SWIFT Wireless Fire Alarm System Analysis

Deniz Keskin (Advisor)
Research Scientist I
Georgia Tech Research Institute
Atlanta, Georgia, United States
deniz.keskin@gtri.gatech.edu

Daniel Chou
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
dchou33@gatech.edu

Siddharth Suman
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
ssuman33@gatech.edu

Aniyah Bussey
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
abussey6@gatech.edu

Jun Yeop Kim
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
jkim3663@gatech.edu

Sidney Wright
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
swright@gatech.edu

***Abstract*—As wireless fire alarm systems gain popularity due to increased convenience and more control over the various components for building administrators compared to a traditional wired system, they also introduce a new class of vulnerabilities that no wired system has. This ongoing study investigates what those vulnerabilities are, and how they might be exploited, for example, to induce panic in or create a dangerous environment for building occupants. This study specifically focuses on well-known building systems manufacturer Honeywell's Smart Wireless Integrated Fire Technology (SWIFT) solution, which includes wireless versions of pull stations, smoke detectors, and other common fire alarm system devices, along with a gateway which interfaces these wireless devices with an existing wired system.**

I. Background

A. Fire Alarm Systems

Fire alarm systems are used to alert building occupants and local fire authorities of the immediate danger of a fire. The core of these systems are the detection nodes, which can include heat, carbon monoxide, and smoke detectors, notification nodes, such as lights and sounders, and the "brain" of the fire alarm system, the fire alarm control panel (FACP) [1].

Traditionally, fire alarm systems have been connected, and thus communicate, via a physical wire. Many shortcomings are associated with a purely wired detection system, including detectors having reduced sensitivity over time and thus needing replacement [2]. Wireless systems, which communicate over radio frequency (RF) waves, allow for easier modification and expansion of the protection system, as it is not necessary to ensure all components are physically connected. While SWIFT does offer wireless devices, it is not a fully standalone system; it requires an existing wired system to hook up to. Thus, a fire alarm system using SWIFT will have both a wired subsystem and a wireless one, with the component known as the wireless gateway acting as a bridge between the two. The wireless subsystem includes the gateway, any wireless devices, and SWIFT Tools, the management software for the wireless network which communicates with the gateway via a wireless USB dongle. The wireless devices and FACP communicate with the gateway, which converts between wired and wireless communication. However, the conveniences of a wireless system come with the downside of exposing a larger attack surface. Since communication with the system is no longer restricted to a physical wire, attacks such as triggering a false alarm are much easier to carry out. Additionally, attacks such as holding a building's fire alarm system for ransom would simply not be possible in a fully wired system.

B. System Components

The FACP is used to control the functions of other systems in the fire alarm system and receiving information from detection and wired devices on the panel.

The wireless gateway is the main target for the team's project, as it controls the mesh network, which is composed of wireless devices, by managing its formation and configuration while also interfacing the wireless mesh network with the wired network. The gateway has two processors that communicate with a Universal Asynchronous Receiver-Transmitter, or UART channel, and this paper refers to them as the SLC and the RF chips, which interface the wired devices and handle wireless communication and the bootup process, respectively. The UART is a protocol that handles asynchronous serial communication between a computer and its attached serial devices (SLC, RF, etc.). The gateway has three firmware update files; they contain the bootloader firmware, the RF firmware, and the SLC firmware. [2]

The RF chip, or Radio Frequency chip, is outlined in Figure 1 with an orange square. RF module in general is an electronic device for wireless communication with other devices, and for the gateway, integrated UHF (ultra-high frequency) transceiver is used for RF transmitting and receiving, and RF chip for signal processing. RF firmware is contained in the `WSG_RF_3.0.bin` file which is run during normal operation of the RF chip. `WSG_BU3_RF_3_0.bin` runs on the RF processor, which contains the bootloader firmware and handles initial bootup and recovery modes. The RF processor is responsible for managing the mesh network that contains

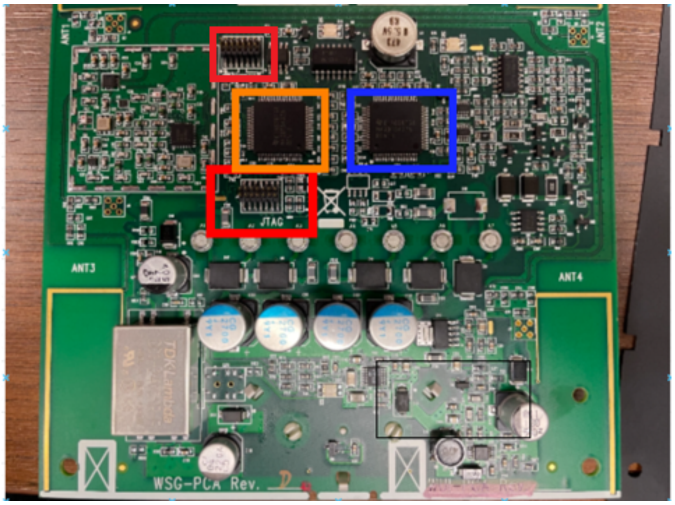


Fig. 1. The wireless gateway board, with various components outlined, such as the RF (orange) and SLC (blue) processors, as well as the two JTAG headers (red).

all of the wireless devices, interpreting wireless signals, and updating itself and the SLC processor when firmware is updated.

One of the primary components of the fire alarm system that the team is analyzing is the Signaling Line Circuit chip, or SLC chip, whose processor is depicted in 1 outlined with a blue square. The primary purposes of SLC are to carry signals and provide power for the addressable device modules in the fire alarm system. It is an important component as it is responsible for carrying the signals that report device status and give instructions to the devices on the panel. For the purposes of this project, the team is trying to reverse engineer the gateway's SLC firmware. This is important because it can show the details about the specific instructions/commands the SLC sends to the fire alarm system so that the team can learn from it and eventually use that information to control the system.

II. Previous Research

Initial work on reverse-engineering the system was mainly done on the SWIFT tools management software and the over-the-air (OTA) protocol used by wireless devices to communicate with the gateway [3]. Both protocols were fully decoded due to major similarities, and the fact that SWIFT tools is distributed as an unobfuscated .NET application, which contains much more information than traditional executables, making the reverse-engineering process easier.

To begin analysis on the wired side of the gateway, the team used BinDiff, which analyzes two binaries and reports similar code segments, to see if any work done on the firmware for the RF chip or wireless pull station would be transferable [4]. The team compared the SLC, RF, and wireless pull station binary files. The RF and pull station binary files had a lot of overlap, but the team didn't find many similarities between the SLC and RF firmware. Next, the team had to import symbol labels into Ghidra in order to identify the memory-mapped peripheral registers provided by the MSP430 processor, and lastly, set up the memory map and entry point, which is known to be at address 0x5c00 for any binary running on the MSP430F5437A processor, in Ghidra. From observing its decompiled outputs, it was found that the SLC firmware exposes its universal serial communication interface (USCI) channel A1 via headers on the gateway board; this channel is used for debugging output from the SLC chip. The team also identified use of the serial peripheral interface (SPI) protocol, though were not able to identify what was being communicated with.

Another previous goal of the team was to bypass the SWIFT wireless gateway's firmware integrity check, since the gateway is a primary target for attackers and it is important to determine how to protect this single point of failure [4]. To do this, the team used Ghidra to analyze the bootup (BU) firmware, which is a small binary that contains logic for checking integrity of the chip's memory after a firmware upload by performing a cyclic redundancy check (CRC) and comparing against a checksum hard-coded into the new firmware. By utilizing the debug features provided by TI's Code Composer software, the team was able to find the exact ranges of memory being accessed, which allowed for a tool to be developed that could calculate the CRC for an arbitrary firmware binary. This tool is a python script that can replace the hard-coded CRC values and replace them with the newly calculated CRC value, which ensures that it will pass Honeywell's firmware certification.

The team also developed a module, which is made up of a CLI (command line interface) and a GUI (graphic user interface), that streamlines the firmware upload process for Honeywell's SWIFT Gateway by re-implementing the full six step protocol used by SWIFT Tools. The CLI is a Python script that accepts up to three firmware binaries (SLC, BU, and RF), patches the BU and RF firmware using the CRC tool, and uploads the inputted firmware to the Wireless Gateway. The GUI does this process graphically. Due to the development of the the firmware update CRC verification bypass, a substantial modification to the firmware can be created, which makes uploading arbitrary firmware to the SWIFT Wireless gateway possible.

III. SLC Analysis

Previously, the team had attempted to reverse-engineer the SLC processor firmware, in an attempt to understand the SLC wire protocol and be able to send arbitrary, well-formed SLC messages, possibly by modifying the gateway firmware and uploading it using the custom firmware upload vulnerability previously found by the team. This would allow for potential attacks on the wired fire alarm network, something that might be difficult to do with just the built-in functionality of the gateway.

To focus and better direct the reverse-engineering effort, the team had previously started continuity tests on the wireless gateway's circuit board, but these were constrained by time. The team has now been able to perform more thorough testing of the gateway circuitry, with the primary goal of determining how the SLC processor can communicate on the SLC line, given the voltage difference between the processor's general-purpose IO (GPIO) pins (3.3V) and the SLC line (15V) [1].

A. SLC USCI B1

However, one additional minor goal from the previous testing was to determine the purpose of the SLC chip's USCI B1 port, which is referenced extensively in the SLC firmware. Based on the team's previous reverse-engineering work, this serial channel is used to send and receive commands, which appear to include synchronization, reboot of the SLC chip, and adding and removing devices from the SLC loop [4]. The team has now identified a connection between this port on the SLC chip, and the USCI A1 port of the RF chip, which almost certainly makes it the method for inter-processor communication, for functionality such as facilitating communication between the (wired) FACP and the wireless devices in the mesh network, and updating the SLC chip's firmware (which is initiated by the RF chip). This would provide an explanation for why commands for device addition and removal exist, which the team was previously less confident in attributing as such.

B. Gateway SLC Message Sending Capability

Returning to the primary goal of determining the method of communication between the SLC chip and SLC line, such information would be useful in determining which parts of the firmware binary deal with SLC communication, and thus which parts to focus on to reverse engineer the SLC protocol. The team has used the results of these continuity tests to start to create a circuit diagram using the schematic software DipTrace for the gateway board components connected to the SLC line. Though not complete, the team has identified a likely method for the gateway to short the SLC line, controlled by the SLC chip's port 1.5 pin (the sixth least significant bit of GPIO port 1), as shown in figure 2. This connects to the gate of an NPN Darlington transistor, whose collector is connected to the shared SLC IN/OUT wire, and whose emitter is tied to SLC ground. Thus, when this pin asserts a logic high, current is allowed to flow from collector to emitter, shorting the SLC line to ground, which will result in a logic zero on the outgoing signal. Since SLC messages are defined in terms of logic lows [5], as the line must normally provide power to the devices connected to it, this constitutes the SLC message-sending capability for the gateway.

While the team has not had time to begin fully investigating how port 1.5 is used and referenced within the SLC firmware, it is very plausible that this is the method used by the gateway to send SLC messages, for two main reasons. The first is that port 1.5 is explicitly configured during the SLC chip's "initialization" functions as output. The second is the use of a Darlington transistor, as opposed to a normal bipolar junction transistor (BJT). The construction of a Darlington transistor makes its logical behavior comparable to a normal BJT, except that because of the configuration of its two constituent BJTs, a Darlington transistor has a much higher current gain, which means it is able to switch much higher currents than a single BJT using the same base current [6]. This is important because the base is driven by a GPIO pin, which is limited to a couple milliamps, while the amount of current resulting from shorting the 15V input to ground will obviously be orders of magnitude higher.

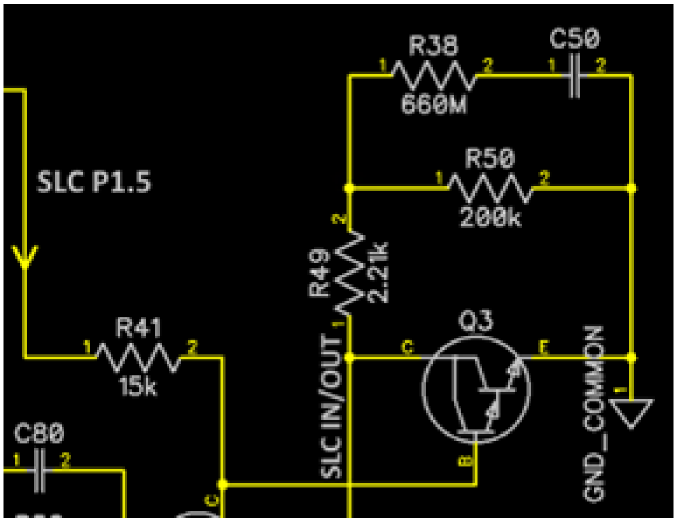


Fig. 2. Schematic diagram showing connections of gateway component Q3, a Darlington transistor, which likely provides a method for the gateway to send messages on the SLC.

C. JTAG Labeling

In addition to continuity tests on the gateway board components, the team also started continuity tests to determine the functions of the two JTAG headers present on the board. Previously, the team had identified several pins on the lower header exposing the RF processor's JTAG debugging pins, which would be used in conjunction with TI's MSP-FET (flash emulation tool) and Code Composer Studio. Additionally, pins 1 and 3 on the upper header had been identified as being connected to the USCI A1 pins of the SLC processor, which were used for that chip's debugging output [4]. The team identified all necessary pins for debugging the SLC processor using the same setup as the RF chip, as shown in figure 3, and successfully connected the FET debugger to the SLC chip.

D. SLC Chip Memory Pull Analysis

With this, the memory contents of the SLC chip were dumped and disassembled in Ghidra. The team was able to confirm that, for the most part, the firmware on the SLC chip was identical to the firmware distributed by Honeywell, and using Ghidra's version tracking tool, analysis done on the firmware version was easily transferred over to the memory pull. However, similar to the RF chip, the SLC firmware has an additional section of code at address 0xD600 that is not present within the distributed firmware. From writes to the

[1] Previously, the team had assumed that the SLC line carried a 24V signal; testing using a multimeter showed this to not be the case. Instead, the line appears to be at a lower voltage of 15V.

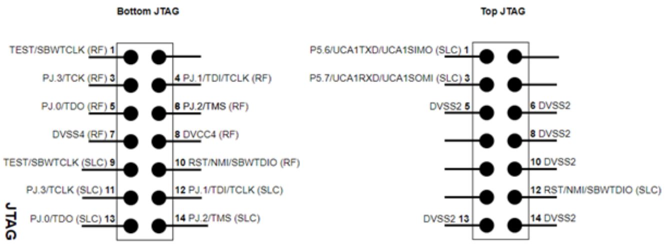


Fig. 3. Labelled diagram of both JTAG headers present on the gateway board. Pins with no label are not known to connect to any other component.

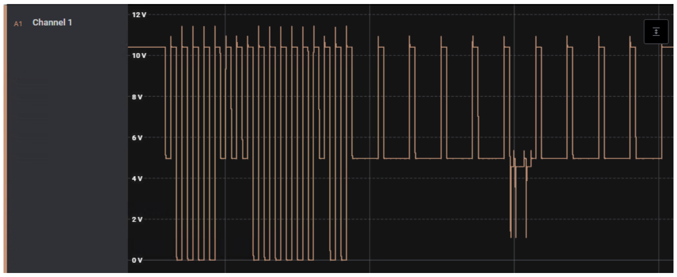


Fig. 4. Analog capture from Saleae's Logic Pro 16 of the SLC line. Three distinct voltage levels are visible: 10V, 5V, and 0V.

USCI A1 debugging output and structural similarities to the RF boot firmware, it is very likely that this section handles firmware verification. Unfortunately, due to the gateway not being in a normal functioning state, the team has not yet been able to fully take advantage of the dynamic reverse engineering capabilities provided by the debugger to make further progress in decoding the SLC protocol.

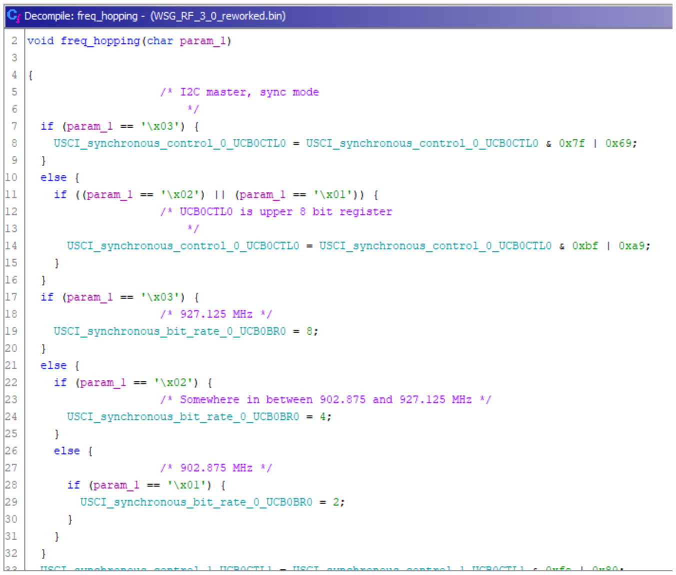
E. *KeySight Logic Analyzer*

Previously, the team had used a Saleae Logic Pro 16 logic analyzer to record and analyze SLC signals. However, because this device only has an input voltage range of ±10V, the full range of the SLC signal is not visible, and the team was unsure as to whether the halfway voltage drops seen in figure 4 actually occur, or if they are simply artifacts of exceeding the input voltage range. In an attempt to resolve this, the team investigated using a KeySight 16860 Series portable logic analyzer, which has a much higher input voltage range, to record the signals on the SLC. However, unlike the Saleae, the team found that the KeySight is not capable of giving analog readings, and the voltage threshold used for its digital waveforms only goes up to 5V. Therefore, this device will unfortunately not be helpful in resolving the halfway voltage drop question.

IV. RF Binary Reverse Engineering

A. *RF USCI Reverse Engineering*

A sub-goal of the software team was to understand SLC USCI functionality through reverse engineering. This was accomplished through analysis of a function namely called ”frequency_hopper”. 5 The function is responsible for changing the operating frequency of the receiver-transmitter. It is still uncertain whether the frequency is responsible for triggering certain mechanisms or commands in the gateway itself, but if it is, there is a possibility for malicious actors to recreate signals at the operating frequency and spoof an outcome.



```c
void freq_hopping(char param_1)

{
                    /* I2C master, sync mode
                        */
  if (param_1 == '\x03') {
    USCI_synchronous_control_0_UCB0CTL0 = USCI_synchronous_control_0_UCB0CTL0 & 0x7f | 0x69;
  }
  else {
    if ((param_1 == '\x02') || (param_1 == '\x01')) {
                    /* UCB0CTL0 is upper 8 bit register
                        */
      USCI_synchronous_control_0_UCB0CTL0 = USCI_synchronous_control_0_UCB0CTL0 & 0xbf | 0xa9;
    }
  }
  if (param_1 == '\x03') {
                    /* 927.125 MHz */
    USCI_synchronous_bit_rate_0_UCB0BR0 = 8;
  }
  else {
    if (param_1 == '\x02') {
                    /* Somewhere in between 902.875 and 927.125 MHz */
      USCI_synchronous_bit_rate_0_UCB0BR0 = 4;
    }
    else {
                    /* 902.875 MHz */
      if (param_1 == '\x01') {
        USCI_synchronous_bit_rate_0_UCB0BR0 = 2;
      }
    }
  }
  USCI_synchronous_control_1_UCB0CTL1 = USCI_synchronous_control_1_UCB0CTL1 & 0xfe | 0x80;
```

Fig. 5. Frequency hopping function with designations of each line's purpose

The operating frequencies are between 902.875 and 927.125 MHz. The if and else parameters are responsible for determining the frequency. The parameter with the lowest hexadecimal is likely the lowest switchable frequency, while the highest is likely the highest frequency. There remaining else statement likely operates between 902.875 and 927.125 MHz. Additionally, it is clear that the USCI_synchronous_control_0_UCB0CTL0 is also changing based on the initial hexadecimal parameter. The synchronous control variable is responsible for setting the sync mode. The 0x03 parameter is likely turning it on while the 0x02 and 0x01 turn sync mode off.

V. Debugging the Gateway

A. *Connecting the MSP-FET*

While connecting the flash emulation tool to the SLC chip, the gateway was placed in an unidentified error state of solid yellow and red indicator lights due to a misalignment of the JTAG expansion board (see 6). In this state, there is no connection to the W-USB in SWIFT Tools, which impedes proper operation and analysis of the device. Despite efforts to reconfigure the setup, the state persisted, and the gateway could not boot.

To test the function of the MSP-FET and observe the bootup procedure, it was connected to the RF chip where it had previously been used to dump firmware binaries for reverse engineering following the connections in 7. Debugging with Code Composer Studio (CCS), TI's development environment for its microcontrollers and processors, would enable the team to set breakpoints for further understanding of the boot process

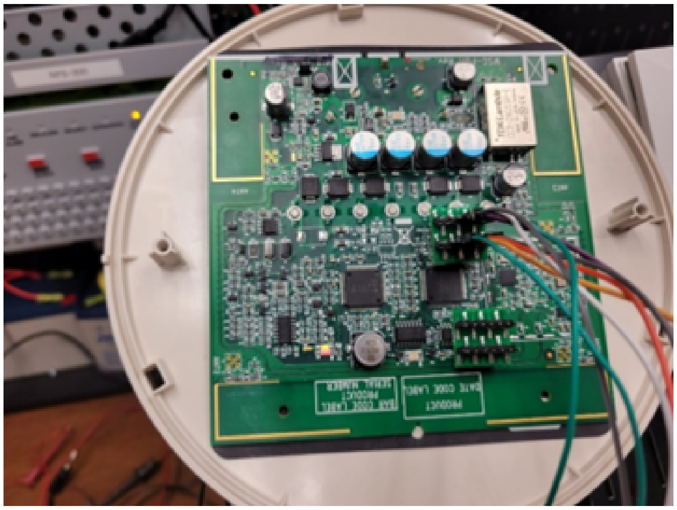


Fig. 6. The gateway board while in the unidentified error state. The indicator lights can be seen in the bottom left.

JTAG symbol shows orientation of the JTAG pins relative to the Gateway PCB

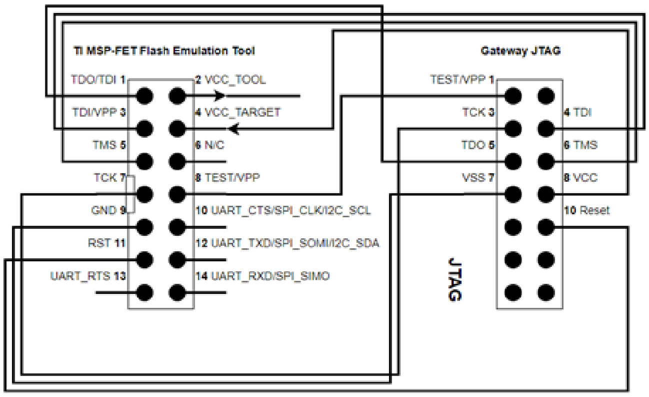


Fig. 7. Wiring diagram for attaching the MSP-FET debugger to the gateway's JTAG header.

that caused the error state. The CCS project's build process failed due to the lack of a function to debug, so the team created a placeholder "main" function to run 6. Although the debug session caused the indicator lights to turn off, it also overwrote a section of memory at 0x10000 with the main function that was required by the CCS project. The memory segment was written to Bank B of code memory on the MSP430 chip [7], meaning it overwrote an unknown function in the chip's firmware.

B. *Flashing Firmware*

With the firmware on the RF chip corrupted, a reset was necessary to restore its function. The Honeywell-provided firmware update service packs contain original firmware known to work with the gateway; as a result, the team decided a factory firmware binary could be loaded onto the chip using CCS alongside the MSP-FET to achieve a reset on the system.

Preliminary attempts to write to memory on the RF chip included flashing the entire file `WSG_BU3_RF_3_0.bin` from location 0x0 to 0x45BFF. The team was unsuccessful because BSL memory is a protected section that requires write permissions gained from a user-defined password [7]. Since BSL memory was not affected by the overwrite, it was not necessary to include it in the next attempt. As such, the correction began at RAM, a non-flash writable section of memory beginning at 0x1C00, until the end of the file at 0x45BFF. A Python script was created to assist in parsing the firmware binary into sections, outputting a new file to load onto the chip. An issue that arose while writing the script was the format of the output file, which did not contain the same raw hexadecimal data as the original file. However, the necessary formatting is achieved by the use of the read and write bytes mode of opening files in Python. The final script opens a firmware file, reads the bytes, and discards data before 0x1C00, storing everything after in `parsed_memory.bin` for loading from a file in CCS.

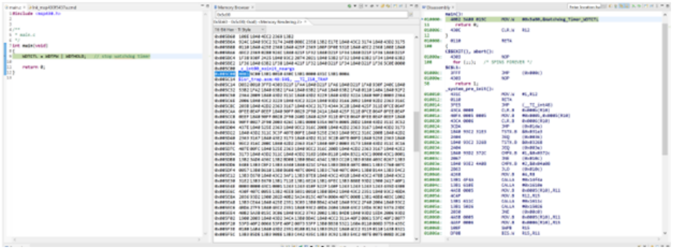
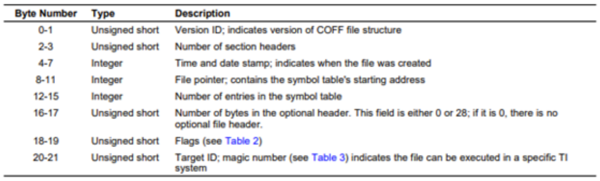


Fig. 8. The C placeholder function flashed onto the RF chip, and its overwritten memory.

Combining the memory parsing script with the gateway bootup firmware, the team was able to obtain a writable section of memory from the provided file. During the process, CCS had problems writing to memory despite a valid section being chosen. The solution seemed to be power cycling the gateway by replacing the PCB and restarting CCS in order to write successfully, but this method does not work effectively every time. Even though `WSG_BU3_RF_3_0.bin` was written, the gateway did not connect to SWIFT Tools. When examining the code in the chip, there were large sections filled with 0xF values that were not meaningful. The same process of loading memory was utilized with firmware from `WSG_RF_3_0.bin`. Once a write succeeded, the gateway was power cycled, and it displayed alternating red and green blink indicators. These indicate that the gateway is in boot-loader normal mode and is ready to update [2].

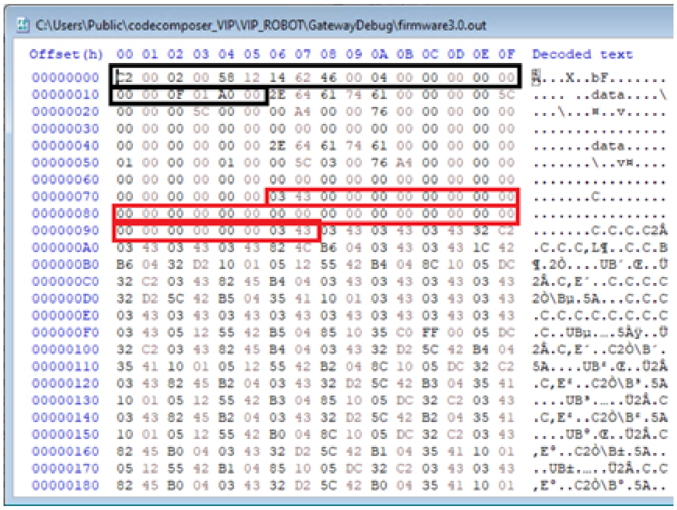
C. *COFF files*

Common Object File Format, COFF, is a file format that is used to store compiled code such as file outputs from a linker or a compiler. [8] In other words, COFF files are files that are created from assembler and link step objects. As the memory parsing script was not able to write to memory despite that a valid, designated memory section was chosen, another approach attempted was using the COFF files to write to memory. The reason for this new approach was the possibility that processors could have protections against flashing raw binary



Byte Number	Type	Description
0-1	Unsigned short	Version ID; indicates version of COFF file structure
2-3	Unsigned short	Number of section headers
4-7	Integer	Time and date stamp; indicates when the file was created
8-11	Integer	File pointer; contains the symbol table's starting address
12-15	Integer	Number of entries in the symbol table
16-17	Unsigned short	Number of bytes in the optional header. This field is either 0 or 28; if it is 0, there is no optional file header.
18-19	Unsigned short	Flags (see Table 2)
20-21	Unsigned short	Target ID; magic number (see Table 3) indicates the file can be executed in a specific TI system

Fig. 9. Designation of bytes for file header defined by TI literature including a target ID for specification of the target processor. [8]



C:\Users\Public\codecomposer_VIP\VIP_ROBOT\GatewayDebug\firmware3.0.out

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   C2 00 02 00 58 12 14 62 46 00 04 00 00 00 00 00  Â...X..bF.......
00000010   00 00 0F 01 A0 00 2E 64 61 74 61 00 00 00 00 5C  ..... ..data....\
00000020   00 00 00 5C 00 00 00 A4 00 00 76 00 00 00 00 00  ...\...¤..v.....
00000030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000040   00 00 00 00 00 00 2E 64 61 74 61 00 00 00 00 00  .......data.....
00000050   01 00 00 00 01 00 00 5C 03 00 76 A4 00 00 00 00  .......\..v¤....
00000060   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000070   00 00 00 00 00 00 03 43 00 00 00 00 00 00 00 00  .......C........
00000080   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090   00 00 00 00 00 00 03 43 03 43 03 43 03 43 32 C2  .......C.C.C.C2Â
000000A0   03 43 03 43 03 43 82 4C B6 04 03 43 03 43 1C 42  .C.C.C‚L¶..C.C.B
000000B0   B6 04 32 D2 10 01 05 12 55 42 B4 04 8C 10 05 DC  ¶.2Ò....UB´.Œ..Ü
000000C0   32 C2 03 43 82 45 B4 04 03 43 03 43 03 43 03 43  2Â.C‚E´..C.C.C.C
000000D0   32 D2 5C 42 B5 04 35 41 10 01 03 43 03 43 03 43  2Ò\Bµ.5A...C.C.C
000000E0   03 43 03 43 03 43 03 43 03 43 03 43 03 43 03 43  .C.C.C.C.C.C.C.C
000000F0   03 43 05 12 55 42 B5 04 85 10 35 C0 FF 00 05 DC  .C..UBµ.…5Àÿ..Ü
00000100   32 C2 03 43 82 45 B4 04 03 43 32 D2 5C 42 B4 04  2Â.C‚E´..C2Ò\B´.
00000110   35 41 10 01 05 12 55 42 B2 04 8C 10 05 DC 32 C2  5A....UB².Œ..Ü2Â
00000120   03 43 82 45 B2 04 03 43 32 D2 5C 42 B3 04 35 41  .C‚E²..C2Ò\B³.5A
00000130   10 01 05 12 55 42 B3 04 85 10 05 DC 32 C2 03 43  ....UB³.…..Ü2Â.C
00000140   03 43 82 45 B2 04 03 43 32 D2 5C 42 B2 04 35 41  .C‚E²..C2Ò\B².5A
00000150   10 01 05 12 55 42 B0 04 8C 10 05 DC 32 C2 03 43  ....UB°.Œ..Ü2Â.C
00000160   82 45 B0 04 03 43 32 D2 5C 42 B1 04 35 41 10 01  ‚E°..C2Ò\B±.5A..
00000170   05 12 55 42 B1 04 85 10 05 DC 32 C2 03 43 03 43  ..UB±.…..Ü2Â.C.C
00000180   82 45 B0 04 03 43 32 D2 5C 42 B0 04 35 41 10 01  ‚E°..C2Ò\B°.5A..
```

Fig. 10. Bytes from `firmware3.0.out` with the COFF file header outlined in black and the beginning of raw data in red. Decoded text is displayed to the right.

files, while COFF files are able to bypass said protections. Initially, a command line tool to convert binaries into COFF was used to generate a writable file [9]. When attempting to flash the generated file, the team encountered an error in which Code Composer did not recognize the target device of the file.

After further investigation of object file dumps from the chip, it was found that TI has specific metadata for COFF files that can be written to their chips. The metadata consists of a 22-byte file header, one or more 48-byte section headers, raw data, data relocation information, a symbol table, and a string table. 9 The team utilized a file that flashed correctly as a template for the metadata, finding that it contained two section headers beginning with label strings which decoded to "data" at 0x16 and 0x46. 10 Outlined in black is a file header that the team examined and recreated within the generated COFF file. Bytes 20 and 21 of the file header at 0x17 and 0x16, respectively, describe the target of the file by a magic number. For the MSP430, these bytes need to be set to 0x00A0 to pass the target check and be written to the chip's memory.

The attempt to flash a COFF with file header bytes has resulted in only a small section of BSL memory and blocks being changed due to the file not mapping to memory correctly. This incorrect mapping happened because it contained metadata that was appended by the team and generated metadata from the command line tool even though they conflicted each other with duplicate headers. The file was edited to contain the appended file header immediately followed by the generated section header, and the correct mapping was produced. As with binary files, COFF files could not be written to protected BSL memory, leading to the write failing.

Later, the team flashed the COFF with a header to start at address 0x1800, which is writing to unprotected information memory instead of BSL, suspecting that the gateway was working incorrectly because neither section corresponded to the accurate firmware in binary. Writing to information memory also failed because Code Composer could not write to flash memory locations. Because writing to flash memory did not work, the team decided to set up a BSL scripter that could write to the corrupted BSL and information memory.

D. BSL Scripter Setup

BSL Scripter refers to a program developed by TI to interface with the bootloader or bootstrap loader (BSL) of their suite of microcontrollers [10]. The scripter is built with various commands that can be written in a text file, passed to the program in command line, and run on the chip. Since the BSL and information sections of memory contained differences with the normal mode firmware, the team took interest in the `RX_DATA_BLOCK` command, which would allow for write access to those flash memory sections of the MSP430. It is important to note that due to BSL Scripter not being provided as an executable but instead as a C++ project, the program needed to be built in Visual Studio (VS). The setup of BSL Scripter consisted of building dependencies, configuring the project, and building the executable.

There were three main project dependencies: Boost libraries for C++, HIDAPI, and libusb. The Boost libraries were configured to be recognized in the TI-provided VS project by adding their path to the project's additional include libraries for the linker and the project in general. Boost also depends on the Microsoft C++ compiler cl.exe, and the installation is contained within the Visual Studio directory because it is included with its C/C++ build tools. After adding cl to environment variables, Boost setup scripts were run to add filesystem, system, and program options libraries. Boost builds different types of library files depending on the target, and BSL Scripter requires statically linked libraries, meaning the parameter `runtime_link` must be static during Boost build. Although HIDAPI and libusb files need to be built according to the guide, the team found that the necessary header files were already included in the BSL download from TI.

While configuring the project, the team was delayed due to versioning issues related to Visual Studio and its build tools. The BSL Scripter getting started guide recommends the 2013 package that includes Visual Studio and the v120 toolset, but VS 2022 was used with an attempted install of v120 alongside it. Backward compatibility appeared to be supported; however, VS continued to use the latest v143 instead. Automatic migration to a newer version was available, but the team was unsure of the effects that it would have on the software. As a result, everything was reconfigured with

```
LOG
MODE 5xx UART COM6
TX_BSL_VERSION
TX_DATA_BLOCK 0x5C00 0x2F output.txt
```

Fig. 11. The script that tested the BSL Scripter setup

VS 2013 Express. Building the executable was as simple as building the VS project itself after setting up the dependencies. Once complete, the BSL Scripter program appears within the project folder, making it straightforward to open the location in the command line to test.

Before running a script, the hardware needed to be connected. The MSP-FET debugger has a BSL interfacing mode that it can switch to after a power cycle, which is what the team used. In addition to the RST and TEST JTAG pins and ground, the debugger requires the voltage supply of the device connected to VCC Tool and UART transmit and receive pins located at 1.1 and 1.2, respectively, for the MSP430F5347A. 3 Unfortunately, the transmit and receive pins are not exposed on the gateway JTAG header, leading the team to use micro grabber test leads to connect to the chip directly. In the connection process, the team encountered further obstacles because the micro grabbers shorted each other due to the chip being so small. Instead of soldering the board, the team conducted continuity tests while maintaining the leads in various positions to determine the best placement even though they were precariously close to touching.

E. Running a Script

On the software side, TI provides test scripts that can blink LEDs connected to the chip, but the team created a small script to send to the MSP430 that would read the BSL firmware version and the block of code beginning at 0x5C00. The script was written to comply with the standards defined in TI's manual.

The first line is the initial setup of the script to log the output in a text file while the second defines the target family of the chip, the serial communication protocol, and the port. The COM port is not that of the MSP-FET itself, but a UART backchannel exposed by the tool for BSL interface purposes. Running the script yielded an Unknown ACK value error after every command except for LOG, and none of the data was transmitted. Since the MODE command failed, it is likely that the debugger was not able to establish communication with the chip. This could be due to the chip not entering BSL mode through a sequence on the RST and TEST pins or a problem with the connection of the transmit and receive pins. The team was not able to resolve the issue and working with BSL Scripter to fix the gateway is a goal of future work.

VI. Conclusion

Overall, the team has made good progress in identifying how the gateway interfaces with the SLC line, and has learned more about various aspects of the MSP430 processor architecture in trying to debug and fix the gateway.

A. Next Steps

The team has two distinct, major goals for future work. The first is to continue working to diagnose the current (non-functioning) state of the gateway and get it into a normal, functioning state. This will be immensely helpful in dynamically reverse-engineering the gateway functions and hardware, and the team has a promising lead in the form of the BSL Scripter. The second is, after identifying a method for SLC message sending, the team would like to know how messages are received by the SLC chip. This will help immensely in understanding the SLC protocol by reverse-engineering the SLC chip's firmware, and ultimately being able to send arbitrary, well-formed commands.

References

[1] Daniel Crimmens. What is a fire alarm system? [Online]. Available: https://realpars.com/fire-alarm-system/

[2] Texas Instruments. SWIFT™ Smart Wireless Integrated Fire Technology Manual. [Online]. Available: https://prod-edam.honeywell.com/content/dam/honeywell-edam/hbt/en-us/documents/manuals-and-guides/user-manuals/LS10036-000FL.pdf?download=false

[3] D. Lawrence, G. Kokinda, G. B. A. Lukman, Y. Kim, J. Smalligan, and C. Roberts, "Swift wireless fire alarm pull station analysis," Nov. 2021.

[4] D. Lawrence, G. Kokinda, G. Brown, D. Chou, Y. Kim, S. Wright, and C. Roberts, "Swift wireless fire alarm system analysis," May 2022.

[5] Douglas Krantz, "Make it work - addressable signaling line circuits," 2021.

[6] Darlington transistor. [Online]. Available: https://en.wikipedia.org/wiki/Darlington_transistor

[7] Texas Instruments. MSP430F543xA, MSP430F541xA Mixed-Signal Microcontrollers. [Online]. Available: https://www.ti.com/lit/ds/symlink/msp430f5419a.pdf?ts=1645687841030

[8] ——. Common Object File Format. [Online]. Available: https://www.ti.com/lit/an/spraao8/spraao8.pdf

[9] Pete Batard. (2011, Nov.) bin2coff. [Online]. Available: https://pete.akeo.ie/2011/11/bin2coff.html

[10] Texas Instruments. Bootloader (BSL) Scripter. [Online]. Available: https://www.ti.com/lit/ug/slau655g/slau655g.pdf