MAX_ROW (uint16)0xFFF;

//-----------------------------------------------------------------------------------+
// Uses SPI to interact witht the touchscreen chip and retrieve coordinates          |
//-----------------------------------------------------------------------------------+
bool LCD_GetXY(SAMPLE_MODE mode, coord* val) {
    const uint SAMPLE_SIZE = 100;
    static coord data[SAMPLE_SIZE];

    // Vars used for poll
    byte miso[5], mosi[5] = {0xD0, 0, 0x90, 0, 0};
    SPI_Frame frame;
    coord* poll;

    // Vars used for get (averaging)
    static uint8 sample = 0;
    uint i, n, col, page;

    if (mode == TOUCH_RESET) {
        sample = 0; // Resets the sampling
    } else if (mode == TOUCH_GET) {
        //-----------------------------------------------------------------------------------+
        // Calculates the average from the sample and returns scaled coord based on LCD size |
        //-----------------------------------------------------------------------------------+
        n = sample < SAMPLE_SIZE ? sample : SAMPLE_SIZE;
```

```c
        sample = 0; // Reset the count now that we've grabbed a value
        if (!n) { return false; } // Report there is no data to average

        // Perform the average
        col = page = 0;
        for (i = 0; i < n; ++i) {
            col += data[i].col;
            page += data[i].page;
        }
        col /= n;
        page /= n;

        // Scale to range in pixels
        col = (col * LCD_COLS) / TOUCH_MAX_COL;
        page = (page * LCD_ROWS) / TOUCH_MAX_ROW;

        // Update the caller with the proper coordinate
        val->col = col;
        val->page = page;
    } else if (mode == TOUCH_POLL) {
        //--------------------------------------------------------------------------------+
        // Uses SPI to interact with the touchscreen chip and retrieve coordinates        |
        //--------------------------------------------------------------------------------+
        frame.route.CS = &TP_CSX;
        frame.route.SSI = SSI1;
        frame.bus.MOSI = mosi;
        frame.bus.MISO = miso;
        frame.bus.N = 5;
        SPI_Transfer(&frame);

        // Read = X C[11:0] X[2:0] (where X is don't care)
        poll = &data[sample++ % SAMPLE_SIZE];
        poll->col  = ((miso[1] << 8 | miso[2]) >> 3) & 0xFFF;
        poll->page = ((miso[3] << 8 | miso[4]) >> 3) & 0xFFF;

    } else { return false; } // Unrecognized command
    return true; // Inidicate success
}

//--------------------------------------------------------------------------------+
// A busy wait that should provide the LCD adequate time to respond to commands   |
//--------------------------------------------------------------------------------+
void LCD_WaitChip() {
    const static int delay = 100000;
    int i = 0;
    for (; i < delay; ++i);
}

//--------------------------------------------------------------------------------+
// Writes a command code to the LCD                                               |
//--------------------------------------------------------------------------------+
void LCD_WriteCmd(const byte cmd) {
    LCD_CSX = 0; // CSX "LCD, pay attention!"
    LCD_DCX = 0; // DCX cmd
    LCD_WRX = 0; // WRX
    LCD_RDX = 1; // Set Read high
    GPIO.PortB->DATA.byte[0] = cmd;
    LCD_WRX = 1; // WRX read on +edge
    LCD_CSX = 1; // "We're done, LCD"
}

//--------------------------------------------------------------------------------+
// Writes a data stream of arbitrary size to the LCD (high screen refresh speeds) |
//--------------------------------------------------------------------------------+
void LCD_WriteData(const byte* data, const int len) {
    int i;
    LCD_CSX = 0; // CSX "LCD, pay attention!"
    LCD_DCX = 1; // DCX Data
    LCD_RDX = 1; // Set Read high
    for (i = 0; i < len; ++i) {
        LCD_WRX = 0; // WRX
        GPIO.PortB->DATA.byte[0] = data[i];
        LCD_WRX = 1; // WRX read on +edge
    }
    LCD_CSX = 1; // "We're done, LCD"
}

//--------------------------------------------------------------------------------+
// Primarily used by FillRegion when the same data stream needs repeated sends    |
//--------------------------------------------------------------------------------+
void LCD_WriteBlock(const byte* data, const int len, const int n) {
    int i, j;
    LCD_WriteCmd(0x2C);
    LCD_CSX = 0; // CSX "LCD, pay attention!"
    LCD_DCX = 1; // DCX Data
    LCD_RDX = 1; // Set Read high
    for (j = 1; j < n; ++j) {
        for (i = 0; i < len; ++i) {
            LCD_WRX = 0; // WRX
            GPIO.PortB->DATA.byte[0] = data[i];
            LCD_WRX = 1; // WRX read on +edge
        }
    }
    LCD_CSX = 1; // "We're done, LCD"
}

//-------------------------------------------------------------------------------------+
// Sets column bounds on the LCD                                                       |
//-------------------------------------------------------------------------------------+
void LCD_SetColumn(const unsigned short Start, const unsigned short End) {
    byte bStream[2];
    LCD_WriteCmd(0x2A);

    bStream[0] = (Start & 0xFF00) >> 8;
    bStream[1] = (Start & 0x00FF);
    LCD_WriteData(bStream, 2);

    bStream[0] = (End & 0xFF00) >> 8;
    bStream[1] = (End & 0x00FF);
    LCD_WriteData(bStream, 2);
}

//-------------------------------------------------------------------------------------+
// Sets page bounds on the LCD                                                         |
//-------------------------------------------------------------------------------------+
void LCD_SetPage(const unsigned short Start, const unsigned short End) {
    byte bStream[2];
    LCD_WriteCmd(0x2B);

    bStream[0] = (Start & 0xFF00) >> 8;
    bStream[1] = (Start & 0x00FF);
    LCD_WriteData(bStream, 2);

    bStream[0] = (End & 0xFF00) >> 8;
    bStream[1] = (End & 0x00FF);
    LCD_WriteData(bStream, 2);
}

//-------------------------------------------------------------------------------------+
// Fills an area on the LCD to a solid color                                           |
//-------------------------------------------------------------------------------------+
void LCD_FillRegion(const Region r) {
    LCD_SetColumn(r.ColumnStart, r.ColumnEnd);
    LCD_SetPage(r.PageStart, r.PageEnd);
    LCD_WaitChip(); // give the controller time to configure the page

    LCD_WriteBlock(r.Color, SIZE_COLOR, (r.ColumnEnd - r.ColumnStart) * (r.PageEnd - r.PageStart));
}

//-------------------------------------------------------------------------------------+
// Writes text horizontally, with x-y referring the bottom left corner of the text     |
//-------------------------------------------------------------------------------------+
void LCD_WriteText(const TextRegion r) {
    byte temp;
    int row, letter, col, bitmask;
    text t = font_get(r.Font, r.Text);

    // We're writing horizontally (x is page-wise)
    LCD_SetColumn(r.y, r.y + r.Font->height - 1);
    LCD_SetPage(r.x, r.x + t.n * r.Font->width * 8);
    LCD_WaitChip(); // give the controller time to configure the page
    LCD_WriteCmd(0x2C); // See LCD_WriteBlock

    for (letter = 0; letter < t.n; ++letter) {
        for (col = 0; col < r.Font->width; ++col) {
            for (bitmask = 0x80; bitmask > 0; bitmask = bitmask >> 1) {
                for (row = r.Font->height - 1; row >= 0; --row) {
                    // The font rows are inline in memory so this reads like assembly array access
                    temp = t.s[letter][row * r.Font->width + col];
                    if (temp & bitmask) {
                        LCD_WriteData(r.Color, SIZE_COLOR);
                    } else {
                        LCD_WriteData(r.BackColor, SIZE_COLOR);
                    }
                }
            }
        }
    }
}

//-----------------------------------------------------------------------------------------+
// Command Codes For The LCD                                                                |
//-----------------------------------------------------------------------------------------+
#define SIZE_CODE_PWRA 6
#define SIZE_CODE_PWRB 4
#define SIZE_CODE_DTCA 4
#define SIZE_CODE_DTCB 3
#define SIZE_CODE_PSQC 5
#define SIZE_CODE_PMRC 2
#define SIZE_CODE_PCL1 2
#define SIZE_CODE_PCL2 2
#define SIZE_CODE_VCM1 3
#define SIZE_CODE_VCM2 2
#define SIZE_CODE_MACL 2
#define SIZE_CODE_PXFS 2
#define SIZE_CODE_FMCL 3
#define SIZE_CODE_DFCL 4
#define SIZE_CODE_3GFD 2
#define SIZE_CODE_GCSL 2
#define SIZE_CODE_SGM0 16
#define SIZE_CODE_SGM1 16
const byte LCD_CODE_PWRA[SIZE_CODE_PWRA] = {0xCB, 0x39, 0x2C, 0x00, 0x34, 0x02};
const byte LCD_CODE_PWRB[SIZE_CODE_PWRB] = {0xCF, 0x00, 0xC1, 0x30}; //Driver spec. has 0x81 (not 0xC1)
const byte LCD_CODE_DTCA[SIZE_CODE_DTCA] = {0xE8, 0x85, 0x00, 0x78};
const byte LCD_CODE_DTCB[SIZE_CODE_DTCB] = {0xEA, 0x00, 0x00};
const byte LCD_CODE_PSQC[SIZE_CODE_PSQC] = {0xED, 0x64, 0x03, 0x12, 0x81};
const byte LCD_CODE_PMRC[SIZE_CODE_PMRC] = {0xF7, 0x20};
const byte LCD_CODE_PCL1[SIZE_CODE_PCL1] = {0xC0, 0x23}; //VRH[5:0]
const byte LCD_CODE_PCL2[SIZE_CODE_PCL2] = {0xC1, 0x10}; //SAP[2:0]; BT[3:0]
const byte LCD_CODE_VCM1[SIZE_CODE_VCM1] = {0xC5, 0x3e, 0x28};
const byte LCD_CODE_VCM2[SIZE_CODE_VCM2] = {0xC7, 0x86};
const byte LCD_CODE_MACL[SIZE_CODE_MACL] = {0x36, 0x48};
const byte LCD_CODE_PXFS[SIZE_CODE_PXFS] = {0x3A, 0x55};
const byte LCD_CODE_FMCL[SIZE_CODE_FMCL] = {0xB1, 0x00, 0x18};
const byte LCD_CODE_DFCL[SIZE_CODE_DFCL] = {0xB6, 0x08, 0x82, 0x27};
const byte LCD_CODE_3GFD[SIZE_CODE_3GFD] = {0xF2, 0x00};
const byte LCD_CODE_GCSL[SIZE_CODE_GCSL] = {0x26, 0x01};

const byte LCD_CODE_SGM0[SIZE_CODE_SGM0] = {0xE0, 0x0F, 0x31, 0x2B, 0x0C, 0x0E, 0x08, 0x4E, 0xF1, 0x37, 0x07,
0x10, 0x03, 0x0E, 0x09, 0x00};
const byte LCD_CODE_SGM1[SIZE_CODE_SGM1] = {0xE1, 0x00, 0x0E, 0x14, 0x03, 0x11, 0x07, 0x31, 0xC1, 0x48, 0x08,
0x0F, 0x0C, 0x31, 0x36, 0x0F};

//-----------------------------------------------------------------------------------------+
// Initialization codes provided by class wiki, then start the screen all black           |
//-----------------------------------------------------------------------------------------+
void LCD_Init() {
    Region r = {
        0, LCD_COLS,
        0, LCD_ROWS,
        LCD_COLOR_BLACK
    };

    LCD_RST = 0;
    LCD_WaitChip(); // Give the LCD time to reset
    LCD_RST = 1;
    LCD_WaitChip(); // Give the LCD time to reset

    TP_CSX = 1;

    LCD_WriteCmd(LCD_CODE_PWRA[0]); // Power Control A
    LCD_WriteData(LCD_CODE_PWRA + 1, SIZE_CODE_PWRA - 1);

    LCD_WriteCmd(LCD_CODE_PWRB[0]); // Power Control B
    LCD_WriteData(LCD_CODE_PWRB + 1, SIZE_CODE_PWRB - 1);

    LCD_WriteCmd(LCD_CODE_DTCA[0]); // Driver timing control A
    LCD_WriteData(LCD_CODE_DTCA + 1, SIZE_CODE_DTCA - 1);

    LCD_WriteCmd(LCD_CODE_DTCB[0]); // Driver timing control B
    LCD_WriteData(LCD_CODE_DTCB + 1, SIZE_CODE_DTCB - 1);

    LCD_WriteCmd(LCD_CODE_PSQC[0]); // Power on sequence control
    LCD_WriteData(LCD_CODE_PSQC + 1, SIZE_CODE_PSQC - 1);

    LCD_WriteCmd(LCD_CODE_PMRC[0]); // Pump ratio control
    LCD_WriteData(LCD_CODE_PMRC + 1, SIZE_CODE_PMRC - 1);

    LCD_WriteCmd(LCD_CODE_PCL1[0]); // Power control 1
    LCD_WriteData(LCD_CODE_PCL1 + 1, SIZE_CODE_PCL1 - 1);

    LCD_WriteCmd(LCD_CODE_PCL2[0]); // Power control 2
    LCD_WriteData(LCD_CODE_PCL2 + 1, SIZE_CODE_PCL2 - 1);

    LCD_WriteCmd(LCD_CODE_VCM1[0]); // VCM control
    LCD_WriteData(LCD_CODE_VCM1 + 1, SIZE_CODE_VCM1 - 1);

    LCD_WriteCmd(LCD_CODE_VCM2[0]); // VCM control2
    LCD_WriteData(LCD_CODE_VCM2 + 1, SIZE_CODE_VCM2 - 1);

    LCD_WriteCmd(LCD_CODE_MACL[0]); // Memory Access Control
    LCD_WriteData(LCD_CODE_MACL + 1, SIZE_CODE_MACL - 1);

    LCD_WriteCmd(LCD_CODE_PXFS[0]); // Pixel format Set
    LCD_WriteData(LCD_CODE_PXFS + 1, SIZE_CODE_PXFS - 1);

    LCD_WriteCmd(LCD_CODE_FMCL[0]); // Frame Control
    LCD_WriteData(LCD_CODE_FMCL + 1, SIZE_CODE_FMCL - 1);

    LCD_WriteCmd(LCD_CODE_DFCL[0]); // Display Function Control
    LCD_WriteData(LCD_CODE_DFCL + 1, SIZE_CODE_DFCL - 1);

    LCD_WriteCmd(LCD_CODE_3GFD[0]); // 3Gamma Function Disable
    LCD_WriteData(LCD_CODE_3GFD + 1, SIZE_CODE_3GFD - 1);

    LCD_WriteCmd(LCD_CODE_GCSL[0]); // Gamma curve selected
    LCD_WriteData(LCD_CODE_GCSL + 1, SIZE_CODE_GCSL - 1);

    LCD_WriteCmd(LCD_CODE_SGM0[0]); // Set Gamma 0
    LCD_WriteData(LCD_CODE_SGM0 + 1, SIZE_CODE_SGM0 - 1);

    LCD_WriteCmd(LCD_CODE_SGM1[0]); // Set Gamma 1
    LCD_WriteData(LCD_CODE_SGM1 + 1, SIZE_CODE_SGM1 - 1);

    LCD_WriteCmd(0x11); // Exit Sleep
    LCD_WaitChip();
    LCD_WriteCmd(0x29); // Display on

    LCD_FillRegion(r);
}
```

Appendix H – Code: Ethernet Driver

enet.h:

```
#ifndef __WIZ550_H__
#define __WIZ550_H__
#include "../Shared/SPI.h"

//-------------------------------------------------------------------------------------+
// Hepler Structs For Configuration                                                    |
//-------------------------------------------------------------------------------------+
typedef enum {
    NET_CHIP_CLIENT,
    NET_CHIP_SERVER
} NET_CHIP;
typedef enum {
    NET_REG_COMMON,
    NET_REG_SOCKET,
    NET_REG_TX,
    NET_REG_RX
} NET_REG;
typedef enum {
    NET_MODE_VAR,
    NET_MODE_F1B,
    NET_MODE_F2B,
    NET_MODE_F4B
} NET_MODE;
typedef union {
    struct {
        uint
            socket:3, // Note: Must be 0 if using common register
            reg:2, // Default: NET_REG_COMMON
            write:1, // Default: Read (false)
            mode:2; // Default: NET_VAR_M
    };
    byte byte;
} ControlByte; // See section 2.2.2 in w550 datasheet

typedef struct {
    uint16 Address;
    ControlByte Control;
    byte* Data;
    uint N;
} NET_Frame;
typedef struct {
    uint16 Address;
    ControlByte Control;
    byte Data;
} NET_Byteframe;

typedef union {
    byte byte[4];
} IPv4;
typedef union {
    byte byte[6];
} MAC;

typedef enum {
    NET_INT_CON =       1 << 0,
    NET_INT_DISCON =    1 << 1,
    NET_INT_RECV =      1 << 2,
    NET_INT_TIMEOUT =   1 << 3,
    NET_INT_SENDOK =    1 << 4,
} NET_INT;

typedef struct {
    uint16 Size; // Data Packet Size
    byte* Data; // Pointer to "Data Packet" (same as Destination)
    byte* Destination; // size 6
    byte* Source; // size 6
    uint16 Type;

    uint16 PayloadSize;
    byte* Payload; // size "Size - 14"
} NET_MACRAW;

static const uint NET_BUFFER_SIZE = 2000;
extern byte NET_Buffer[NET_BUFFER_SIZE];
extern uint NET_Size;
```

```
static const uint NET_PACKET_BUFFER_SIZE = 50;
extern NET_MACRAW NET_Packet[NET_PACKET_BUFFER_SIZE];
extern uint NET_Packets;

//-------------------------------------------------------------------------------------+
// Driver Functions (specific to interpreting macraw data and blocking requests)       |
//-------------------------------------------------------------------------------------+
void NET_READDATA(NET_CHIP);
void NET_WRITEPACKET(NET_CHIP, uint);
void NET_SENDRESET(NET_CHIP);
bool NET_CHECKBLOCK(void);

//-------------------------------------------------------------------------------------+
// Driver Functions                                                                    |
//-------------------------------------------------------------------------------------+
byte NET_GetInterrupt(NET_CHIP);
void NET_ClearInterrupt(NET_CHIP, byte);
bool NET_SPI_BYTE(NET_CHIP, NET_Byteframe*);
bool NET_SPI(NET_CHIP, NET_Frame*);
void NET_Init(void);

//-------------------------------------------------------------------------------------+
// Communication signals for the wiznet chips                                          |
//-------------------------------------------------------------------------------------+
#define NET_RST             BAND_GPIO_PA6 // Attached to both chips
#define NET_RDY_SERVER      BAND_GPIO_PC4 // Attached router (server)
#define NET_RDY_CLIENT      BAND_GPIO_PC5 // Attached computer (client)
#define NET_CS_SERVER       BAND_GPIO_PE1 // Attached router (server)
#define NET_CS_CLIENT       BAND_GPIO_PE2 // Attached computer (client)

//-------------------------------------------------------------------------------------+
// Register Offset Addresses                                                           |
//-------------------------------------------------------------------------------------+
#define NET_COMMON_MODE                 (uint16)0x0000;
#define NET_COMMON_GATEWAY              (uint16)0x0001;
#define NET_COMMON_SUBNET               (uint16)0x0005;
#define NET_COMMON_MAC                  (uint16)0x0009;
#define NET_COMMON_IP                   (uint16)0x000F;
#define NET_COMMON_INTLEVEL             (uint16)0x0013;
#define NET_COMMON_IR                   (uint16)0x0015;
#define NET_COMMON_IMR                  (uint16)0x0016;
#define NET_COMMON_SIR                  (uint16)0x0017;
#define NET_COMMON_SIMR                 (uint16)0x0018;
#define NET_COMMON_VERSIONR             (uint16)0x0039;

#define NET_SOCKET_MR                   (uint16)0x0000;
#define NET_SOCKET_CR                   (uint16)0x0001;
#define NET_SOCKET_IR                   (uint16)0x0002;
#define NET_SOCKET_SR                   (uint16)0x0003;
#define NET_SOCKET_RXBUF_SIZE           (uint16)0x001E;
#define NET_SOCKET_TXBUF_SIZE           (uint16)0x001F;
#define NET_SOCKET_TX_FSR               (uint16)0x0020;
#define NET_SOCKET_TX_RD                (uint16)0x0022;
#define NET_SOCKET_TX_WR                (uint16)0x0024;
#define NET_SOCKET_RX_RSR               (uint16)0x0026;
#define NET_SOCKET_RX_RD                (uint16)0x0028;
#define NET_SOCKET_RX_WR                (uint16)0x002A;
#define NET_SOCKET_IMR                  (uint16)0x002C;

#endif
```

enet.c:

```
#include "../Shared/Controller.h"
#include "../Shared/GPIO.h"
#include "enet.h"
#include "filter.h"

//-------------------------------------------------------------------------------------+
// RX/TX Buffer for transfer between chips                                             |
//-------------------------------------------------------------------------------------+
byte NET_Buffer[NET_BUFFER_SIZE] = {0};
uint NET_Size = 0;

NET_MACRAW NET_Packet[NET_PACKET_BUFFER_SIZE] = {0};
uint NET_Packets = 0;

//-------------------------------------------------------------------------------------+
// Addressing information for configuration                                            |
//-------------------------------------------------------------------------------------+
typedef enum {
    ADDR_CLIENT,
    ADDR_SUBNET,
    ADDR_GATEWAY,
} addresses;
typedef enum {
    MAC_GHOST_CLIENT,
    MAC_GHOST_SERVER,
    MAC_CLIENT,
    MAC_SERVER,
} macs;

byte address[3][4] = {
    {129, 123, 5, 74}, // IP seen by client before disconnecting
    {255, 255, 254, 0}, // subnet mask; usu is weird
    {129, 123, 5, 254}, // default gateway seen by client before disconnecting
};
byte mac[4][6] = {
    {0x00, 0x90, 0xF5, 0xE9, 0xAA, 0x21}, // Obtained via ipconfig
    {0x00, 0x12, 0xF2, 0xC2, 0x4D, 0x00}, // Obtained via wireshark
    {0x00, 0x08, 0xDC, 0x1E, 0xB8, 0x73}, // On the sticker
    {0x00, 0x08, 0xDC, 0x1E, 0xB8, 0x7D}, // On the sticker
};

//---------------------------------------------------------------------------------------+
// Parse all the data into packet groups                                                 |
//---------------------------------------------------------------------------------------+
void NET_ParsePackets() {
    uint i = 0;
    NET_Packets = 0;
    while (i < NET_Size && NET_Packets < NET_PACKET_BUFFER_SIZE) {
        // Get the packet size
        NET_Packet[NET_Packets].Size = ((NET_Buffer[i] << 8) | NET_Buffer[i + 1]) - 2;
        i += 2;

        // Get the contents of the Data Packet
        NET_Packet[NET_Packets].Data = NET_Buffer + i;
        NET_Packet[NET_Packets].Destination = NET_Buffer + i;
        NET_Packet[NET_Packets].Source = NET_Buffer + i + 6;
        NET_Packet[NET_Packets].Type = (NET_Buffer[i + 12] << 8) | NET_Buffer[i + 13];
        NET_Packet[NET_Packets].Payload = NET_Buffer + i + 14;
        NET_Packet[NET_Packets].PayloadSize = NET_Packet[NET_Packets].Size - 14;
        i += NET_Packet[NET_Packets].Size;

        // Move onto the next packet
        if (i <= NET_Size) { ++NET_Packets; }
    }
}

//---------------------------------------------------------------------------------------+
// Read all available data from the given chip                                           |
//---------------------------------------------------------------------------------------+
void NET_READDATA(NET_CHIP chip) {
    NET_Frame frame;
    byte data[2];
    uint16 size = 0;

    // Constant through the function
    frame.Control.socket = 0;
    frame.Control.mode = NET_MODE_VAR;

    // Determine the amount of data to read
    frame.Control.reg = NET_REG_SOCKET;
    frame.Control.write = false;
    frame.Address = NET_SOCKET_RX_RSR;
    frame.Data = data;
    frame.N = 2;
    do{
        NET_Size = size;
        NET_SPI(chip, &frame);
        size = (data[0] << 8) | data[1];
    } while (NET_Size != size);

    // find the RX read pointer
    frame.Address = NET_SOCKET_RX_RD;
    NET_SPI(chip, &frame);

    // read the data on the buffer
    frame.Control.reg = NET_REG_RX;
    frame.Address = (data[0] << 8) | data[1];
    frame.Data = NET_Buffer;
    frame.N = NET_Size;
    NET_SPI(chip, &frame);

    // Update the RX read pointer
    data[0] = ((frame.Address + NET_Size) & 0xFF00) >> 8;
    data[1] = ((frame.Address + NET_Size) & 0x00FF);

    frame.Control.write = true;
    frame.Control.reg = NET_REG_SOCKET;
    frame.Address = NET_SOCKET_RX_RD;
    frame.Data = data;
    frame.N = 2;
    NET_SPI(chip, &frame);

    // Give RECV command to the CR
    frame.Address = NET_SOCKET_CR;
    data[0] = 0x40;
    frame.N = 1;
    NET_SPI(chip, &frame);

    NET_ParsePackets();
}

//------------------------------------------------------------------------------------------+
// In MacRaw mode, packets must be sent one at a time                                       |
//------------------------------------------------------------------------------------------+
void NET_WRITEPACKET(NET_CHIP chip, uint packet) {
    NET_Frame frame;
    byte data[2];

    // Initial frame setup
    frame.Control.socket = 0;
    frame.Control.mode = NET_MODE_VAR;
    frame.Data = data;
    frame.N = 2;

    // Find the TX write pointer
    frame.Control.write = false;
    frame.Control.reg = NET_REG_SOCKET;
    frame.Address = NET_SOCKET_TX_WR;
    NET_SPI(chip, &frame);

    // Write the data to the buffer
    frame.Control.write = true;
    frame.Control.reg = NET_REG_TX;
    frame.Address = (data[0] << 8) | data[1];
    frame.N = NET_Packet[packet].Size;
    frame.Data = NET_Packet[packet].Destination;
    NET_SPI(chip, &frame);

    // Update the TX write pointer
    data[0] = ((frame.Address + frame.N) & 0xFF00) >> 8;
    data[1] = ((frame.Address + frame.N) & 0x00FF);

    frame.Address = NET_SOCKET_TX_WR;
    frame.Control.reg = NET_REG_SOCKET;
    frame.Control.write = true;
    frame.Data = data;
    frame.N = 2;
    NET_SPI(chip, &frame);

    // Give SEND command to the CR
    frame.N = 1;
    data[0] = 0x20;
    frame.Address = NET_SOCKET_CR;
    NET_SPI(chip, &frame);

    // Poll to confirm the command was processed
    frame.Control.write = false;
    frame.Address = NET_SOCKET_IR;
    do { NET_SPI(chip, &frame);
    } while(!(data[0] & NET_INT_SENDOK));

    // Acknowledge (only) the SendOk interrupt
    frame.Control.write = true;
    data[0] = NET_INT_SENDOK;
    NET_SPI(chip, &frame);
}

//------------------------------------------------------------------------------------------+
// Interrupt Handlers                                                                       |
//------------------------------------------------------------------------------------------+
byte NET_GetInterrupt(NET_CHIP chip) {
    NET_Frame frame;
    byte Int;

    frame.Control.mode = NET_MODE_VAR;
    frame.Control.reg = NET_REG_SOCKET;
    frame.Control.write = false;
    frame.Control.socket = 0;

    frame.Address = NET_SOCKET_IR;
    frame.Data = &Int;
    frame.N = 1;

    NET_SPI(chip, &frame);
    return Int;
}
void NET_ClearInterrupt(NET_CHIP chip, byte Int) {
    NET_Frame frame;
    frame.Control.mode = NET_MODE_VAR;
    frame.Control.reg = NET_REG_SOCKET;
    frame.Control.write = true;
    frame.Control.socket = 0;

    frame.Address = NET_SOCKET_IR;
    frame.Data = &Int;
    frame.N = 1;

    NET_SPI(chip, &frame);
}

//-------------------------------------------------------------------------------------+
// The ControlByte object is not packing properly, so this is a fix for byte packing     |
//-------------------------------------------------------------------------------------+
byte NET_ControlFix(ControlByte cb) {
    byte value = 0;
    value |= cb.socket << 5;
    value |= cb.reg << 3;
    value |= cb.write << 2;
    value |= cb.mode;
    return value;
}

//-------------------------------------------------------------------------------------+
// Alias the more generic NET_SPI to handle case of single byte transmission             |
//-------------------------------------------------------------------------------------+
bool NET_SPI_BYTE(NET_CHIP chip, NET_Byteframe* frame) {
    NET_Frame rogue;
    rogue.N = 1;
    rogue.Address = frame->Address;
    rogue.Control = frame->Control;
    rogue.Data = &frame->Data;
    return NET_SPI(chip, &rogue);
}

//-------------------------------------------------------------------------------------+
// Converts address into bytestream, does input validation, and calls spi handler        |
//-------------------------------------------------------------------------------------+
bool NET_SPI(NET_CHIP chip, NET_Frame* frame) {
    SPI_Frame spi;
    byte mosi[3] = {0},
        miso[3] = {0};

    // Safety precautions
    if (frame->Control.mode == NET_MODE_F1B) {
        if (frame->N > 1) { frame->N = 1; }
        else if (frame->N < 1) { return false; }
    }
    if (frame->Control.mode == NET_MODE_F2B) {
        if (frame->N > 2) { frame->N = 2; }
        else if (frame->N < 2) { return false; }
    }
    if (frame->Control.mode == NET_MODE_F4B) {
        if (frame->N > 4) { frame->N = 4; }
        else if (frame->N < 4) { return false; }
    }
    if (frame->Control.reg == NET_REG_COMMON) {
        frame->Control.socket = 0;
    }

    // Type conversion
    mosi[0] = (frame->Address & 0xFF00) >> 8;
    mosi[1] = (frame->Address & 0x00FF);
    mosi[2] = NET_ControlFix(frame->Control);

    // Setup the spi frame
    spi.bus.N = 3;
    spi.bus.MOSI = mosi;
    spi.bus.MISO = miso;
    spi.route.SSI = SSI0;
    spi.route.CS = (chip == NET_CHIP_CLIENT)
        ? &NET_CS_CLIENT
        : &NET_CS_SERVER;

    // Fancy SPI_Transfer
    SPI_Begin(&spi.route);

    // Send the Address/Control Block
    SPI_Block(&spi.route, &spi.bus);

    // Send the data block
    spi.bus.N = frame->N;
    if (frame->Control.write) {
        spi.bus.MISO = 0;
        spi.bus.MOSI = frame->Data;
    } else {
        spi.bus.MISO = frame->Data;
        spi.bus.MOSI = 0;
    }
    SPI_Block(&spi.route, &spi.bus);

    // End Fancy SPI_Transfer
    SPI_End(&spi.route);

    return true;
}

//-------------------------------------------------------------------------------------------+
// Initialize ip addressing to a ghost configuration                                         |
//-------------------------------------------------------------------------------------------+
void NET_Init() {
    uint i;
    NET_Frame frame;
    NET_Byteframe byteframe;

    // Reset the chips
    // doc: http://wizwiki.net/wiki/doku.php?id=products:wiz550io:allpages#reset_timing
    // Busy wait to keep reset pin low for longer than 400us
    NET_RST = 0;
    for (i = 0; i < 5000; ++i);
    NET_RST = 1;
    while (!NET_RDY_CLIENT); // Wait for ready signal from CPC
    while (!NET_RDY_SERVER); // Wait for ready signal from ISP

    // Initialize the frame for all configuration needs
    frame.Control.write = true;
    frame.Control.reg = NET_REG_COMMON;
    frame.Control.mode = NET_MODE_VAR;
    frame.Control.socket = 0;

    // Initialize the byteframe for all configuration needs
    byteframe.Control.write = true;
    byteframe.Control.reg = NET_REG_COMMON;
    byteframe.Control.mode = NET_MODE_VAR;
    byteframe.Control.socket = 0;

    //-------------------------------------------------------------------------------------------+
    // Chip Addressing Configuration                                                             |
    //-------------------------------------------------------------------------------------------+
    // MAC (physical address)
    frame.N = 6;
    frame.Data = mac[MAC_GHOST_CLIENT];
    NET_SPI(NET_CHIP_SERVER, &frame);

    frame.Data = mac[MAC_GHOST_SERVER];
    NET_SPI(NET_CHIP_CLIENT, &frame);
    frame.N = 4; // Following transmissions are 4-bytes

    // Subnet Mask
    frame.Address = NET_COMMON_SUBNET;
    frame.Data = address[ADDR_SUBNET];
    NET_SPI(NET_CHIP_CLIENT, &frame);
    NET_SPI(NET_CHIP_SERVER, &frame);
    // IP Address
    frame.Address = NET_COMMON_IP;
    frame.Data = address[ADDR_CLIENT];
    NET_SPI(NET_CHIP_SERVER, &frame);

    // Look like the default gateway to the client chip
    frame.Data = address[ADDR_GATEWAY];
    NET_SPI(NET_CHIP_CLIENT, &frame);

    // Default Gateway
    frame.Address = NET_COMMON_GATEWAY;
    frame.Data = address[ADDR_GATEWAY];
    NET_SPI(NET_CHIP_SERVER, &frame);

    //--------------------------------------------------------------------------------------+
    // Common Chip Configuration                                                            |
    //--------------------------------------------------------------------------------------+
    // Interrupts
    byteframe.Address = NET_COMMON_IMR;
    byteframe.Data = 0xC0; // enable interrupts for ip conflict, and dest unreachable
    NET_SPI_BYTE(NET_CHIP_CLIENT, &byteframe);
    NET_SPI_BYTE(NET_CHIP_SERVER, &byteframe);

    byteframe.Address = NET_COMMON_SIMR;
    byteframe.Data = 0xFF; // enable interrupts from all sockets
    NET_SPI_BYTE(NET_CHIP_CLIENT, &byteframe);
    NET_SPI_BYTE(NET_CHIP_SERVER, &byteframe);

    byteframe.Control.reg = NET_REG_SOCKET;
    byteframe.Address = NET_SOCKET_IMR; // Configure socket 0
    byteframe.Data = 0xF; // Enables most interrupts
    NET_SPI_BYTE(NET_CHIP_CLIENT, &byteframe);
    NET_SPI_BYTE(NET_CHIP_SERVER, &byteframe);

    // MacRaw Mode
    byteframe.Address = NET_SOCKET_MR;
    byteframe.Data = 0x04;
    NET_SPI_BYTE(NET_CHIP_CLIENT, &byteframe);
    NET_SPI_BYTE(NET_CHIP_SERVER, &byteframe);

    byteframe.Address = NET_SOCKET_CR;
    byteframe.Data = 0x1; // Open Socket 0
    NET_SPI_BYTE(NET_CHIP_CLIENT, &byteframe);
    NET_SPI_BYTE(NET_CHIP_SERVER, &byteframe);

    // Wait for the sockets to finish opening
    byteframe.Address = NET_SOCKET_SR;
    byteframe.Control.write = false;

    do { NET_SPI_BYTE(NET_CHIP_CLIENT, &byteframe);
    } while (byteframe.Data != 0x42);

    do { NET_SPI_BYTE(NET_CHIP_SERVER, &byteframe);
    } while (byteframe.Data != 0x42);
}
```