# A DIY Hardware Packet Sniffer

Lead: It is not uncommon for system support staff to secure a system from external attack, but to overlook an internal threat which can be exploited due to vulnerabilities in physical security. In order to test such for such a vulnerability, we have developed a Hardware Packet Sniffer (HPS) device, which sniffs network traffic and retransmits it to another machine. By making the device small, it can easily be overlooked by support staff and other employees. By making the device low-cost, it is effectively disposable, requiring no retrieval.

### What you will learn

The article presents an in depth explanation about the structure and function of the ENC28J60 Ethernet controller chip from Microchip, and how it can be used together with an 8-bit microcontroller to build a low-cost packet sniffer. You will learn about the inner workings of SPI (Serial Peripheral Interface) and how it is used to communicate between different hardware components.

### What you should know

It is assumed that the reader has a basic knowledge of the way network traffic operates. The reader should know what a MAC address and IP address are and have an understanding of C programming. It is also assumed that if the reader wishes to recreate this project, he or she has basic circuit fabrication knowledge and skills, as this is not intended to be a fabrication tutorial.

### About the Authors

Ian G. Harris is currently an Associate Professor in the Computer Science Department at the University of California Irvine. He received his BS degree in Computer Science from Massachusetts Institute of Technology in 1990. He received his MS and PhD degrees in Computer Science from the University of California San Diego in 1992 and 1997 respectively. He was a member of the faculty in the Electrical and Computer Engineering Department at the University of Massachusetts Amherst from 1997 until June 2003. His field of interest includes validation of hardware systems to ensure that the behavior of the system matches the intentions of the designer. He also investigates the application of testing for computer security. His group's security work includes testing software applications for security vulnerabilities and designing special-purpose hardware to detect intrusions on-line.

Veronica Swanson is an Undergraduate Student at the University of California at Irvine going into her junior year studying Mechanical Engineering. She is interested in the marriage of hardware and software systems and hopes to use her knowledge of mechanical systems and programming to work in the field of robotics. Already she has been involved in several projects requiring a functioning knowledge of embedded systems and mechanical and electronic fabrication skills.

Randy Harper is a third year Computer Science and Engineering Undergraduate student at the University of California Irvine. His interests include software development for computer networking, computer vision, and embedded computer systems. He has participated in and developed software for computer vision and embedded systems projects.

## Purpose of the HPS Device

This project began with the purpose of creating a small and inexpensive, do it yourself packet sniffer for penetration testing. In terms of small penetration testing technologies, there are plenty of powerful tools available on the market. Global Scale Technologies sells the Sheeva Plug [1], essentially a mini computer that plugs into your wall that can run network-based software. A number of Linux distributions have even been created for it. Pwnie Express offers a number of devices designed specifically for penetration testing including the Pwn Plug Mini, Pwn Plug Wireless, Pwn Plug Elite, and the Power Pwn [2]. However these can be rather expensive. At the time of writing this article, Global Scale sold the US model of their SheevaPlug Development Kit for $99 (USD) and Pwnie Express sold their cheapest testing platform for $395 (USD). Our development costs came out to about $43.99. Other devices may be more powerful, but our project, the Hardware Packet Sniffer, or HPS, is simple, small, and efficient. It measures at only 6cm x 4cm x 4cm and was designed to steal packets from a network, and that is all it does. Buying an off-the-shelf product would mean that a team of developers has already taken care of all the debugging and trouble shooting, but part of the appeal of DIY projects is working through problems and customizing capabilities and performance yourself.

### What does it do?

# The HPS was designed to be a small, concealable packet sniffer. Simply plug the HPS into an open Ethernet port and it will receive packets of data being sent on the network and then transmit them to your computer.

## Hardware

# The HPS is composed of two main pieces of hardware: a mikroETH Board using an ENC28J60 Ethernet controller from Microchip, and an Atmega328p microcontroller from Atmel Corporation. The Atmega328p operates as the host controller for the Ethernet controller and operates and programs the Ethernet controller via SPI (Serial Peripheral Interface). The Ethernet controller then interfaces with the system network using an Ethernet cable and reads packets from network and sends them to an offsite device with known MAC and IP addresses.

### Ethernet Controller

# The Ethernet controller’s function, a device in a network will only receive network traffic intended for that device, but by setting the Ethernet controller in promiscuous mode, the HPS will capture and store packets of data from the network that it is plugged into regardless of what MAC address the packets were intended for. One of the nice features of the Ethernet controller is that not only can it receive data over Ethernet, but it can also send data. By hard coding a MAC address into the Ethernet controller, the HPS can then send the data it collects to that specified device.

#### Ethernet Memory Organization

The communications between the microcontroller and the Ethernet controller can be classified as one of three types.

1. Configuration of the Ethernet controller.
2. Issuing commands to the Ethernet controller (i.e. transmit, receive).
3. Accessing the message data, either the data to transmit or the data received.

Each type of communication is performed by accessing dedicated regions of memory in the Ethernet controller. The memory in the Ethernet controller is all static RAM, and it is organized into three sections: *control registers*, *Ethernet buffers*, and *physical registers*.

#### Physical Registers

# The PHY (Physical) registers are used for configuration and control of the physical device, and are unique in that they cannot be accessed directly, but must be accessed through a special set of MAC (Medium Access Control) registers. Unlike the control registers, the physical registers are sixteen bits long and must be written all at once. The lower eight bits are written first to the MIWRL (Media Independent Interface Write Data Low) register, and then the upper eight bits are written to the MIWRH (Media Independent Interface Write Data High) register. The factory configuration is mostly adequate, but for this project we needed to access the physical registers once to configure the Ethernet controller for Full Duplex mode. Full duplex mode allows the Ethernet controller to simultaneously send and receive data, whereas in Half Duplex mode, which is the default setting, traffic can only flow one way. This slows down the send and receive process and can cause collisions as data is received while the Ethernet controller is in the process of transmitting data. To configure the controller for full-duplex, bit eight of PHYCON1 (Physical Control Register 1), labeled PDPXMD, must be set to one and bit zero of MACON3 (MAC Control Register 3), FULDPX, must also be set to one.

#### Control Registers

# The control registers provide access to the on-chip Ethernet controller logic through the SPI interface. Writing to the individual registers allows the host controller, the Atmega328p microcontroller, to operate the Ethernet controller, and reading from the registers facilitates monitoring the Ethernet controller’s operation. The control register memory is sectioned into four banks, each containing different registers. The last five addresses in each bank, however, access a common set of registers, including ECON1 (Ethernet Control Register 1) which contains the bit used for switching between banks so that the user can easily switch between banks to access any register. There are three types of control registers: ETH (Ethernet), MAC, and MII, both of which have been previously described. The group that a register belongs to can easily be determined by looking at the first few letters of the register name: ETH register names start with ‘E,’ MAC with ‘MA,’ and MII with ‘MI.’ The ETH registers contain most of the logic used in receiving and transmitting packets. The MAC registers contain most of the configuration settings for the Ethernet controller operation and require initialization upon start up. The ENC28J60 data sheet contains a ten step initialization procedure for the MAC registers. The MII registers are used primarily to access the PHY registers.

#### Ethernet Buffer

# The Ethernet buffer is an eight Kbyte memory split into a receive buffer and a transmit buffer where the data received and the data to be transmitted are stored. The size and location of the receive buffer is determined by the host controller by programming the ERXST, Ethernet Receive Buffer Start, and ERXND, Ethernet Receive Buffer End, pointers to define the start and end of the receive buffer. All memory not allocated to the receive buffer is then used as the transmit buffer. The ERXWRPT, Ethernet Receive Buffer Write Pointer, contains the location within the receive buffer where data received through the Ethernet interface will be written. The pointer is read-only and automatically updates as data is written. Upon start up, the ERXWRPT is written to match the ERXST pointer so that data starts writing at the beginning of the receive buffer. Data is read using the Read Buffer Memory SPI command and the ERDPT, Ethernet Receive Buffer Pointer. The ERDPT specifies the location within the memory that is to be read, and if the AUTOINC bit in the ECON2 register is set, ERDPT will automatically increment to the next address after the last bit of the previous byte has been read. The receive buffer is a circular first in first out buffer. When reading from the buffer, if ERDPT equals ERXND, the hardware will automatically update the read pointer to equal the receive buffer start pointer to facilitate continuous reading from the end of the buffer to the start. However, this wrapping condition does not apply to writing to the buffer, and when new data is received, the old data is not overwritten. The new data is instead discarded, and the buffer memory must be cleared of old data before new data can be written. To transmit the received data, the host controller must write the data and the appropriate network headers into the transmit buffer using the Write Buffer Memory SPI command and the EWRPT pointer, which designates the address to be written to. The ETXST, Ethernet Transmit Buffer Start, and ETXND, Ethernet Transmit Buffer End, pointers are programmed with the beginning and ending addresses of the packet, and are then used to determine the location of the packet to be sent.

### Atmega328p

The second component of note is an Atmega328p microcontroller from Atmel Corporation. The microcontroller is the host controller used to program and operate the Ethernet controller over SPI, Serial Peripheral Interface. Before the Atmega328p can be used as the host controller, the device needs to be configured for Master SPI mode. This is simply done by setting the directions of the SPI pins appropriately as shown in Table 1.

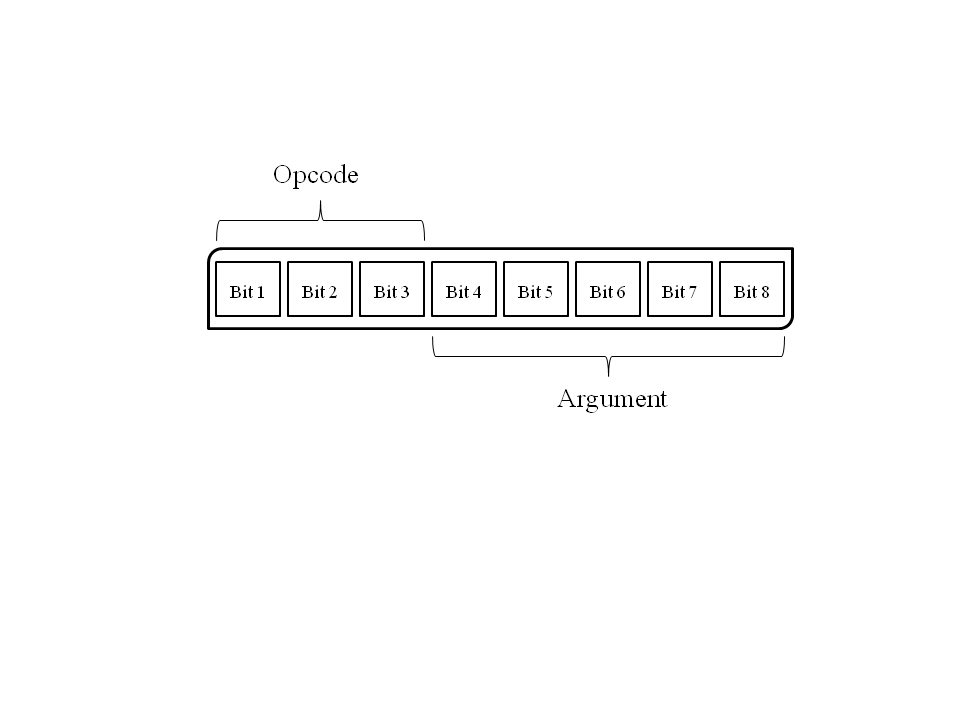
*Table 1. List of Atmega328p pins used in SPI [3]*

|  |  |  |
| --- | --- | --- |
| Pin Name | Pin Number | SPI Master Direction |
| MOSI (Master Out Slave In) | PB3 | Output |
| MISO (Master In Slave Out) | PB4 | Input |
| SCK (Serial Clock) | PB5 | Output |
| SS (Slave Select) | PB2 | Output |

Three bits in the SPI control register also need to be set: bit six (SPE), bit four (MSTR), and bit zero (SPR0). SPE, SPI Enable, enables SPI operation when set to one. MSTR, Master/Slave Select, configures the device for Master SPI mode when set to one, and Slave SPI mode when set to zero. SPR0, SPI Clock Rate Select 0, controls the clock rate when the device is configured as Master. Setting SPR0 to one, in combination with bits SPI2X and SPR1 which do not need to be altered, sets the oscillator frequency 1/16.

#### SPI Interface

The Ethernet controller is designed to respond to a set of seven SPI instructions. Each instruction is one byte long followed by one byte of data if appropriate for the instruction.



*Figure 1. 8 bit SPI Ethernet controller instruction*

As shown in Figure 1, the first three bits contain the opcode defining which instruction you would like to use, followed the five bit argument which, depending upon the instruction being used, is either a constant or the register address that you are trying to access. The seven instructions are Read Control Register, Read Buffer Memory, Write Control Register, Write Buffer Memory, Bit Field Set, Bit Field Clear, and System Command which initiates a soft reset of the chip. Table 2 contains the opcodes and arguments of each instruction at the bit level, and can be found in the ENC28J60 datasheet on page twenty-six [4].

*Table 2. Bit-level view of SPI Ethernet controller instructions*

|  |  |  |  |
| --- | --- | --- | --- |
| SPI Ethernet controller Instruction | Opcode | Argument | Data |
| Read Control Register | 0 0 0 | a a a a a | n/a |
| Read Buffer Memory | 0 0 1 | 1 1 0 1 0 | n/a |
| Write Control Register | 0 1 0 | a a a a a | d d d d d d d d |
| Write Buffer Memory | 0 1 1 | 1 1 0 1 0 | d d d d d d d d |
| Bit Field Set | 1 0 0 | a a a a a | d d d d d d d d |
| Bit Field Clear | 1 0 1 | a a a a a | d d d d d d d d |
| System Command (Soft Reset) | 1 1 1 | 1 1 1 1 1 | n/a |

## Software Block Diagram.1.pngThe Software

The programming for this project was done in C using Atmel Studio 6 [5] and loaded on to the Atmega328p using an AVR JTAGICE mkII from Atmel Corporation. Comparable C programming environments and in-system programmers may be used in place of Atmel Studio 6 and the mkII. Detailed instructions on programming with the mkII can be found on the Atmel website under documents [6]. The software is composed of the three layers shown in Figure 2. The lowest layer contains the code initializing the Atmega328p to operate in master mode for SPI and the SPI Ethernet controller instructions. The middle layer contains the code for properly configuring the Ethernet controller, as well as the receive and transmit functions. The top layer, the sniffing application, appropriately executes the receive and transmit functions.

Figure 2. Diagram of the HPS software layers

### SPI Functions

The bottom layer of the software contains the code to configure the Atmega328p for Master SPI mode and the SPI Ethernet controller instructions. The following code is the masterInit() function used to configure the Atmega328p.

<<LISTING 1>>

***Listing 1. ATMEGA 328p initialization code***

void masterInit(void){

// Set MOSI, SCK, and SS output, all others input

1. DDR\_SPI = (1<<DD\_MOSI) | (1<<DD\_SCK) | (1<<DD\_SS);

// Enable SPI, Master, set clock rate fck/16

2. SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);

// Set SS to HIGH

3. PORTB |= (1<<DD\_SS);

}

<</LISTING 1>>

In line one, the direction of the SPI pins are set for SPI Master mode, and bits SPE, MSTR, and SPR0 in the SPI control register are set to one in line two. The SS, Slave Select, pin is then driven high in line three. In SPI, the Master pulls SS low to initiate a data transfer cycle, so at all other times SS must be high. The next function, writeControl(), executes the Write Control Register SPI Ethernet controller instruction.

<<LISTING 2>>

***Listing 2. Function to write to the control registers of the enc28j60***

void writeControl(unsigned char a, unsigned char d){

// Set SS to LOW

1. PORTB &= ~(1<<DD\_SS);

// Concatenate Opcode and Address

2. a = WRITE\_CONTROL\_OP | a;

// Send Address

3. SPDR = a;

// Wait until the transmission completes

4. while(!(SPSR & (1<<SPIF)));

// Send Data

5. SPDR = d;

// Wait until the transmission completes

6. while(!(SPSR & (1<<SPIF)));

// Set SS to HIGH

7. PORTB |= (1<<DD\_SS);

}

<</LISTING 2>>

The Write Control Register instruction is two bytes long, one bit to send the opcode and address, one bit to send the data to be written. Only one byte is sent per transmission cycle, so the writeControl() function initiates two cycles. To initiate the transmission, SS is pulled low. Then the opcode and address are concatenated in line 2 and loaded into SPDR in line three. Writing data to SPDR, the SPI Data Register, begins the transmission and starts the SPI clock generator. One bit is shifted per clock cycle, and after all 8 bits have been transferred to the slave, the clock generator stops and the end of transmission flag (SPIF) in the SPI status register is set. Line 4 runs a while loop that continuously checks to see if the flag has been set, and once it has been set the transmission is complete. Lines five and six repeat the transmission process, sending the data to be written to the specified register. Once both transmission cycles are complete, SS is driven high to end the transmission sequence.

<<LISTING 3>>

***Listing 3. Function to read the control registers of the enc28j60***

unsigned char readControl(unsigned char a){

// Set SS to LOW

1. PORTB &= ~(1<<DD\_SS);

// Load Address Byte to SPDR

2. SPDR = READ\_CONTROL\_OP | a;

// Wait until the transmission completes

3. while(!(SPSR & (1<<SPIF)));

// Reload Address Byte to SPDR to get SCK to cycle

4. SPDR = a;

// Wait until the transmission completes

5. while(!(SPSR & (1<<SPIF)));

// Temporarily store SPDR into a

6. a = SPDR;

// Set SS to HIGH

7. PORTB |= (1<<DD\_SS);

// Return the received byte

8. return a;

}

<</LISTING 3>>

The readControl() function, which executes the Read Control Register instruction, is very similar to the write function, but it initiates and extra transmission containing a dummy byte. In SPI, data is shifted to and from the Slave simultaneously, however the data requested by a read command can be sent only after all a bits of the command have been sent at the start of the next transmission cycle. So the readControl() function has to initiate a second transmission cycle to obtain the data requested by the first cycle. Once the cycle is complete, the data received is contained in SPDR, which is then stored in the variable ‘a’ in line 6, and returned in line 8 once SS has been driven high to end the transmission sequence.

### Ethernet Firmware

The Ethernet firmware contains three main sections: the configuration which properly initializes the Ethernet controller’s operating procedures, and the receive and transmit functions.

#### Configuration

The Ethernet controller has several registers that need to be set in order to initialize the controller. Two functions, macInit() and bufferInit(), are used to initialize the controller. Using the Write Control Register and Bit Field Set instructions, macInit() configures the MAC registers as recommended by the Ethernet controller datasheet, sets the maximum packet size that can be received or transmitted, configures the controller to operate in full-duplex, sets the source MAC address, and sets the filter mode. bufferInit() configures the Ethernet buffer memory. The functions sets the AUTOINC bit in ECON2, and defines the size of the receive buffer by assigning addresses to the receive buffer start and end pointers, ERXST and ERXND. Then the receive buffer read pointer is assigned the ERXST address to begin reading at the start of the receive buffer, and the transmit buffer pointer is assigned the ERXST address to begin reading at the start of the transmit buffer which begins at the end of the receive buffer.

#### Reception

The actual reception of packets is completed automatically by the Ethernet hardware once the controller is configured. What the program needs to do is read the packet from the receive buffer so the host controller can then write the packet data into the transmission buffer when the transmit operation takes place. The readPacket() function is used to read the packet out of the buffer and then clear the packet from memory to free up buffer memory space. All received packets have the following layout. The data is preceded by a six byte header. The first two bytes of the header contain the next packet pointer with the location of the beginning of the next packet. The other four bytes of the header are a receive status vector which contains information about the packet and messages about any errors that may have occurred during receiving. After the data, a four byte CRC is appended. In order to process the actual data, readPacket() strips the received packet of the header and CRC, saving the length, next packet pointer, and status vector, and then copies the data into an array called packet. The function then clears the buffer memory, decrements the packet pointer, and returns the length or the received packet.

#### Transmission

transmitPacket(l) is the function that handles data transmission of a packet of length l, and there are five basic steps the function must complete to transmit a packet. First, the transmit buffer start pointer, ETXST, must be programmed to an unused location that will then be used as the start of the packet. Using the Write Buffer Memory instruction, the packet is written to the transmit buffer. The packet starts with a mandatory per packet control byte, followed by the necessary Ethernet, IPv4, and UDP headers, after which the data is finally written. Then the transmit buffer end pointer, ETXND, must be programmed to point to the last byte of the data. Then the desired interrupts for transmission need be set. Finally, the TXRST, Transmit Logic Reset, bit in ECON1 is set and cleared to initiate transmission. Then the TXERIF, Transmit Error Interrupt Flag, bit in EIR, Ethernet Interrupt Request Register, is cleared to denote that no error occurred during transmission and the TXRTS, Transmit Request To Send, bit in ECON1 is set, meaning a packet transmission is in progress.

### Sniffing Application

The sniffing application is contained in an infinite while loop in the main function.

1. while(1)

2. if(packetReceived())

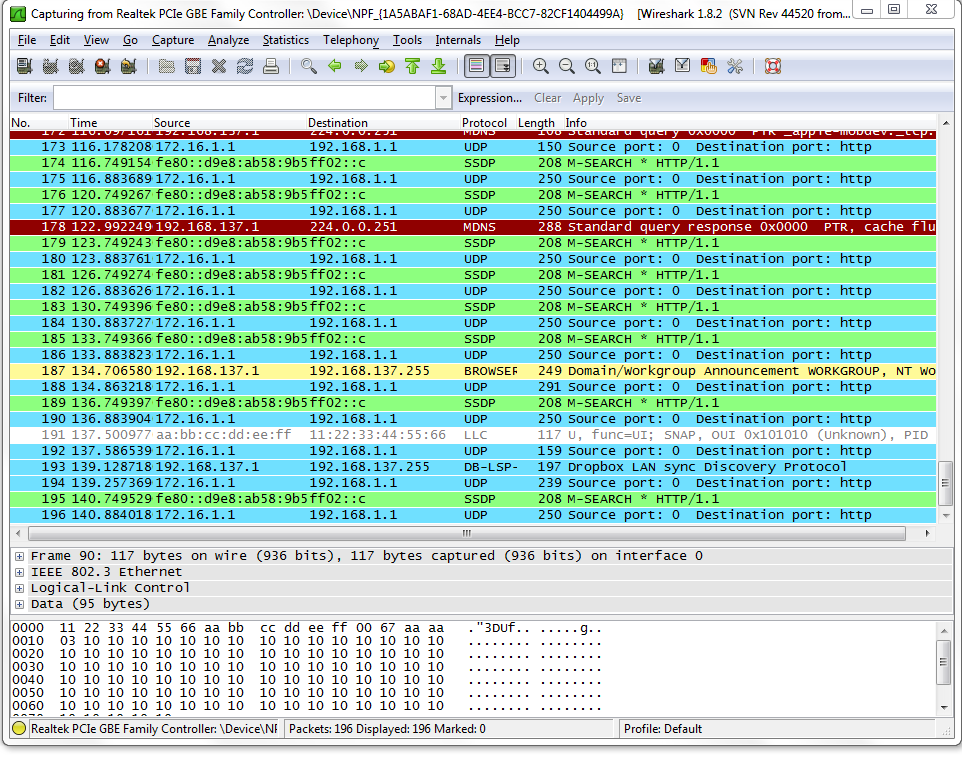
3. if (transmitPacket(readPacket()))

4. while (transmitPending());

The function continually checks the value of function packetReceived(). packetReceived() of the PKTIF (Receive Packet Pending Interrupt Flag) bit in EIR. If a packet has been received, the flag will be set to one and the function will return a value of one. If packetReceived() equals one, the function proceeds to line 3 which transmits the packet and returns a value of one once the function has been run. Line 4 then runs a while loop checking the value of the function transmitPending(), which checks the status of the TXIF (Transmit Interrupt Flag) bit in EIR. TXIF is set once the transmit request has ended. Once TXIF is set, the program exits the while loop and repeats the sniffing process.

### Handling the Acquired Data

Once the HPS is in place and switched on, it will begin transmitting data to the designated destination MAC and IP address. To view and capture the data, a network protocol analyzer such as Wireshark, can be used. Implementing capture filters for source IP or MAC address can isolate the target packets.



*Figure 3. Wireshark running a capture for received HPS packets*

Figure 3 shows Wireshark receiving packets from the HPS. The HPS packets are coming from the source IP address 172.16.1.1 to the destination address 192.168.1.1 and the protocol they are running is UDP. Running a capture to view all in coming network traffic picks up many irrelevant packets, as can be seen in Figure 3. So it is suggested that the user experiment with capture filters to avoid unnecessary packets. Once the packets have been captured in Wireshark, the data payload can be viewed at the bit level by selecting the desired packet.

## Building the Board

The physical device is fairly simple to make. The components include the Ethernet chip, the Atmega328p, an Ethernet cable, a 3.3V power source, a small section of perfboard, and the wires needed to connect it all together. Assuming you have a soldering iron, some available solder, a bit of spare electrical wire, and the tools necessary to shorten an Ethernet cable your cost should come close to around $43.99. The Ethernet controller board operates at 3.3V, so the entire system needs to be run at 3.3V (which is an acceptable level for the Atmega328p) or use some combination of regulators and level translators. For this project, the entire circuit was run off a 3.3V power supply, which was achieved with the Bodhilabs AA battery holder that has a 3.3V regulator built in. This could also be achieved by using any other stand alone power source and a corresponding step up or step down regulator to achieve the desired 3.3V. Table 3 contains a list of the parts used in this project. There is no need to use exactly what is listed here. Parts other than the Ethernet controller and the microcontroller can be substituted with what you have on hand or can find in stores. The Ethernet controller used has been discontinued, but MikroElektronica now produces a substitute chip, the ETH Click [7] which may be used instead. In fact, most any Ethernet development board could be integrated into this project as long as it uses the ENC28J60 microchip.

*Table 3. List of parts used in the HPS and their price and a URL to site of purchase*

|  |  |  |  |
| --- | --- | --- | --- |
| # | Part | Price | URL |
| 1 | mikroETH Board | $24.00 | http://www.mikroe.com/add-on-boards/communication/mikroeth/ |
| 2 | AA 1.5V Battery | $0.40 | http://www.mouser.com/Search/ProductDetail.aspx?R=LR6XWAvirtualkey65800000virtualkey658-LR6XWA |
| 3 | Bodhilabs AA Battery Holder with 3.3V Regulator | $10.95 | http://www.pololu.com/catalog/product/796 |
| 4 | ATMEGA328P-PU Microcontroller | $2.24 | http://www.mouser.com/ProductDetail/Atmel/ATMEGA328P-PU/?qs=sGAEpiMZZMtVoztFdqDXO6rEZqxeooRg |
| 5 | Prototyping Products PERF | $3.52 | http://www.mouser.com/Search/ProductDetail.aspx?R=3400virtualkey53400000virtualkey534-3400 |
| 6 | Short Ethernet Cable | $1.85 | http://www.mouser.com/Search/ProductDetail.aspx?R=73-7772-3virtualkey60140000virtualkey601-73-7772-3 |
| 7 | 328P Socket | $0.17 | http://www.mouser.com/ProductDetail/ADAM-TECH/ICS-328-T/?qs=sGAEpiMZZMs%2fSh%2fkjph1tjJZclYmfaNPQIybKKimdeU%3d |
| 8 | Slide Switch | $0.86 | http://www.mouser.com/ProductDetail/CK-Components/SS-22D0205-G-2/?qs=sGAEpiMZZMtHXLepoqNyVaknRufv4Zo6qcVo9DbgqFo%3d |
| Total: | | $43.99 |

The circuit diagram below in Figure 4 shows the proper wiring for the device. It is important that the master and slave pins be wired correctly for the SPI protocols to function properly.

Schematic.tif

*Figure 4. Circuit diagram showing the proper connection of the pins*

To fabricate the device, we placed the microcontroller and the Ethernet controller on a square piece of perfboard. A smaller board will make the device more concealable, but also slightly more difficult to wire and solder.



*Figure 5. Completed device, the battery pack has been detached from the right figure to display the underside of the board*

As can be seen in Figure 5, wire running along the top of the board was used to connect pins that were more than a row apart, and solder bridges underneath connected adjacent pins and the ends of wires to their destination pins. One important design note, as can be seen in Figure 5, the Reset pin on the Ethernet controller board is wired directly to the power source. This is to keep the reset logic stable, if the reset pin is not kept high, the board could reset at random and indefinitely. To prolong the battery life of the device, a power switch is included on the board. The switch used in Figure 5 is a slide switch, which diverts the flow of current from one set of pins to the other on either end of the switch. All the pins to be wired to power, are wired to one of the outer set of pins on the switch, while the power source is wired to the center set of pins. When not in use, slide the switch to the set of pins not wired to the circuit to open the circuit and prevent the flow of current.

## Step By Step Implementation

After building the board, this is how you can get it working.

1. Download and Install Atmel Studio 6(<http://www.atmel.com/microsite/atmel_studio6/>).
2. It would be helpful to install Wireshark (<http://www.wireshark.org/>) on a machine o your network in order to observe the sniffing results.
3. Download the source code files, which can be found at <http://www.ics.uci.edu/~harris/hps/>
4. Edit the *PacketSniffer.h* file to change the destination IP and MAC addresses to the addresses of the machine which you want to receive the sniffed packets.
5. Using Atmel Studio 6, build the solution and program the microcontroller.
6. Place the microcontroller into the socket on the HPS device.
7. Plug the HPS into the target Ethernet jack and turn on the device.
8. View the transmitted packets at the destination computer, using Wireshark or another tool.

## Summary

The HPS is a simple, do it yourself packet sniffer for penetration testing. The device is easy to fabricate, and much cheaper than other penetration testing devices offered on the market. Using the source code discussed in this paper, the device can be programmed to send data to a device with a known MAC and IP address.

#### References

* *http://www.globalscaletechnologies.com/p-46-sheevaplug-dev-kit.aspx* [1]
* *http://pwnieexpress.com/products/elite-plug* [2]
* *http://www.atmel.com/Images/doc8161.*pdf [3]
* *http://ww1.microchip.com/downloads/en/devicedoc/39662a.pdf* [4]
* *http://www.atmel.com/Microsite/atmel\_studio6/default.aspx* [5]
* *http://www.atmel.com/tools/avrjtagicemkii.aspx?tab=overview* [6]
* *http://www.mikroe.com/click/eth/* [7]