jhart99

AUCTUS A6: The hidden interface

Radio 2021-10-18 1316 words 7 minutes

> In addition to the normal buttons, knobs and screens used in the AUCTUS A6 DMR radios, there exist a second and third interface for these radios. One is used by the CPS software to read and write the codeplug to the radio. But there is another interface where the radio can be controlled through AT commands.

Introduction

The AUCTUS A6 based radios include COTRE and GOCOM models with varying interfaces. While the normal programming interface is nice enough, it would be great if it could be programmed using QDMR. To get this working, it would be nice to understand what is going on at the beginning of programming as there are a lot of commands being sent back and forth prior to the upload or download of the codeplug.

Not only are these commands sent back and forth, but the commands have specific timing requirements. Sending the commands too quickly causes the radio to crash.

Luckily, we have the firmware to guide us as to the nature of these commands and in the process revealed another hidden interface to the radio.

RDA development tools

As mentioned previously, the AUCTUS A6 seems to be derived from the RDA8809 processor and luckily there are several sources of information for these processors. AI-thinker, the group behind the very popular ESP8266 modules, have another module called A9 or A9G which use the related RDA8955 processor. There is an SDK and set of development tools available on Github.

The CPS cycle

We can capture the serial communication between the CPS software and the radio using Wireshark and UsbPCAP. These show that before the CPS software reads anything, it issues over 100 commands to the radio.

Not only does it issue a ton of commands, but there is particular delays between commands that are essential. If the commands come too fast, the radio will lock up and can only be reset by pulling the battery.

All of the commands use the RDA debug protocol discussed previously in making the radiodump utility. To keep things simple, just realize that everything is formatted into “ad” frames but I’ll just summarize the commands by their function.

First command

Write a byte (0x80) to register 03.

The documentation for this register is in coolwatcher which comes with the development tools. Coolwatcher has the definitions for all of the RDA8809 registers. These registers are different from the general purpose MIPS registers. They are memory mapped registers and are the primary means that the A6 interacts with its different subsystems. Everything is mapped to a memory address, the RF system, the UARTs, the timers.

In this case register 03 is the INT_REG_DBG_HOST register (0x00000000) and the (0x00000003) register is CMD. Setting this to 0x80 ensures that the UART is locked to debug mode.

Hidden gems

After this, the CPS reads a couple of locations in RAM. Specifically 0x81c00270 which points to a location in RAM where commands will be sent back and forth. However, looking at the neighboring memory, there are many other useful pointers here.

address	function	comments
0x81c00264	ATECPS responses pointer	Where to read responses from commands
0x81c00268	CPS pointer	Where CPS is made available to download from the radio
0x81c0026c	CPS length struct pointer	The length of the codeplug header and contents
0x81c00270	ATECPS command pointer	Where to write command to the radio

In addition to a pointer in memory where the commands are written, there is a second location where responses to the commands are made available.

The ‘Magic’ Commands

Following this, there are many writes to the ATECPS command pointer which have a write to register 05 both before and afterward. Preceding the command is a write of 0x00 to register 05, the command is written to memory and followed by a write of 0xa5 to register 05.

Using the information from coolwatcher, we find that register 0x00000005 is H2P_STATUS. The description is that writing to this register makes a value available to the processor in the APB register named STATUS. This corresponds to memory address 0x01a3f010 and will kick off an interupt when changed.

Now digging through the firmware, we can find the function which processes the H2P Status. The function looks for one of 4 values in the H2P register:

command	meaning
0x00	Clear command and response areas
0xa5	Process the command and write the response
0xee	reboot radio
0xff	Unknown

The commands

Between the writes to the 05 register, a command, like the following, is written to memory.


```
aa 06 0a 06 0a bb 00 00
```

These commands are actually a second protocol encapsulated inside of the RDA debug packets.

```
aa XX CC CC PP... ZZ bb 00 00 00 ...
```

In this protocol aa and bb are the start and end of frame markers. XX is the length of the frame including the start and stop. CC CC is a two byte command and PP is 0 or more bytes of parameters that go along with the command. ZZ is the xor of command and parameters used as a check of frame integrity. Following the bb, there are multiple 00s added as padding to make the whole frame a multiple of 4 bytes in length.

The timing

In the captures, the CPS software leaves 70ms between the different writes involved in these magic commands. This makes sense as it gives the processor in the radio enough time to read the command and run it. Too fast and we send a second command while the first is still being executed.

The Commands

It is possible to read out the commands from the firmware function that parses the commands, but the CPS software only uses 8 commands.

Command	Description
002b	Checks the programming password
0a00	Concatenates the CPS information into RAM to be downloaded
0a01	Unknown
0a02	Unknown
0a03	Unknown
0a04	Unknown
0a05	Unknown
0a06	Unknown
0a07	Unknown
0a08	PROT_SendDataFrame Seems to echo the sent packet back

In addition to these there are many more commands. Some have pretty obvious functions but others are unknown and this includes most of the commands that the CPS software uses.

The functions seem to have error checks built into them. There are functions which write to the flash in the radio and so it is possible that a malformed command could result in a bricked radio. Be careful.

The hidden AT Commands

After finding the function that processes the aa-bb style commands, I noticed that the same function was looking for another sequence “AT+…\r”. Those sequences are normally AT commands and you find them all over the place for example with modems and bluetooth modules and even some radio modules such as the DRA818. There are about 100 of these commands.

To use these commands, you send a command such as “AT+DMOCONNECT\r” to the same address as those aa…bb type commands. The response can be read from the same region.

```
$ python3 atcommander.py AT+DMOCONNECT
OnCmd_DMOCONNECT
ATE_SendCmdAck: cmd:DMOCONNECT isOk:0x1
tx_length:17
ATE_SendDataFrame:
+DMOCONNECT:0

ATE pipe. used[HOST]
```

atcommander

I made a quick script atcommander that can interact with this command interface, sending commands and receiving the responses.

Conclusion

This started off as a way to figure out what is needed to program the radio with a 3rd party CPS software. The commands sent by the CPS software to the radio have been partially solved and this brings the COTRE radio a bit closer to that goal.

Additionally, it looks like these radios can be controlled using the serial port. This opens up new and interesting ways to use these radios. The frequency and most mode setting can be made from a computer. I think that these radios can be used like a radio module with a small computer such as a Raspberry Pi Zero driving them. There would no longer be issues with limited channels, no display, limited space for RadioID, etc. All of the fancy functionality could be pushed out of the radio and into an external frontend.