

# Project Final Report SCADA Attack System

April 24, 2007

Ryan Festag - [rfestag@gwu.edu](mailto:rfestag@gwu.edu)

## *Project Abstract*

The SCADA Attack System (SAS) provides wireless attack vector for devices receiving commands via a serial connection. Two devices are used in order to accomplish this: a SAS-Listener and a SAS-Attacker. The Listener is attached directly to an RS-232 line between a Remote Terminal Unit (RTU) and a serial modem. The serial modem is used to allow the command station to remotely dial in to the RTU. The Listener is placed on this communications channel so that it will not interfere with communications between the two devices, while still allowing it to transmit commands directly to the RTU and read what is sent by the command station. All data received from the command station is forwarded on the RF Link, and any data received on the RF Link from the Attacker is forwarded on the serial port. The Attacker is designed so that the attacker can send commands via the keyboard, or by pressing a button on the board. The button will be programmed to transmit the necessary reset command to the RTU.

## *Status*

The project was completed and everything ended up working as expected. The spy cable proved to be slightly difficult, but it has now been constructed in such a way that it allows for the Listener to sniff transmissions from the command station, as well as inject data into the line to the RTU, without affecting communications between the two. The RF Link works in both directions, allowing the Listener to forward received data to the Attacker, as well as allowing the Attack to send commands to the Listener so that they can be injected. I did end up needing to implement simple framing of my data in order to filter out most of the extraneous transmissions in my test environment. However, further development would need to be done on this in order to ensure reliable transmissions

## *Specification*

### *Hardware – Z8F6423 Development Kit:*

The Z8F6423 development kit was used in both the Listener and Attacker devices. It runs a Z8F6423 processor from Zilog at 18.432 Mhz, with 64 KB of programmable Flash memory and 4 KB of register RAM. Each comes with two built in UARTs, with one being connected directly to an RS-232 interface. The other UART can be accessed through on board pins, or it can be configured to utilize the on board infrared transceiver.

### Hardware – RF Link:

Two 434 Mhz RF Links from Sparkfun (TLP/RLP 434-A)<sup>1</sup> were used to establish bidirectional communications between the two devices. Each link contains a transmitter and receiver pair that has a maximum baud rate of 4800 bps. These devices are simple pass-through integrated circuits, meaning that simply writing bits to the data port of a transmitter will cause it to be transmitted, while any signal picked up by the receiver will be written to its data port. The transmitter requires 2-12 V to be supplied, while the receiver requires 3.3-6 V. The pinouts and dimensions of the transmitter and receiver pair are given below in Figure 1.

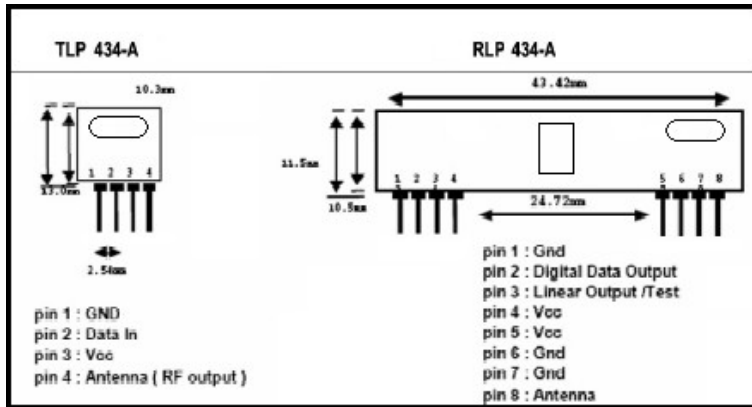


Figure 1: TLP/RLP 434-A Pinout

### Software - Overview

Since both devices function in essentially the same way, the same code was used on both boards. Each listens on RS-232 connection for any data. Once data is received, it is forwarded to the RF Link. When data is received on the RF Link, it is forwarded onto the RS-232 connection. In short, the boards act as bridges between the RF Link and the RS-232, essentially acting as wireless RS-232.

### Software – Serial Communications

The software utilizes two monitors: one for serial communications, and one for wireless communications. Serial communications are handled by two libraries: `sio.h` and `stdio.h`. The `sio.h` library gives access to the `kbhit()` function, which makes it easy to determine if a character has been transmitted on the serial line. By continually polling this function, it is possible to determine if a value is waiting to be handled. If a character is received, then it is forwarded to the RF Handler so it can be transmitted. In order to transmit data over the serial connection, I simply made calls to the standard `printf()` function, which then handled the rest of the transmission.

### Software – Wireless Communications

Unlike serial communications, where I was able to rely on standard libraries, I had to write my own simple driver for transmitting and receiving RF data. I elected to use

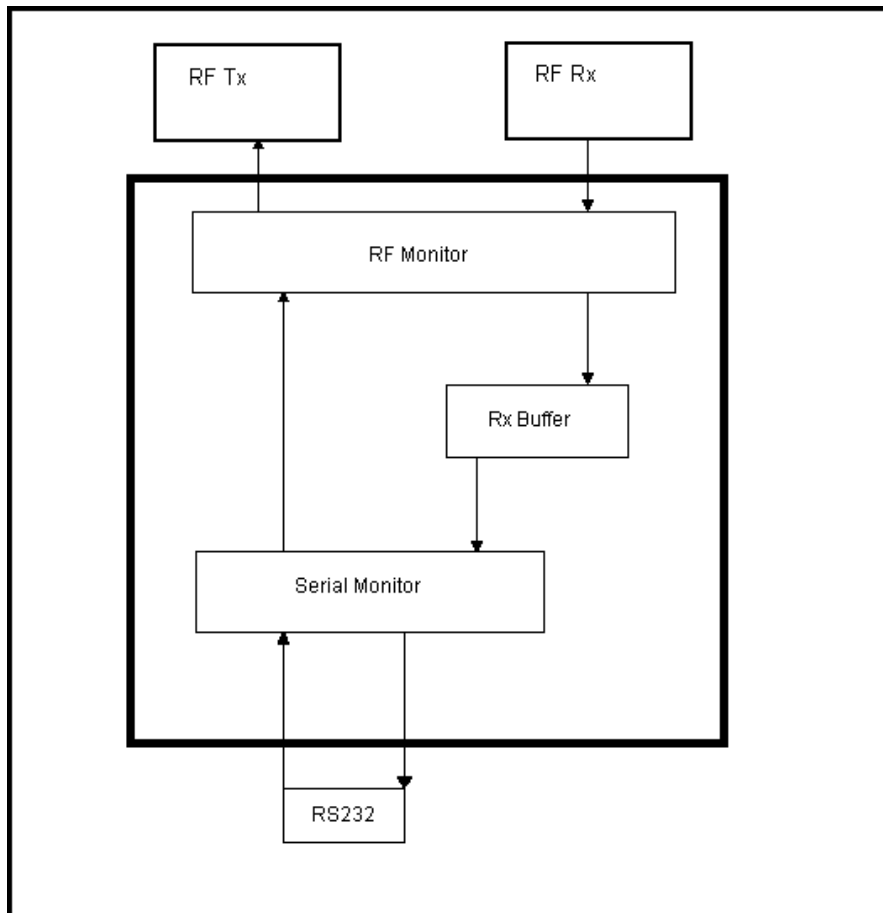
<sup>1</sup> Note: I could not find the actual data sheet for this device. Sparkfun has an incorrect datasheet. Instead, I referenced the KLP Walkthrough, since it contains the correct information regarding this device. The datasheets on Sparkfun's website are for a similar device that has a higher power requirement for the receiver than is actually necessary.

UART1 to handle general formatting, and to allow me to handle received data as an interrupt. A simple interrupt initialization function was written to set up the transmitter and receiver, as well as to create a vector for the ISR and set the baud rate for UART1.

The RF Handler required three components to function properly: an initialization function to prepare the pins and set the interrupt vector, a print function to allow for writing to the device, and a receive data interrupt. The printRF() function utilized a polled transmission method on UART1 to handle the transmission, while data reception was handled exclusively by interrupts that determined the validity of data and buffered it as necessary.

*Software – Overall Communications Infrastructure:*

The overall infrastructure allows the Z8 to act as a bridge between the serial RS-232 wired connection and the RF Links. Due to limitations on the baud rate for the RF Links, received data is buffered before it is transmitted via the serial line. The overall communications infrastructure is depicted below in Figure 1.



*Figure 2: Software Block Diagram*

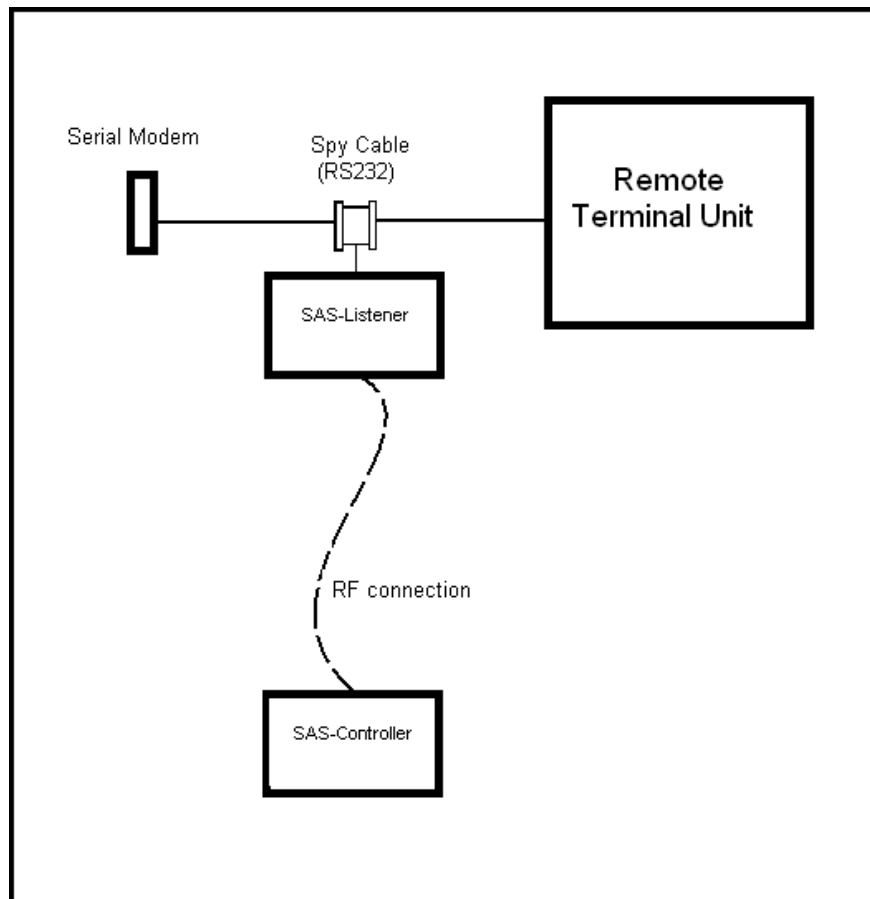
### *Software – Button and Timer*

In order to transmit special characters that could not be typed at the command line, the test button on the Z8 board was used to transmit the correct string of characters to cause a reset. Rather than using interrupts, I chose to use a timer that would cause an interrupt every .1 seconds. At that time, the button would be polled to determine if it had been pressed. If the button was pressed, a pre-configured character string was transmitted through the printRF() function.

## *Implementation & Construction*

### *Hardware – Overall Design*

The hardware was interfaced as described in Figure 4:



*Figure 4: Overall Implementation*

### *Hardware – Spy Cable*

The spy cable is essentially a modified null-modem cable between the Modem and the RTU. The Z8 is inserted into the line by tapping the transmit line of the modem. The wiring permits the Z8 to listen to communications between the Modem and the RTU, as well as transmit the data directly to the RTU.

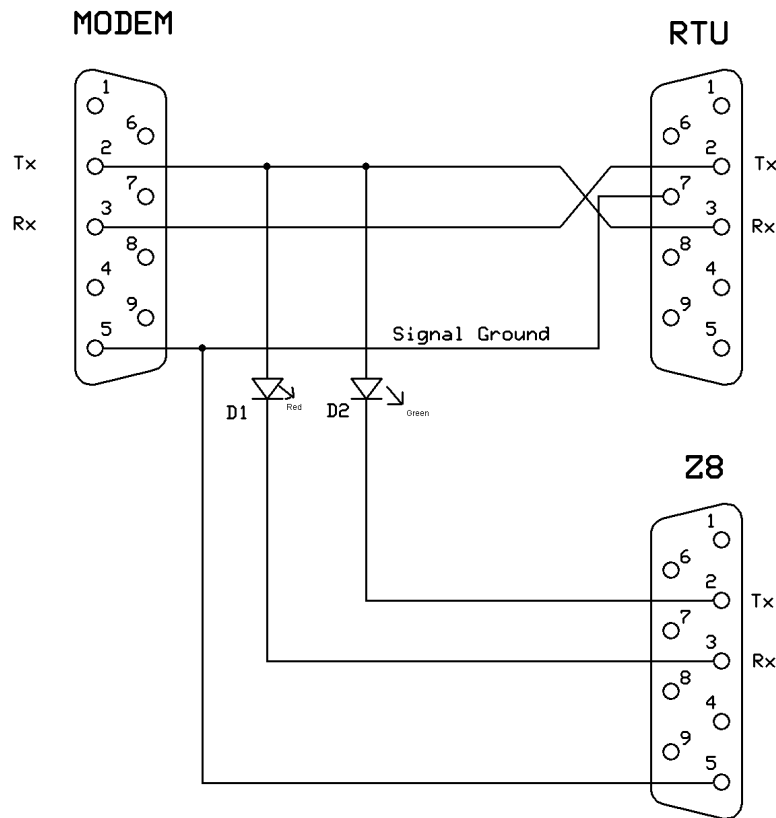


Figure 3: Spy Cable Wiring Diagram

### Software – RF Communications

RF Communications were accomplished using three components: an initialization function, a print function, and a receive ISR. Since the RF Links merely needed to be connected to the appropriate pins corresponding to UART1, no additional hardware was necessary to establish RF Communications.

The initialization function set up the appropriate pins to utilized UART, as well as to set the interrupt vector received data. It also set the baud rate of the communications. Because transmissions only occur when the printRF() function is called, it was simpler to implement data transmission using the polled method, rather than letting it be interrupt driven. As a result, interrupts are only generated when data is received.

Printing data to the link was accomplished using the printRF() function. This function was written to take in a pointer to the first character of a string, as well as the integer length. From experience, the first byte sent in a string was often lost by the receiver, so the first byte sent in any transmission was a “dummy” byte, which by default is a START. This is followed by an actual START byte, and then all data bytes (up to the length specified by the caller). Once all data has been transmitted, a STOP byte is sent.

Data is sent by first waiting until the Transmit Data Register Empty (TDRE) flag has been set to true. Once the data transmit register is empty, the new byte to be transmitted is loaded into the TXD register. The UART will automatically reset the TDRE flag when data is written to the TXD register. It is necessary to poll the TDRE bit before writing new data to the TXD register. Failure to do so will corrupt the data currently being transmitted.

Data reception is handled by the receive data ISR. The ISR determines what data to buffer, and when to print the data. Every valid piece of data received is checked to determine what should be done with it. If the data is a START byte, the RF Handler enters receive mode. Once in receive mode, any byte received by the RF Link will be saved in a buffer. If the buffer receives more than 100 bytes of data before receiving a STOP byte, the buffer is reset and the data is assumed to be junk, and the handler drops out of receive mode. If a STOP byte is received in that time period, the data is assumed to be legitimate, and is printed to the screen. The buffer is then reset, and the handler again drops out of receive mode. The block diagram for this functionality is given below:

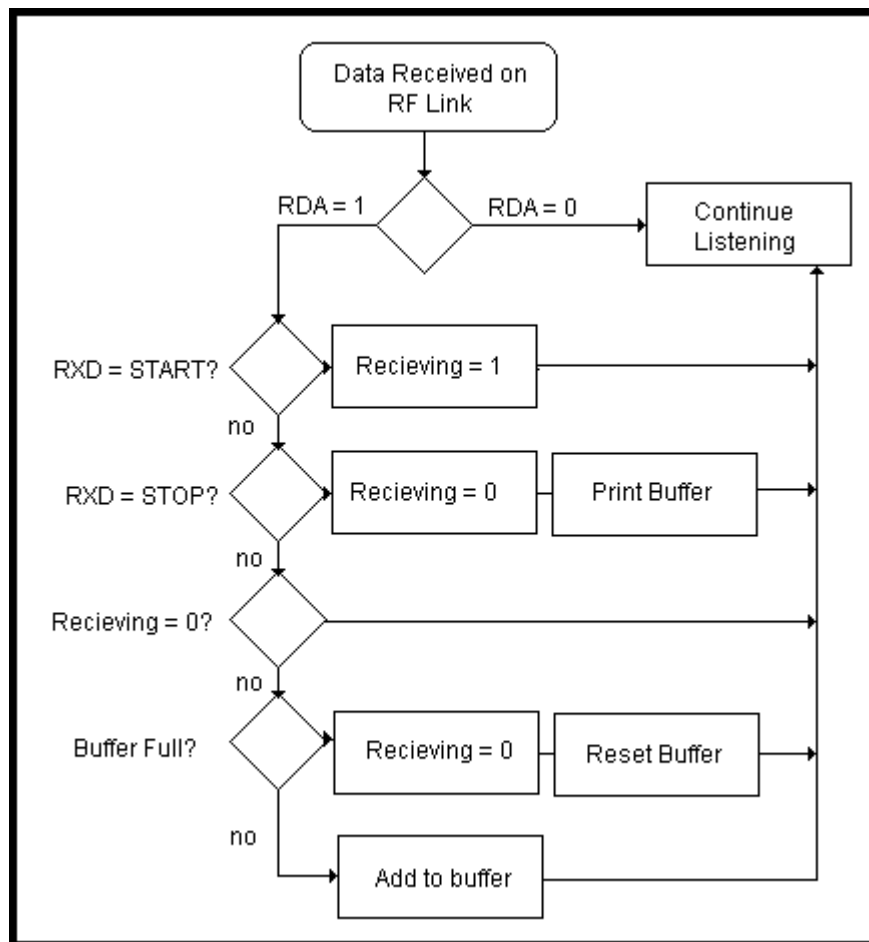


Figure 5: Receive Data Block Diagram

### *Milestones:*

#### February:

- All materials were purchased. Spy cable was designed.
- Initial tests of attack against RTU proved to be successful

#### March:

- Spy cable was constructed and tested.
- Initial coding for wireless link was completed. Rudimentary filtered communications worked, although noise had not yet been filtered. Transmissions only worked in one direction.

#### April:

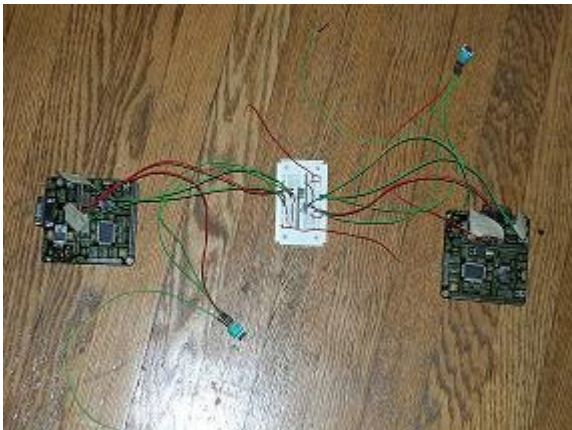
- Wireless link coding was completed to filter out most extraneous communications. Bidirectional communication was completed.

### *Tests and verifications:*

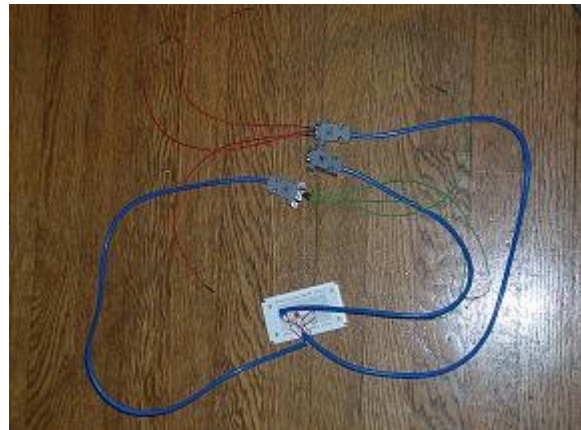
**Spy cable:** The spy cable was tested by connecting the phone port to the serial modem, the RTU port to the RTU, and the Z8 port to the Z8. I then dialed in to the serial modem, and was able to connect to it. When I connected to it and sent data from the command station, I was able to receive the data on the Z8. This was verified by the fact that the data was transmitted wirelessly from the Listener to the Attacker and displayed on the terminal attached to the Attacker. I also tested this by transmitting the reset command from the Attacker to the Listener. That data was forwarded to the Listener, and then forwarded to the RTU. The RTU restarted, indicating that it received the command correctly. The command station never printed out the message, indicating that it never received the data from the Z8, as expected.

**RF Link:** The RF Link was tested by transmitting data between the two devices. When both devices were connected directly to terminals, I was able to type at one and have the text appear on the other. The transmissions worked in both directions, proving that the communications were bidirectional.

### *Final Project:*



*Figure 6: Final Project (Listener and Attacker)*



*Figure 7: Final Project (Spy Cable)*

## *Retrospective*

In order to create a more useful wire tapping system, I would probably implement a means of tapping the data from both the command station and the RTU. Due to time constraints, I was unable to implement a potential design that would allow me to do this, but it is a potential continuation of this design. Theoretically, it should be possible to tap the transmit lines from both devices, and direct that to an unused pin on the Listener. By checking those pins whenever data is transmitted, it should be possible to determine the source of the transmission.

I would also like to find a pair of links that operate at frequency that is not used quite as often as the 434 Mhz links. Others are available from Sparkfun, so I would probably look into them for something that would be more likely to function in an environment with other devices on the same frequency. Unfortunately, the wireless keyboards we use in our classrooms operate on that frequency, and it took me a while to understand the strange results I was seeing in my tests were actually caused by this.

Finally, I would develop a more robust communications protocol, rather than relying on the UARTs and simple byte framing. In its current implementation, extraneous signals sometimes end up being received via the RF Links when there are other devices operating at the same frequency. A more robust solution would ensure that every bit received via the RF Link was transmitted and received by the intended recipient.

This project taught me a great deal about how devices communicate with one another. I learned how RS-232 works, and gained a greater understanding of signal propagation by creating the spy cable. In retrospect, only one diode is necessary in my current implementation of the spy cable. This diode should be used to keep transmissions by the Z8 from ending up back on the Z8's receive line. This can be done by connecting the receive line of the Z8 directly on to the transmit line of the command station, and then connecting the transmit line of the Z8 directly to the receive line of the RTU. A diode would be placed to allow the transmit line of the modem to connect with the receive line of the RTU. In doing so, transmissions from the Z8 would never touch the receive line of the Z8, but the transmissions from both the Z8 and the modem would reach the RTU.

I also learned about how RF communications work. I did not expect to have trouble with noise or extraneous signals, and was surprised when I ended up receiving gibberish on my devices. I was confused even more when I could not get bidirectional communications to work. It turned out that having the transmitters on the same board as the receivers was somehow interfering with the signals. When all transmitter and receiver pairs were wired initially, all communications were suppressed. I believe this is because both links are simple pass-through devices. Regardless of whether the UART is transmitting, its transmit line is always high or low, and that value is passed by the RF Link. If I were to go back, I would probably rewrite the code so that the power for the transmitters actually came from a GPIO pin, and I would treat that pin as an enable line. Although I would have to delay transmissions while I ensure that the device is powered up, it would reduce the amount of noise in the environment that needs to be filtered out.

## *Attachments*

This project includes the following:

Code:

- main.c
- buttons.c
- buttons.h
- rfhandler.c
- rfhandler.h
- timers.c
- timers.h
- timerdefs.h
- pindefs.h

Spec Sheets:

- KLP\_Walkthrough.pdf