Mike Naberezny
EECS X497.2 Final Project, Winter 2017

VW Radio Volume Level Monitor

Introduction

Some background of my car repair tribulations is useful to understand the origin of this project. My EECS X497.2 final project description is at the bottom of this page.

In 2014, the radio in my 1999 VW Jetta broke. I bought a replacement on eBay for $15, but it did not come with the four-digit security code required to unlock it. The local VW dealerships all wanted $50 to look up the code. Rather than pay them more than the cost of 3 radios to give me this code, I wired up some relays to the radio's button contacts. I connected them to my PC via a LabJack USB digital I/O interface and brute force guessed the code. It worked, the radio unlocked, and I fixed the car.

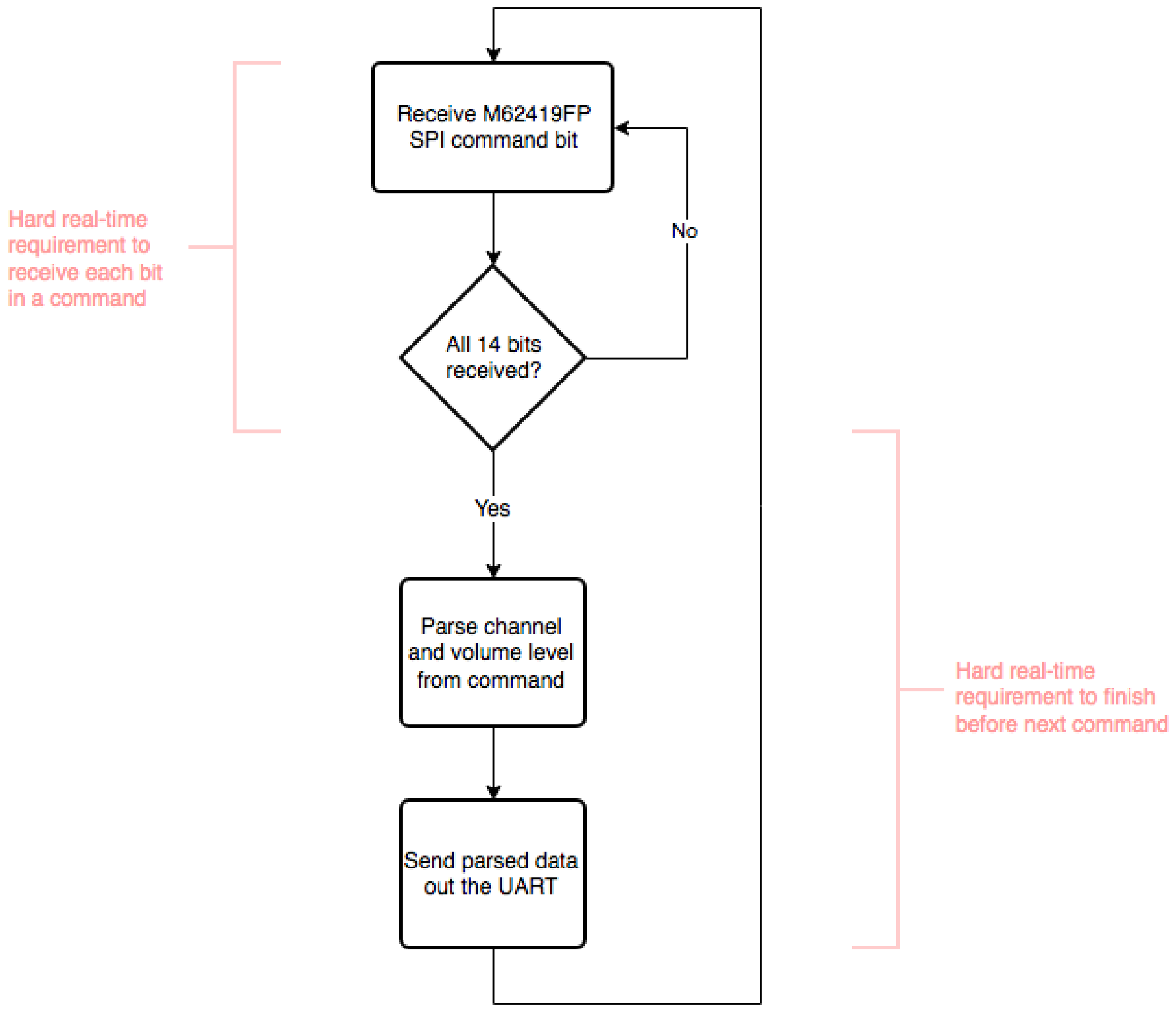
In 2016, the radio inside my 1998 VW Jetta broke. It has the same radio as my other car. I bought a replacement on eBay for $10 this time. It also did not come with the unlock code. With the realization that I would now be cracking a radio every couple years, I improved my setup. I ditched the relays and learned how to program an Atmel AVR microcontroller in C. I replaced the radio's faceplate with an AVR that emulated it and used it to find the code. It worked, the radio unlocked, and I fixed the car.

After the second radio repair, I bought another radio to prepare for round three. I kept this radio on my desk and expanded my project while learning more AVR C. The project is now at the point where I can control the radio from my PC via a serial cable. For example, I can send a command that will set the radio to FM 90.3 MHz. I can also write my own text on the LCD and read the buttons independently of the radio. The only thing I cannot control is volume. The AVR can change the volume up or down but it cannot read the current volume level. The AVR can only read the faceplate, and the radio never displays the volume level on the LCD. A way to read the radio's volume level is necessary for remote control.

For my EECS X497.2 final project, I developed a way to read the radio's volume level. I had no prior knowledge of how the radio's volume control worked, so I had to do that reverse engineering as part of the project. My project uses an Atmel AVR microcontroller to capture and decode SPI data between the radio's own microcontroller and one of its sound chips, the Mitsubishi M62419FP. I built it as a new, standalone project using AVR assembly language. This was challenging since I had no prior experience with AVR assembly language. The project was a success, and I plan to integrate it with my other work.

High Level Design

The project is an Atmel AVR microcontroller circuit that listens passively to the SPI clock and data lines of the Mitsubishi M62419FP sound controller. The AVR receives a command, decodes it, and sends portions of the decoded data out its UART transmit pin where it can be received by a PC.


Receive M62419FP SPI command bit
All 14 bits received?
No
Yes
Parse channel and volume level from command
Send parsed data out the UART
Hard real-time requirement to receive each bit in a command
Hard real-time requirement to finish before next command

The project presented two main challenges:

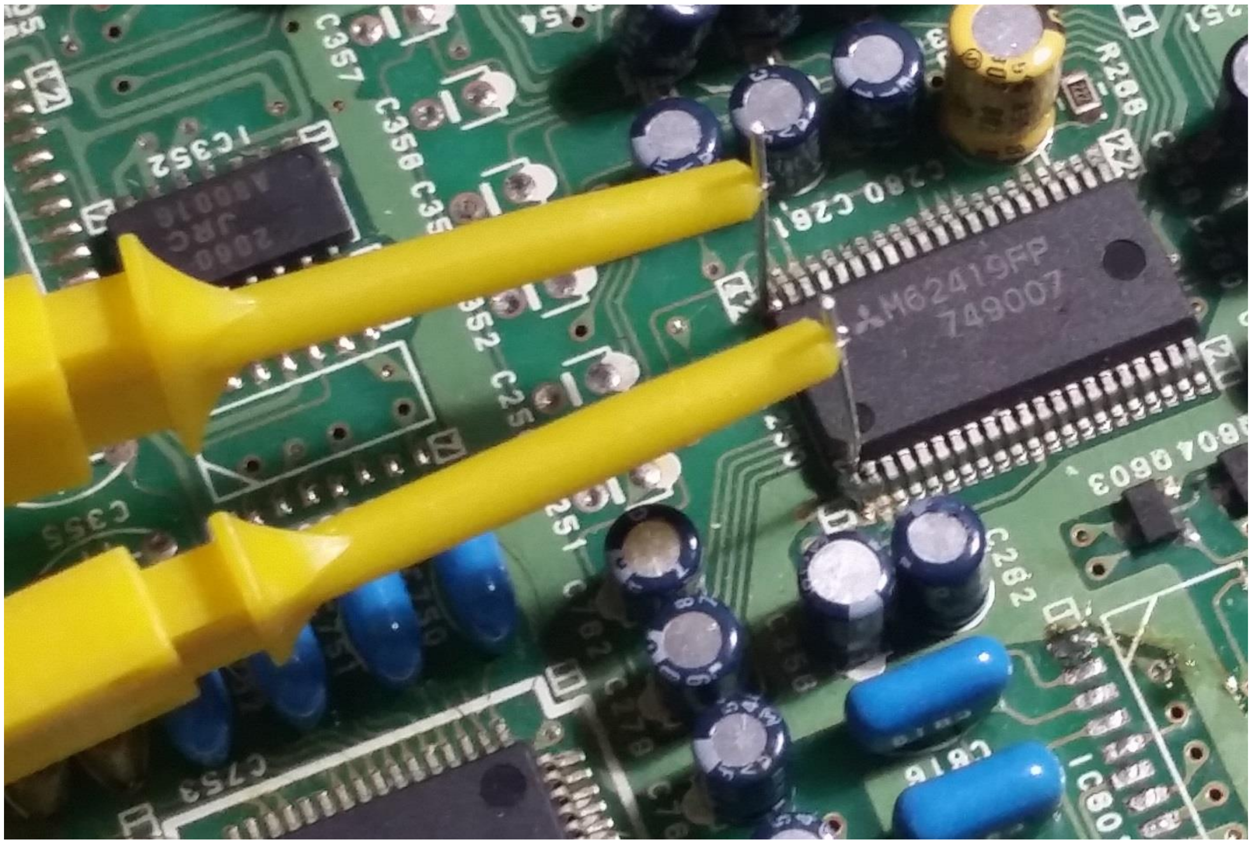
1. Reverse engineering the radio to determine how volume control works. This included identifying the M62419FP as the volume control and then understanding its specific details.

2. Developing AVR assembly language code that could passively listen to SPI commands sent to the M62419FP, while reliably meeting timing deadlines measured in microseconds.

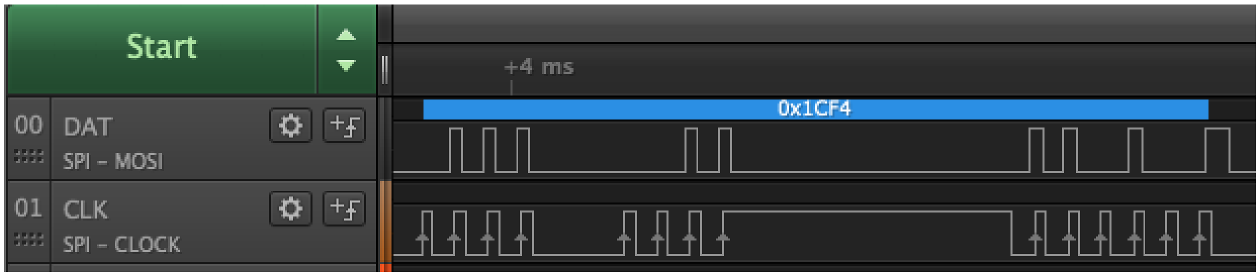
The above aspects are the focus of this report.

Initial Exploration

At the start of the project, I did not know anything about how the radio controls its volume. To find out, I took inventory of all the chips in the radio and found as many datasheets as possible. The M62419FP was the only chip featuring volume control. I learned that it is controlled by an SPI interface. I soldered resistor legs to the SPI clock and data pins to allow me to probe them. I connected a logic analyzer to capture the SPI signals and prove it was indeed the chip responsible.



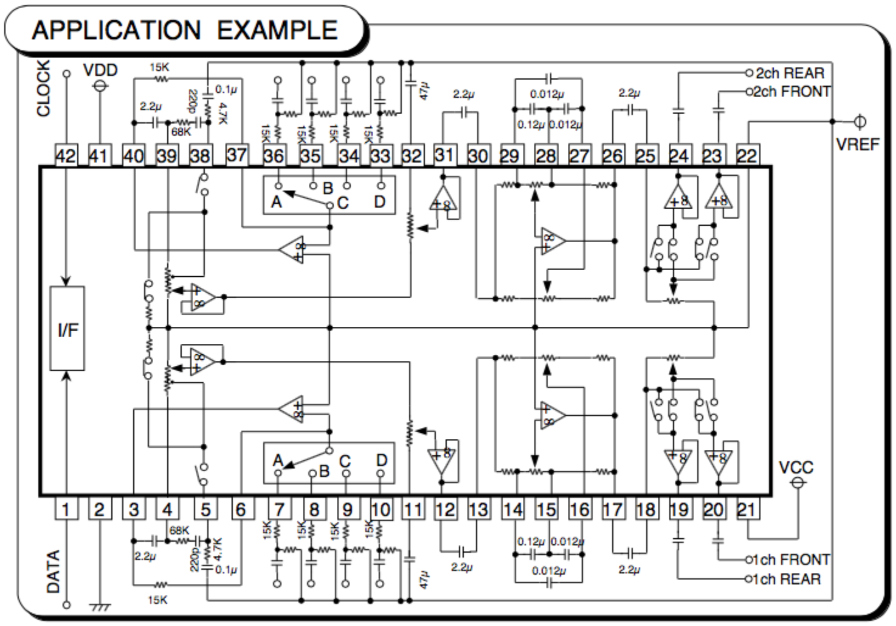
I adjusted the volume on the radio and saw corresponding activity on the M62419FP SPI lines. I knew I had found the right chip, but then I had to do more reverse engineering to understand the signals.



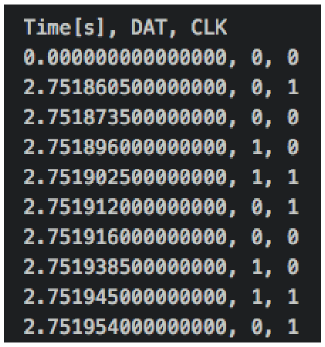
Reverse Engineering

I knew the chip pinout and SPI command format of the M62419FP from the datasheet. That was about all I knew.

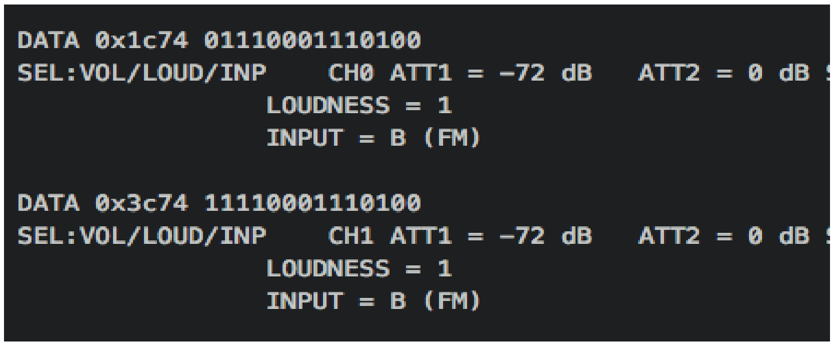
I did not know if the signals I was capturing made sense according to the datasheet. I also did not know how M62419FP was connected to the rest of the radio. Finally, I did not know how the adjustments I was making on the radio mapped to M62419FP commands.



I adjusted the volume on the radio faceplate while capturing the SPI lines on the M62419FP with a logic analyzer. I then exported the captures to CSV files and wrote a Python script to parse them into 14-bit packets of 0's and 1's. After comparing a few packets to the datasheet, they looked reasonable, so I improved my Python script to parse the command and display it in human readable form.



```
Time[s], DAT, CLK
0.000000000000000, 0, 0
2.751860500000000, 0, 1
2.751873500000000, 0, 0
2.751896000000000, 1, 0
2.751902500000000, 1, 1
2.751912000000000, 0, 1
2.751916000000000, 0, 0
2.751938500000000, 1, 0
2.751945000000000, 1, 1
2.751954000000000, 0, 1
```



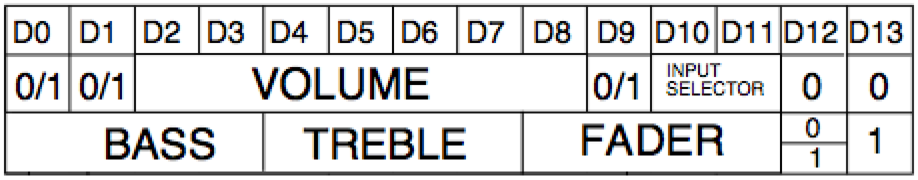
```
DATA 0x1c74 01110001110100
SEL:VOL/LOUD/INP    CH0 ATT1 = -72 dB   ATT2 = 0 dB
                    LOUDNESS = 1
                    INPUT = B (FM)

DATA 0x3c74 11110001110100
SEL:VOL/LOUD/INP    CH1 ATT1 = -72 dB   ATT2 = 0 dB
                    LOUDNESS = 1
                    INPUT = B (FM)
```

During this process, I had to develop an understanding of both the M62419FP registers and how the M62419FP is wired into the radio. To map out the input multiplexer, I selected different modes (AM, FM, CD, TAPE) on the radio and observed the commands to the M62419FP. To identify the left versus right channel, I set the audio balance all the way to the right and observed which channel was most attenuated in the M62419FP registers. I improved the Python decoder script iteratively as I learned.

It took quite a bit of playing with the radio, capturing the SPI commands, and iterating on my Python decoder. I eventually reached a point where the decoder was complete. I could capture and decode any M62419FP command using the logic analyzer and Python, and I understood what I was looking at. I was confident that I could then translate this knowledge into a working EECS X497.2 project.

AVR Investigation

The M62419FP is controlled by a 14-bit SPI transfer.

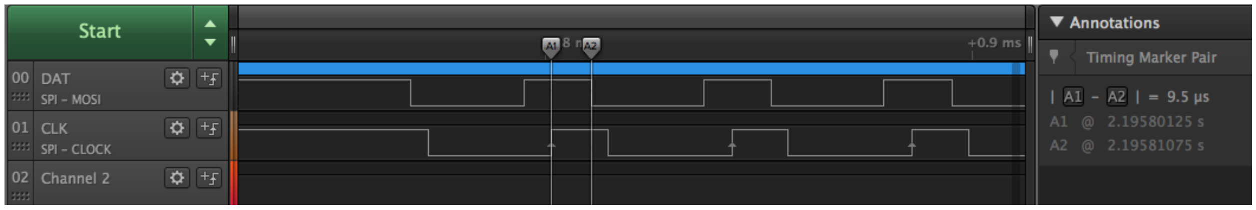


D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13
0/1	0/1	VOLUME							0/1	INPUT SELECTOR		0	0
BASS				TREBLE				FADER				0/1	1

I wanted to use an Atmel AVR microcontroller to capture this 14-bit M62419FP command. Many AVRs include a hardware SPI receiver but it is limited to 8 bits. Since 14 is not a multiple of 8, the hardware SPI of an AVR is not suitable for receiving the 14-bit command of the M62419FP. I had to determine if an AVR would be fast enough to collect a bit on every clock edge using software ("bit banging").

The SPI data bit is supposed to be latched by the M62419FP on the rising clock edge. The software approximation of this is to wait in a tight loop for that edge, then read the data bit as quickly as possible, and hope that the data bit is still valid.

To determine if this was feasible, I set up a logic analyzer to capture SPI commands from the radio's microcontroller to the M62419FP. I then measured the timing on the capture.



| A1 - A2 | = 9.5 µs
A1 @ 2.19580125 s
A2 @ 2.19581075 s

The logic analyzer capture above shows the time from a rising clock edge (when the data should be latched) to when the data bit from the radio is no longer valid. This time is not consistent, hinting that the radio's microcontroller most likely generates the signals in software. I looked at many edges and took the lowest time, which was 9.5 microseconds. To be reliable, the AVR must always sample the data bit before that deadline. I decided to take half the value for a safety margin, so the AVR would need to capture the data bit within 4.75 microseconds of the clock rising edge.

Many AVR microcontrollers can run up to 20 MHz clock frequency. Since time is critical, I went with 20 MHz. That gives a period of 0.05 microseconds, or 0.05 microseconds per cycle. Dividing 4.75 microseconds by 0.05, there are 95 cycles available while the data bit is valid.

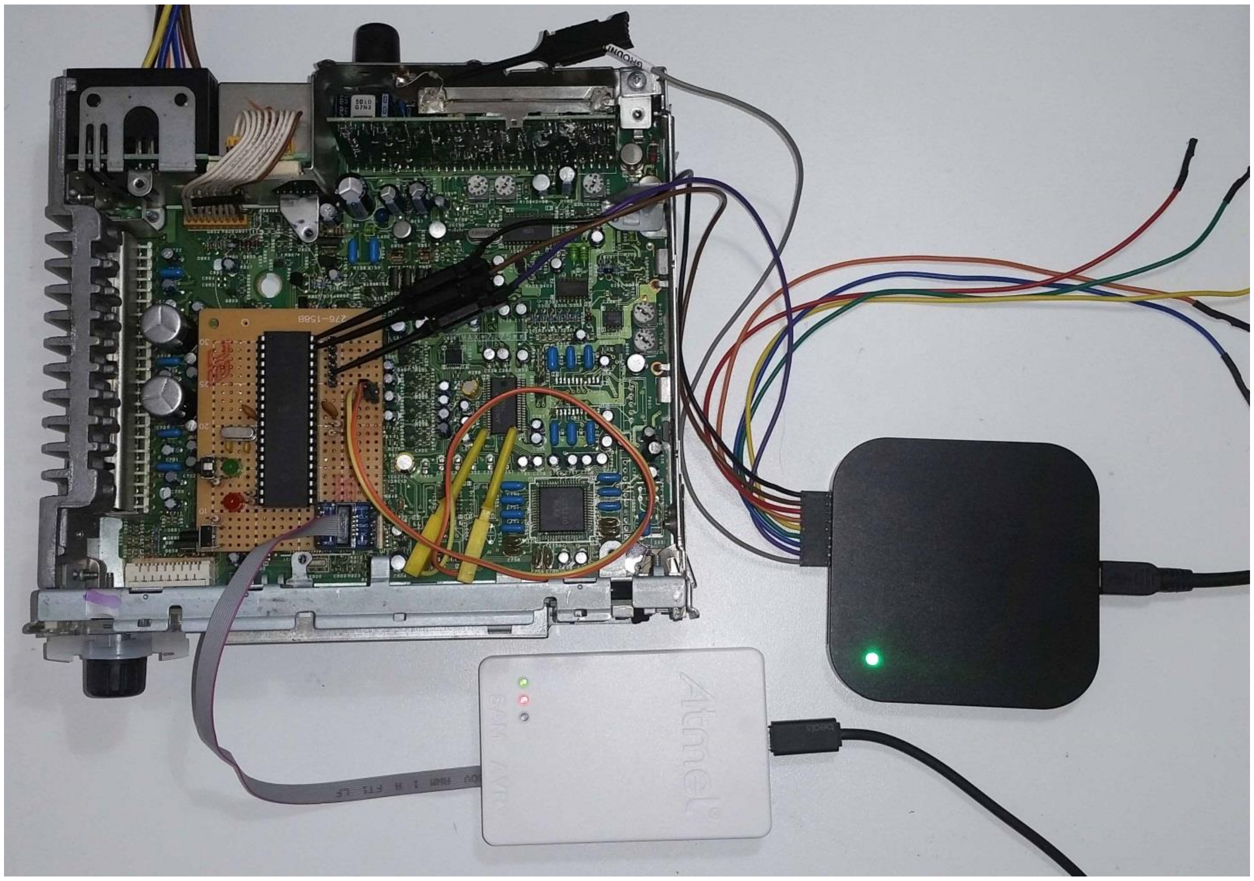
I estimated that I would need something along the lines of 25 instructions to wait for, sample, and store the data bit. Most AVR instructions are 1-2 cycles. I decided that 95 cycles would be enough time to do the work plus some other things with careful programming.

AVR Hardware

Since the entire point of my project is to produce real hardware to work with a real radio, I decided to build hardware early and program against that hardware instead of simulation.

I had previously worked with AVRs, but only programmed them in C. I had two hand-wired boards from my previous AVR projects. Both of them had an ATMega1284 running at 20 MHz, a reset button, two LEDs, a JTAG connector for the Atmel-ICE programmer/debugger, and a UART header for an FTDI cable.

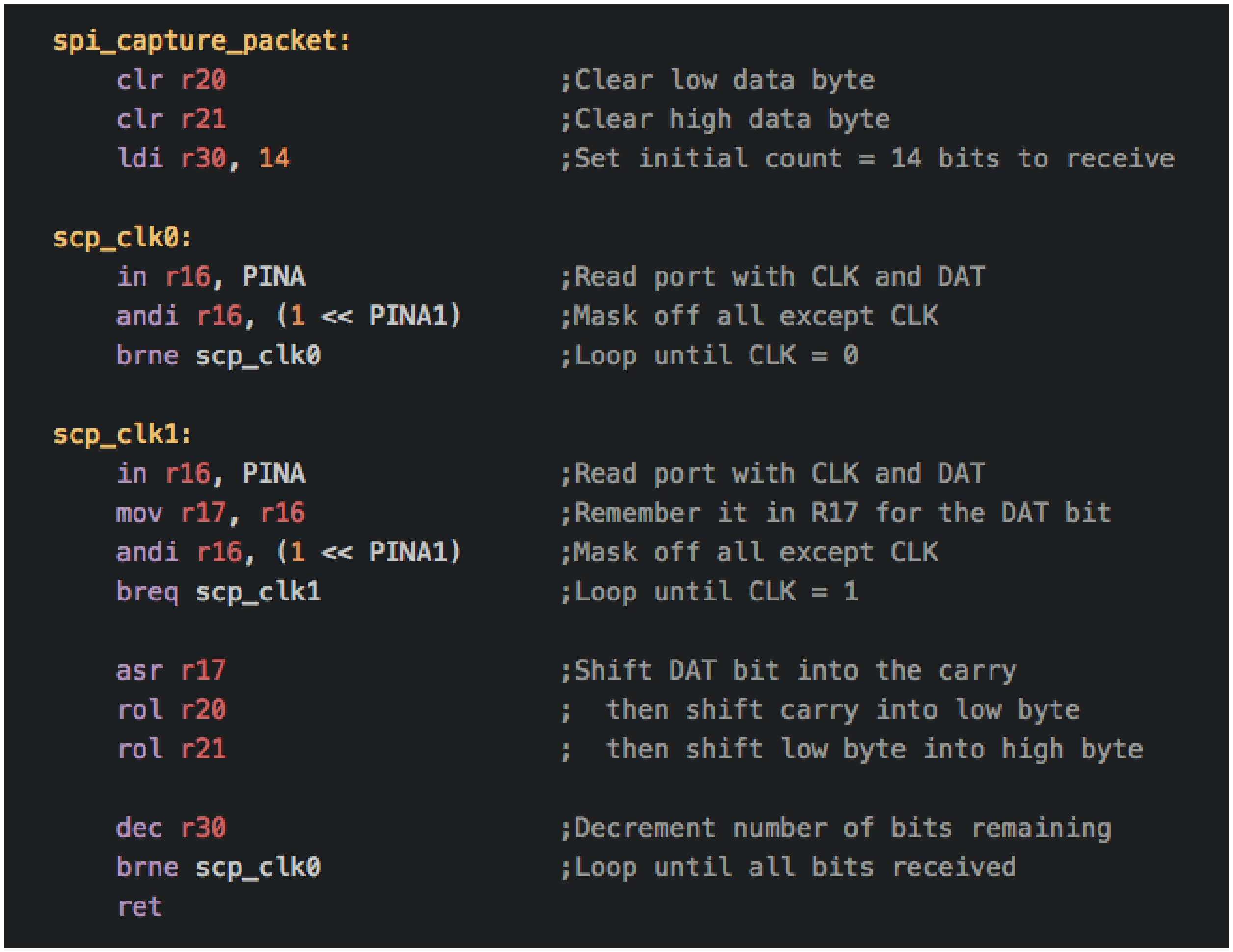
The M62419FP project only needed two I/O pins (SPI clock and data) and both my boards had two unused I/O pins. One of the boards had a free I/O pin on PA0. I thought it was desirable to have the data bit be on either bit 0 or bit 7 of a port so that one assembly instruction could rotate the bit directly into the carry flag. I decided to use the board with the free PA0 pin for this reason. I added a two-pin header for the M62419FP's clock and data. The clock ended up on PA1.



The photo above shows hardware used for the project. The board sitting inside the radio holds the Atmel ATMega1284. The two yellow clips connect the M62419FP clock and data lines to it. The white box is an Atmel-ICE connected to the JTAG port on the ATMega1284, which was used to flash the software. The black box is a Saleae logic analyzer that also connects to the M62419FP clock and data lines, along with the ATmega1284's UART transmit pin. The UART is also connected to a PC using an FTDI serial to USB cable (not shown).

AVR Software Iteration: Bit Banging SPI Receive via Polling Loop

With the hardware in place, I wrote the routine below and was able to receive M62419FP packets. As noted in the investigation, "bit banging" was necessary to receive the 14-bit SPI packets.



```
spi_capture_packet:
    clr r20                     ;Clear low data byte
    clr r21                     ;Clear high data byte
    ldi r30, 14                 ;Set initial count = 14 bits to receive

scp_clk0:
    in r16, PINA                ;Read port with CLK and DAT
    andi r16, (1 << PINA1)      ;Mask off all except CLK
    brne scp_clk0               ;Loop until CLK = 0

scp_clk1:
    in r16, PINA                ;Read port with CLK and DAT
    mov r17, r16                ;Remember it in R17 for the DAT bit
    andi r16, (1 << PINA1)      ;Mask off all except CLK
    breq scp_clk1               ;Loop until CLK = 1

    asr r17                     ;Shift DAT bit into the carry
    rol r20                     ;  then shift carry into low byte
    rol r21                     ;  then shift low byte into high byte

    dec r30                     ;Decrement number of bits remaining
    brne scp_clk0               ;Loop until all bits received
    ret
```

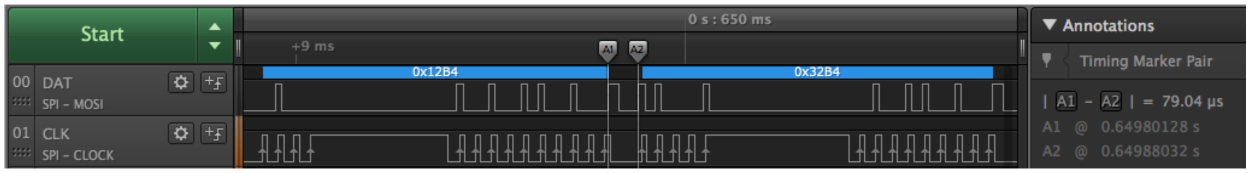
Receiving 14 bits of data requires 2 bytes of storage (16 bits). Registers R20 and R21 are used for this. The number of bits received must be counted so the routine knows when to stop. R30 stores the count.

The routine loops over "scp_clk0" and then over "scp_clk1" to detect a rising clock edge, indicating data valid. The data bit is shifted into the carry flag, then the carry is shifted into the low byte, then the low byte is shifted into the high byte. The routine repeats until 14 bits have been received.

Recall that only 95 cycles are available to capture a data bit. Ignoring the first 3 initialization instructions, there are 13 instructions looped over. Each instruction above is 1 cycle, except for branches taken which are 2. It's asynchronous to the radio, so sometimes it will loop on "scp_clk0" or "scp_clk1" more times than others, depending on when the clock changes. Still, it easily meets the cycle time requirements, assuming no interrupt code is running that would steal cycles.

AVR Software Iteration: Bit Banging SPI Receive via Pin Change Interrupt

This project is all about timing. By watching cycle counts, I was able to write the SPI bit bang routine on the previous page and it worked. I then hit a new timing constraint: the time between packets.

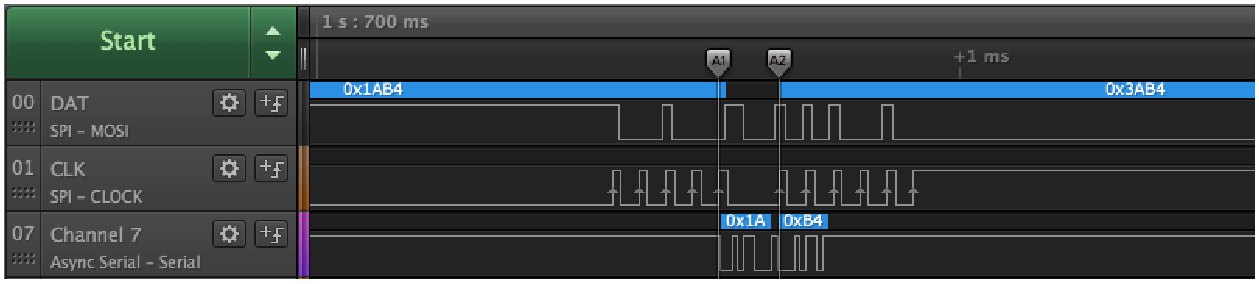


The capture above shows that only 79 microseconds are available between packets. With the polling method, the AVR must return to the polling loop before this deadline or it will miss the start of the next packet. When I used synchronous code to send the SPI packet data out the UART, I could not transmit it fast enough and it ran into the next SPI packet. To solve this, I rewrote the code to be interrupt-driven.

The AVRs can be programmed to jump to an interrupt service routine when a pin changes logic level. I set up an interrupt to happen when the SPI clock line changed. It used the same algorithm to collect the bits, but it happens asynchronously from the main program.

```
isr_pcint0:
    in r18, PINA                ;Sample port immediately (SREG unaffected)
    in r31, SREG                ;Save SREG
    mov r19, r18                ;Save original port value before we mask it

    andi r18, (1 << PINA1)      ;Mask off all except CLK
    breq isr_done               ;CLK=0?  Nothing to do; it was a falling edge.

    ;This is a rising edge and the DAT bit is valid.
    asr r19                     ;Shift DAT bit into the carry
    rol r23                     ;  then shift carry into low byte
    rol r24                     ;  then shift carry into high byte

    ;... remaining ISR code elided
```

By doing the bit banging via interrupt, new SPI data could be received in the background while processing old SPI data continued. The capture below shows the main program transmitting received SPI data out its UART while receiving the new SPI packet via interrupt at the same time.



AVR Software Iteration: Parsing SPI Commands

When receiving M62419FP commands was working reliably, the last piece of the project was to decode the volume data from each command received. I had already written a working Python script to do this from logic analyzer captures. This greatly aided the work and the development process was quite literally rewriting portions of my Python script in AVR assembly language. This work was mostly simple bit manipulation and value conversions. While it was time consuming, it is not particularly interesting, so I will not cover it in this report. Refer to the source code for more information. After the AVR receives an SPI packet, it parses the volume data from it, and then sends the parsed data out the UART.

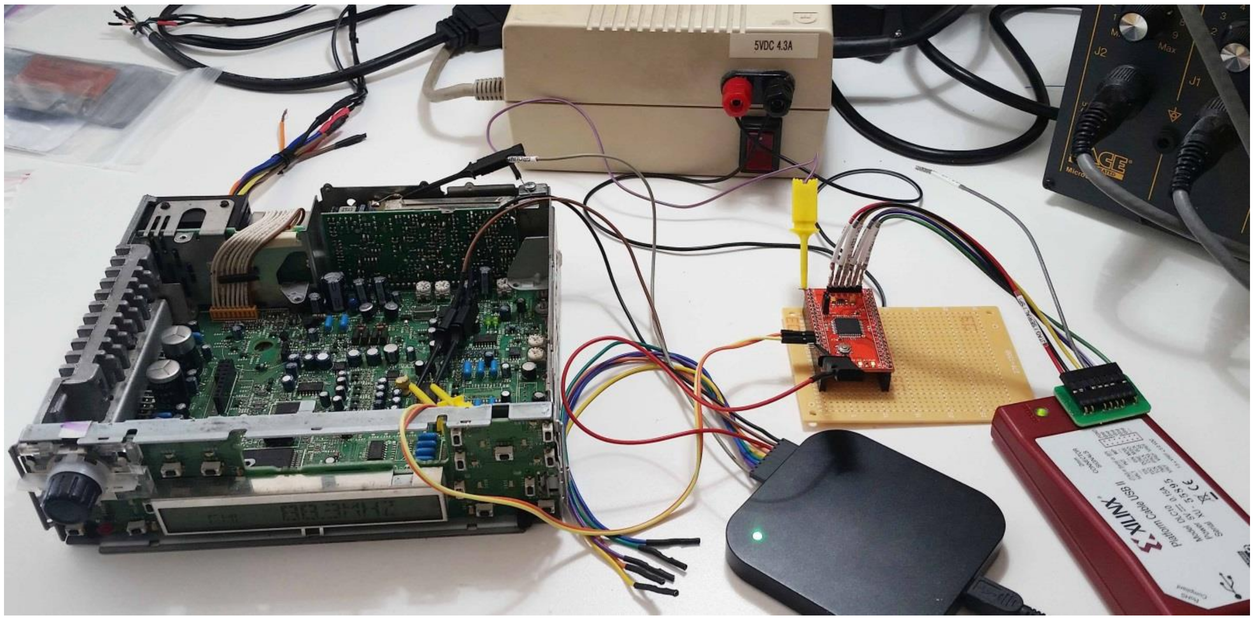
Conclusion

The VW radio discussed in this report does not display the current volume level on its LCD as the volume is changed. A method to read the radio's volume level is needed to add remote control capabilities to the radio. The radio's volume control mechanism was reverse engineered. An AVR assembly language project was developed to monitor the volume level. It captures SPI commands to the radio's Mitsubishi M62419FP sound processor in real time, decodes volume data from those commands, and transmits that information back out its UART. The project was challenging on many levels, particularly due to unknowns about the radio and significant timing constraints. The project was a success. Real hardware was built that can show the volume level as the volume knob is turned.

Future Work

This volume monitor was built as a new, standalone AVR assembly language project for the EECS X497.2 course. Prior to the course, I had been working on a separate project to remotely control the radio using AVR C. That project can control the faceplate (LCD and buttons) only. I plan to combine the two projects after the course to achieve complete remote control of the radio.

Problem Encountered: VHDL Project



I actually did the final project 1.5 times because I started it in VHDL. Since the project is basically a fancy shift register, I thought it would be perfect for hardware. The photo above is from that first effort.

Hardware Selection

I wanted real hardware instead of simulation. A small, inexpensive FPGA or CPLD board was desirable so I could add it to my existing radio project. It would ideally be 5V tolerant. Being new to programmable logic, I wanted one with a lot of tutorials online. I settled on a Xilinx XC9572XL CPLD breakout board.

Debugging Experience

When I coded the first shift register in VHDL and programmed the CPLD, I used a logic output to indicate when the 14th bit had been received. I observed the M62419FP's clock, data, and my logic output to see that it was capturing the commands as expected. My output pin seemed random. I checked the wiring twice and it was correct. I tried various code changes with no effect. Finally, I decided re-check my assumptions by changing the VHDL so my logic output just mirrored the clock input. I tried again, and it did not. It mirrored the data input! The wiring was correct but I had accidentally reversed the signals in the CPLD pin assignments (UCF file).

CPLD Ran Out of Space

VHDL tutorials I found online for the XC9572XL looked to be similar complexity to my project, so I thought this chip would work. With the receive shift register and only some basic munging of the data, the CPLD was at 75% of the available macrocells. As I began to add more functionality, it became obvious that I would not have enough space. The XC95144XL probably would have worked but I could not find a breakout board for it on time. I then thought to use the XC9572XL for the receive shift register only, working cooperatively with an AVR to do the rest. That led me to investigate whether doing it with the AVR alone would be possible, and it was, so I switched to an AVR assembly project.

Problem Encountered: M62419FP Datasheet Unclear

Details about the M62419FP were not clear from the datasheet, such as the volume attenuators. Each audio channel has two attenuators in the M62419FP to control the volume. I did not know why one channel had two attenuators or the relationship between them. To work this out, I turned the volume knob on the radio all the way to minimum, started capturing with the logic analyzer, and then turned the volume knob all the way up. This captured all the commands setting the volume from lowest to highest. I then wrote a Python script to read the logic analyzer capture and display the attenuator values across the whole range. For each click of the volume knob, I noticed that sometimes one attenuator changed, sometimes the other changed, and sometimes both changed. This did not make much sense initially until I started playing and displayed the sum of the two attenuator values. The sum always increased by an exact increment of decibels. From this, I was able to deduce that the two attenuators are additive: one is a course step and one is a fine step.

Problem Encountered: M62419FP Synchronization

The M62419FP has SPI clock and data lines only. It does not have a chip select line. In addition to all the timing issues discussed in this report, there is an issue of synchronization. If a chip select line had existed, it would have provided a way to continuously ensure synchronization: after the edge where chip select becomes active, the shift register will always receive bit 0. Since the line does not exist, there is no way to identify the start or end of a 14-bit packet. This means that my project must never miss an SPI clock edge, or it will be forever out of sync with the data stream. This required careful programming. To validate reliability, I continuously turned the volume knob back and forth while varying turn speed for over a minute. At the end, I compared the commands reported by the logic analyzer and the AVR. There was no difference, indicating that the AVR stayed in sync after a large number of commands.

Problem Encountered: Unused Bass and Treble Registers

The radio faceplate has volume, bass, and treble controls. The M62419FP has attenuation (volume), bass, and treble registers. I surmised that adjusting the sound on the radio faceplate would send corresponding commands to the M62419FP. Using the logic analyzer, I found that volume worked as expected but I saw no activity when bass or treble were adjusted. After repeated attempts, I reviewed the datasheets for all the chips in the radio again. I determined that another chip provides a more advanced sound equalizer and it is actually what is used to adjust bass and treble. The bass and treble registers of the M62419FP are simply unused (always 0, or flat sound). The radio only uses the M62419FP for channel multiplexing and volume control. This did not impact the project, since I only intended to capture volume data anyway. Bass and treble levels are displayed on the radio's LCD.

Problem Encountered: Missing ATMega1284 Support in AVRA

I needed an assembler. I found that Atmel Studio was several gigabytes in size and using it was sluggish. I wanted to just use my regular text editor. I also wanted to do the work on macOS. I found the open source AVRA assembler worked on macOS. I had some ATMega1284 chips already and wanted to use those chips. AVRA supported many different AVR chips but not the ATMega1284. I downloaded the C source code for AVRA and then used "grep" to search for the name of an AVR that AVRA did support. I found a file "device.c" that had a table of all supported devices and their parameters like memory sizes. I added the ATMega1284 to the table using info from the datasheet and then compiled my own version of AVRA. This worked but then I found I still could not write ATMega1284 programs with AVRA because I did not have a definitions file to ".include" for it. To solve this, I temporarily installed Atmel Studio on a VM and then grabbed the definitions file from it. After some editing, it worked with AVRA. After all that, I was finally able to build an ATMega1284 project on macOS using AVRA. I wrote a very simple program and assembled it on both Atmel Studio and AVRA. The hex files were identical so at that point I blew away Atmel Studio and used AVRA for the rest of the project. I used another open source program, AVRDUDE, to program the target.

Problem Encountered: Unfamiliar with AVR Instruction Set

I had no experience with AVR assembly language before this project, yet this project required me to do cycle time estimations and write efficient code. I often wrote inefficient code because I did not know any better. An example of this can be found in the routine “scp_clk1” of the polling loop version of the SPI receiver in this report. In that routine, I needed to test a bit but also needed to preserve the original byte. I saved the byte in a temporary register, used ANDI to mask, and then conditionally branched with BRNE. The only reason I did the bit test this way is that I had not discovered the “Skip if Bit in Register” instructions SBRC and SBRS. I was frequently referring to the AVR Instruction Set Manual and rewriting code as the project progressed. There is very likely still room for improvement. Thankfully, the timing margins were enough to forgive the inefficiencies.

Problem Encountered: No Edge Detection for AVR Pin Change Interrupts

After switching from polling to an interrupt-driven SPI receiver, I wanted an interrupt on every rising edge of the SPI clock line. The ATMega1284 provides hardware rising edge detection with the INTx (External Interrupt) pins but not with the PCINTx (Pin Change Interrupt) pins. I had already wired the clock line to a PCINTx pin. The PCINTx interrupts always fire for any change in level. Since the clock line holds its level for microseconds after changing, I was able to detect which edge by reading the pin in software at the beginning of the interrupt service routine. It would have been more efficient to move the SPI clock line to one of the INTx pins and use the hardware edge detection. The ATMega1284 board I was reusing already had those pins wired for other purposes so I decided not to do this.

Resources

The following documentation was used for this project:

- Mitsubishi M62419FP Datasheet
- Atmel ATMega1284 Datasheet
- Atmel AVR Instruction Set Manual
- Atmel Application Note AVR108: Setup and Use of the LPM Instruction
- My own notes on the radio developed from reverse engineering

The following hardware was used:

- My own homemade ATMega1284 board
- Saleae Logic-16 logic analyzer
- Atmel-ICE programmer/debugger
- Dangerous Prototypes Xilinx XC9572XL Breakout Board (abandoned)
- Xilinx Platform Cable USB II (abandoned)
- Computer, soldering iron, coffee maker, and other common shop tools

The following software tools were used:

- AVRA assembler, with my patch for ATMega1284 support
- AVRDUDE uploader/downloader, to flash the target via the Atmel-ICE
- Atmel Studio, for about 10 minutes to get the ATMega1284 assembly definitions file
- Xilinx ISE Webpack IDE and Xilinx iMPACT, for the CPLD development (abandoned)
- Python, to write the script that analyzed the logic analyzer captures
- macOS, for a Unix-like command line and tools like vim and grep

The following existing source code was used:

- My own C code for setting up the ATMega1284's UART, which I compiled and ran through the avr-objdump tool to see the generated assembly language, which allowed me to get all the register settings I needed to set N-8-1 and 115200 bps for the 20 MHz clock

Non-Resources

The following things would have been great but I did not have them:

- Schematic diagram of the Clarion PU-1666A radio
- More detailed Mitsubishi M62419FP datasheet
- Chip select pin on the M62419FP that could be used to resynchronize SPI receive state
- AVR with hardware SPI capable of 14-bit receive
- Larger capacity CPLD or FPGA on-hand so I could have kept going with VHDL
- VW dealership that looks up radio codes for free