

Shining some light on the Amazon Dash button

Hunz <itsec@firlefa.nz>

28.12.2016

33C3

The Dashbutton

- available in the US since 2014
- in Germany since August 2016
- 2 hardware-revisions
- this talk: rev. 2



What is it, what does it do?

- **wifi-connected button** tied to your amazon-account
- button can be used to reorder certain consumables
- only available for certain brands/products
- costs 5€ with refund upon first button-triggered order
- customizable button for \$20 available - uses AWS

What's interesting about it?

- it has wifi
- sort of internet-of-shit device
- **how does it work?**

- **what about security?**
- security risk to put it in our networks?
- can it be (ab)used for **CYBER**?

- can it be repurposed for custom IoT-projects?
- more powerful than ESP8266, comparable price
- **if we cannot run code on it we don't own it**



CYBER

- old button:
`https://mpetroff.net/2015/05/amazon-dash-button-teardown/`
- new button: `https://mpetroff.net/2016/07/new-amazon-dash-button-teardown-jk291p/`
`[mpetroff]`
- audio protocol: `http://www.blog.jay-greco.com/wp/?p=116` `[jaygreco]`

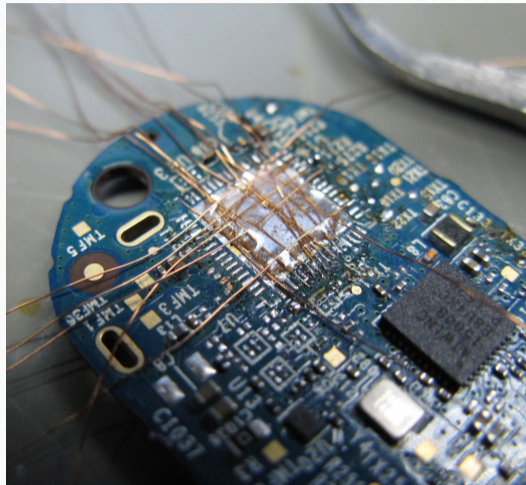
Repurposing the Dash the easy way

- Amazon smartphone app used to configure the dash
- last step of configuration is choosing a product
- aborting here prevents the dash from ordering
- product selection stored server-side
- dash stores wifi config nevertheless
- button contacts server, server says nope, button blinks red
- upon press button does 802.11 probe, auth, association, DHCP, ARP and DNS
- monitoring DHCP logfile with custom hook is easy

In this talk

- hardware
- communication protocols & crypto
- firmware (version: 19.4.10 Svnrev 12577)
- running custom code on the button

I didn't analyze the amazon smartphone apps



Hardware

Opening the Dash

- housing is **heat-sealed plastics**
- opening without damage is non-trivial
- 1st attempt with knife destroyed some SMD-components
- carefully applying a cutting-wheel seems to be best option



What's in there?

PCB: 4 layers, SMD 0201 parts

Microcontroller: Atmel ATSAMG55J19

- **120MHz ARM Cortex-M4** with FPU
- 512 kBytes Flash, 160 kBytes RAM
- QFN64 package with black stuff around the pads
- black stuff can be softened with acetone
- remove carefully - SMD-components underneath

Wifi-IC: Atmel ATWINC1500B

- 2.4GHz, up to 72 MBps, WPA(2), etc.
- **builtin IP-stack** with DHCP, DNS, SSL, etc.



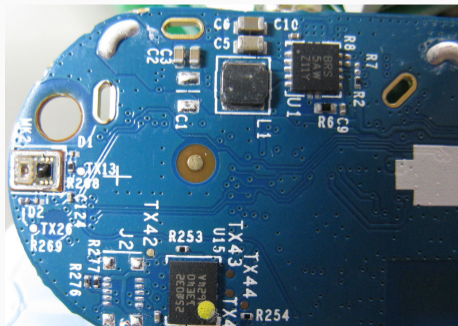
What's in there? (2)

3.3V voltage-regulator: TI TPS61201DRC

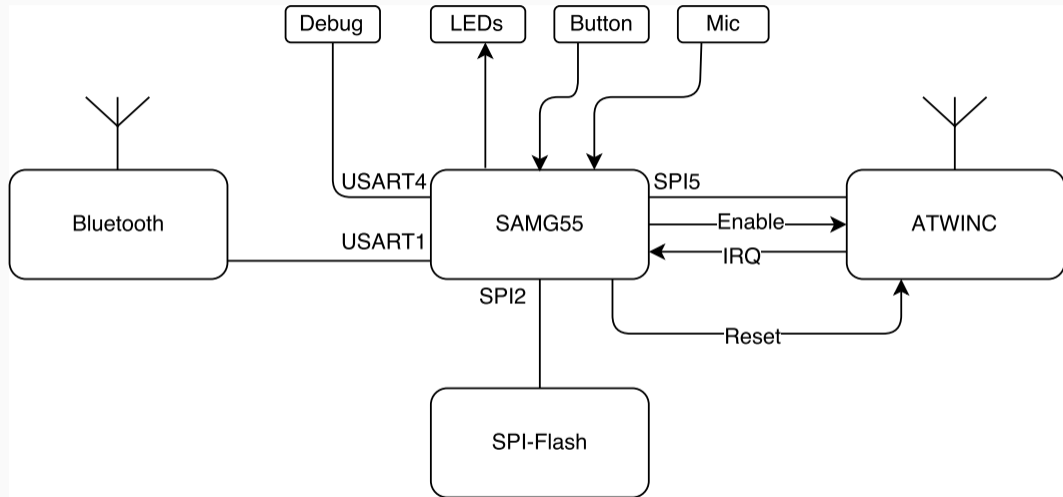
- boost- & downconversion-mode
- 0.3V .. 5.5V operating range

Other parts:

- **Bluetooth Low-Energy:** Cypress CYBL10563-68FNXIT
- **4MByte SPI-Flash:** Micron N25Q032
- MEMS microphone (SiSonic PDM?)
- RGB LED
- 32kHz oscillator
- some discrete semiconductors
- AAA battery: $\leq 1.5V$



Putting the pieces together



Overview of Dash components

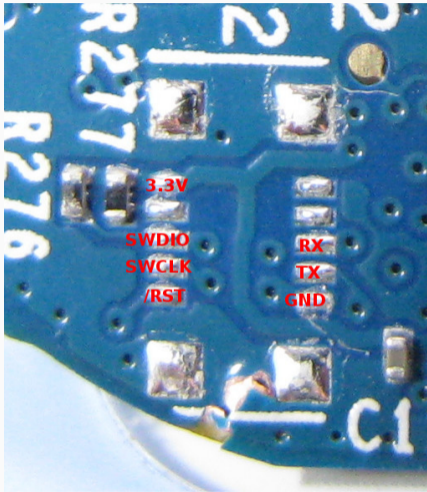
Power supply

- voltage regulator started by button-press
- there is **NO** other wakeup-source - no RTC, etc.
→ **button can never wake up on its own terms**
- power enable is held by external latch
- MCU clears latch for shutdown
- MCU can measure battery voltage using ADC
- enable-signal connects battery to ADC

Power consumption

- [mpetroff] already did some measurements of power consumption
- wifi draws a lot of power (roughly 0.4W)
- MCU working but no wifi: roughly 0.08W
- with wifi disabled and heavy MCU powersaving $<0.05W$ might be possible
- builtin AAA battery holds about 0.5Wh
 - **about 75 wifi-minutes**
 - **about 10h given a 0.05W consumption**

Debugging interfaces (bottom side)



Note: all IOs are 3.3V

UART commands

DEV MODE MENU

=====

```
> Add
> wifiscan
> wificonnect
> httpclient
> led
> shutdown
> order
> register
> deregister
> fw_update
> sendmetrics
> resetmetrics
> reset
> config
> apmode
> sonic
> immortal
> mortal
> fwver
> vbatt
> chiplock
> chiplockstatus
> switchlog
> help
> menu
> exit
> userdata
> bleboot
> bleVer
> pwrtable
```

USER MODE MENU

=====

```
> fwver
> bleVer
> macr
> chiplockstatus
> vbatt
> reset
> shutdown
> exit
> immortal
> mortal
```

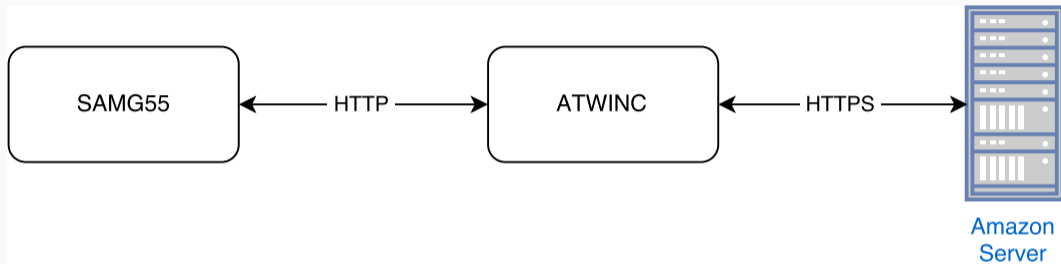
TEST MODE MENU

=====

```
> ate_wl_example
> ate_wl_txtest
> ate_wl_rxtest
> ate_wl_txstart
> ate_wl_rxstart
> ate_wl_txstop
> ate_wl_rxstop
> ate_wl_txstatus
> ate_wl_rxstatus
> ate_wl_setrxchan
...
```

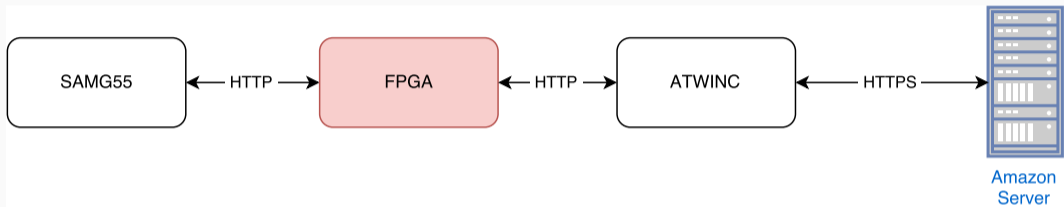

Communication protocols & crypto

Analyzing the communication



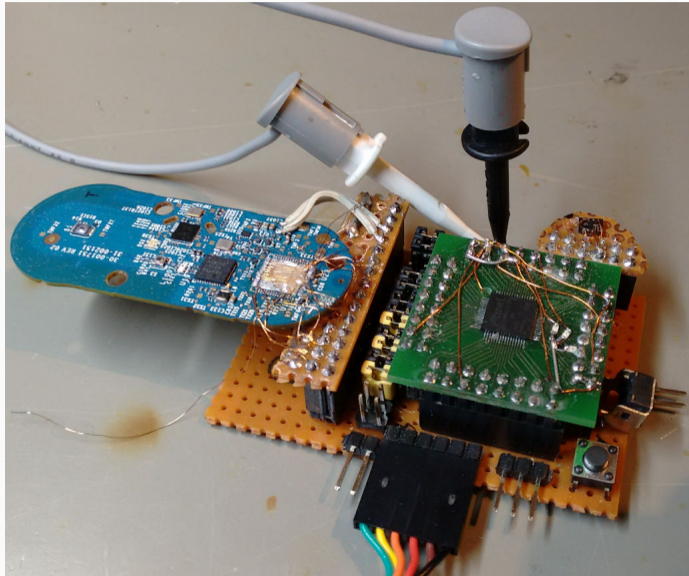
- SPI bus between ARM and ATWINC clocked at 40 MHz
- carries plaintext data
- TLS during communication with amazon server is done by ATWINC

Analyzing the communication (2)

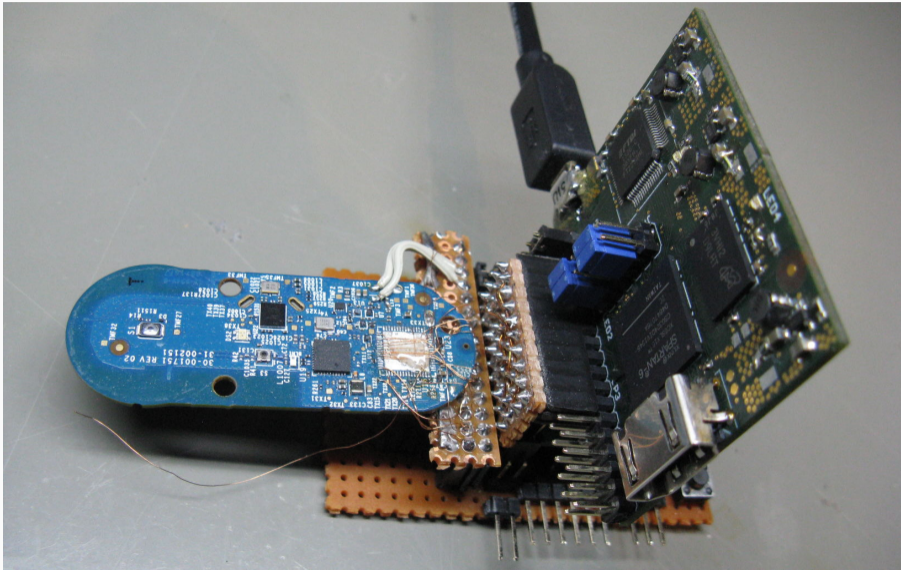


- I sniffed the communication between the ARM and ATWINC
- FPGA was used to allow for man-in-the-middle experiments
SAMG55 is SPI master, drives the clock - timing is challenging
- I did this before I had the full dash firmware

Accessing ALL THE SIGNALS



Analyzing the communication - FPGA board




Wifi-based configuration

- Android Amazon app uses this
- pressing button for a few seconds enables configuration mode
- button goes into AP mode (SSID: Amazon ConfigureMe)
DHCP server for IP assignment
- simple HTTP server running on SAMG55
- webpage with basic info

Amazon Dash™

Device Info

Serial Number	XXXXXXXXXXXXXXXXXX
MAC Address	
Firmware	30017420_EU
Battery	81

Configuration with Android Amazon app

app does the following:

1. fetch device info with GET /
(Content-Type: application/json)
2. post own ECDH pubkey to **/pubkey**
3. read dash pubkey from **/pubkey** with GET
4. post locale config to **/locale**
5. post encrypted stoken to **/stoken**
6. post encrypted network config to **/network**

dash then connects to wifi, registers with amazon server & obtains customer secret

Crypto details

- **device secret**: 20 chars uppercase + digits - written to flash during production
- **customer secret**: 20 bytes - obtained from amazon server after configuration
- both of these stored in flash, used for HMAC on requests

- **ECDH** (during config) uses prime256v1 curve
- temporary symmetric encryption for token and network data uses **AES-GCM**
- temporary symmetric key derived from ECDH data using SHA256
- AES-GCM data is TLV-encoded
(16bit len, type 1: IV, type 2: tag, type 0: ciphertext)
- plaintext data (pubkeys, token, network, locale) is JSON-encoded
- I heard you like parsers, so I put encrypted JSON into TLV data :-)

Example data

```
pubkey: {"publicKey":"-----BEGIN PUBLIC KEY-----\nMFkwEwYHK<...>  
f6YAIg==\n-----END PUBLIC KEY-----\n","scheme":0}  
locale: {"cc":"DE","realm":"DEAmazon"}  
token: {"expiry":1477282311,"token":";o}-"}  
network: {"priority":0,"psk":"test123","keyMgmt":"WPA_PSK",  
"ssid":"Cyber"}
```

Note:

- unused HTTP location: **/flash**
- seems to allow flash-access
- looks like authentication is needed
- haven't had a closer look at this

Registration with amazon server

- done by button after config with app
- POST to `/2/r/oft?countryCode=XX&realm=XXAmazon`

body:

```
fe XX 01 ca 07 8c      XX: battery level
G023232323232323      device serial number
03 01 00 00           transaction counter (1e32)
00 00 00 00 00 00 00 00
22 4a 55 3a           token from app
<HMAC using device-secret> (20 bytes)
```

response:

```
00 00 00 00 00 00      flags?
58 0f dd 20            unix date
<customer-secret> (20 bytes)
60 38 a7 31           ???
```

onButtonPressed()

- two POST requests to `parker-gw-eu.amazon.com`
- `Content-type:binary/rio`, chunked encoding
- POST to `/2/b`: actual order request
- POST to `/2/d`: debugging info (metrics)

if server demands a firmware update:

- additional POST to `/2/f` and firmware download

POST to /2/b

chunk 1:

fe XX 01 ca 07 8c

XX: battery level

G023232323232323

device serial number

06 01 00 00

transaction counter (le32)

00 00 00 00 00 00 00 00

chunk 2:

31

chunk 3:

<HMAC using customer-secret> (20 bytes)

POST to /2/b - response

- HTTP status code used for feedback to button
- e.g. 200 for order successful (LED green),
412 with no product selected (LED blinking red)
- body contains server timestamp (binary)
- flag for firmware update request

example body:

```
00 00 00 00 00 00                                flags
    ^----- 01 for fw update available
57 cd 54 13                                       unix time
23 23 23 23                                       u-/nseconds?
```

Security conclusions

- configuration phase with AP mode allows for **evil twin & MitM** attacks
- attacker can obtain wifi-credentials & dash token
- rather low risk due to **short time span**

- communication with server uses HTTPS
- server cert is checked (source: internet - didn't check this myself)
- client requests includes **counter and HMAC**
- prevents replays & ordering without knowing secret key

- button only active & connected to wifi for a few seconds following button press
- no self-induced wakeup, limited battery life, no open ports
- risk of **CYBER**: negligible

Firmware analysis

The firmware

- old button: Broadcom WiCED with Express Logic RTOS and NetX IP stack
- new button: **custom OS**

multiple tasks:

- main
- transaction task
- avocado button
- LED task
- command handler
- net manager

```
**** TA05 Bootloader 0.2.11 ****
0x00000004 ms    0x000000FB us
Reset Trigger : FIRST POWER UP
(APP)(INFO)Chip ID 1503a0
(APP)(INFO)DriverVerInfo: 0x134a134a
(APP)(INFO)Firmware ver   : 19.4.10 Svnrev 12577
(APP)(INFO)Firmware Build May 10 2016 Time 00:50:07
(APP)(INFO)Firmware Min driver ver : 19.3.0
(APP)(INFO)Driver ver: 19.4.10 Svnrev 12577
(APP)(INFO)Driver SVN URL branches/WIFIOT-1400_2
(APP)(INFO)Driver built at Jul 15 2016  11:31:12
DBG: Set MAC address [REDACTED]
[000.500] IF0: CMDHDLR:

Welcome to TaOS

[000.500] IF0: NETMGR: WiNC FW version: 19.4.10
[000.500] IF0: NETMGR: default power save mode is 0
[000.500] IF0: MAIN: Firmware Version: 30017420
[000.500] IF0: MAIN: Journal Counter Entry Recovered,:0
[000.500] DBG: MAIN: (main_task): Current PT Region:2
```

Dumping the firmware

- SWD cannot be used to dump firmware
- flash-access using the BootROM isn't possible either
- security lockbit cannot be cleared without full flash erase
- clearing the readout protection can be done with ERASE-pin
- but this erases the flash contents as well
- MCU needs to be desoldered for this (ERASE tied to GND)

Analyzing the firmware

- firmware was obtained by **dumping the SPI flash** (used <https://www.flashrom.org> and a Raspi)
- direct execution from SPI flash isn't possible
- therefore firmware also must be present in internal flash
- SPI flash probably used during firmware update
- firmware in SPI flash should be a duplicate of internal flash firmware
- analysis of firmware with hexeditor and disassembler



Analyzing the SPI flash contents

- flash contains firmware and dynamic storage with journaling
- dynamic storage seems to start at 0x19 0000
(includes debug logs and transaction counter)
- start of flash contains list of static blocks

Analyzing the SPI flash contents (2)

List of static blocks:

```
00000000 | 73 61 6D 67 35 35 00 00 FE FF FF FF 00 02 00 00 44 48 | samg55.....DH
00000012 | 07 00 30 2E 33 2E 31 37 34 00 00 00 00 00 00 00 00 | ..0.3.174.....
00000024 | 77 69 6E 63 00 00 00 00 F3 FF FF FF 00 4B 07 00 48 4E | winc.....K..HN
00000036 | 06 00 31 39 2E 34 2E 31 30 00 00 00 00 00 00 00 00 | ..19.4.10.....
00000048 | 62 6C 65 00 00 00 00 00 FF FF FF FF 00 9A 0D 00 4E E9 | ble.....N.
0000005a | 02 00 30 2E 32 2E 34 30 00 00 00 00 00 00 00 00 00 | ..0.2.40.....
0000006c | 70 77 72 74 62 6C 00 00 FB FF FF FF 00 84 10 00 00 32 | pwrtbl.....2
0000007e | 00 00 30 2E 31 2E 30 00 00 00 00 00 00 00 00 00 00 | ..0.1.0.....
00000090 | 62 75 72 73 74 54 78 00 FF FF FF FF 00 B6 10 00 48 FE | burstTx.....H.
000000a2 | 01 00 37 39 36 00 00 00 00 00 00 00 00 00 00 00 00 | ..796.....
```

Guessed structure:

```
struct block_s {
    char name[8];
    uint32_t unk1;
    uint32_t start_ofs;
    uint32_t len;
    char version[16]; };
```

Analyzing the SPI flash contents (3)

Parsed list:

```
samg55  ofs    200 len 74844 (477252) version 0.3.174
      dst 404000 len   74844 nvic_ofs  404200 flags 4
winc    ofs 74b00 len 64e48 (413256) version 19.4.10
ble     ofs d9a00 len 2e94e (190798) version  0.2.40
pwrtbl  ofs 108400 len  3200 ( 12800) version  0.1.0
burstTx ofs 10b600 len 1fe48 (130632) version      796
```

- **samg55** block is the firmware for the ARM
- payload of this block starts with additional header
- I dumped the samg55 block to extra file for further analysis

Understanding ARM Cortex-Mx firmware

- SRAM usually starts at 0x2000 0000
- internal flash starts at 0x40 0000
- Nested Vector Interrupt Controller (NVIC) needs list of interrupt handler entrypoints
- pointer to this table is written to Vector Table Offset Register (VTOR)
- NVIC table starts with stack pointer, reset vector and other exception handlers
- this is what you look for
- **stack pointer** should point to **RAM**, **handlers** should point to **flash**

Analyzing the samg55 firmware

Hexdump of firmware:

```
0000: 00404000 00074844 00404200 00000004
```

```
    ^---- additional header
```

```
<plenty of zeroes>
```

```
0200: 200204d0 00433eed 00433fbd 00433fbd
```

```
    ^---- stack pointer, handler entries
```

```
<more handler entries>
```


Analyzing the samg55 firmware (2)

Where in internal flash does the firmware end up?

- initial assumption: with first 0x200 bytes stripped
firmware should reside at 0x40 4000
- assumption was wrong
- don't strip additional header
firmware with additional header at 0x40 4000
→ NVIC table located at 0x40 4200
- how to tell?
- base offset is wrong if references during disassembly don't make sense

Analyzing the samg55 firmware (3)

- after reset NVIC table is expected at 0x40 0000
- dumped firmware starts at 0x40 4000
→ there must be some **bootloader code** at 0x40 0000

a glimpse into the future:

- **bootloader** size is 0x1800 (with CRC32 at the end?)
- **config storage** at 0x40 1E00 (MAC addresses, device serial & secret)
- **user config** at 0x40 2000 (wifi config & customer secret)

Let's try something...

Can we execute the dumped firmware on another SAMG55 without the bootloader?

- wrote firmware to empty SAMG55
- NVIC table duplicated from 0x40 4200 to 0x40 0000
- lockbit GPNVM[1] set to start from flash
- **firmware works**

Let's try something...

Can we execute the dumped firmware on another SAMG55 without the bootloader?

- wrote firmware to empty SAMG55
- NVIC table duplicated from 0x40 4200 to 0x40 0000
- lockbit GPNVM[1] set to start from flash
- **firmware works**
- **debugging via SWD** possible :-) (I used OpenOCD + ST-Link)
- **devmode console** on UART :-))
firmware checks security lockbit GPNVM[0]
enables devmode console if security bit not set

I want it all

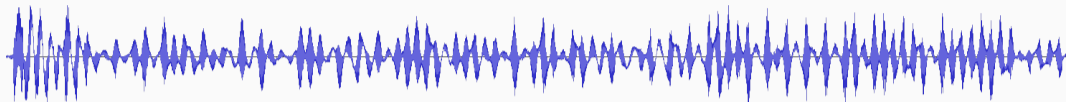
- amazon server doesn't like my "new" button
- apparently valid **credentials missing** in firmware from SPI flash
- need to somehow dump the internal flash of locked SAMG55
- I want a dump of the bootloader anyway

Code execution

Exploiting the firmware

- disassembly of firmware and debugging access with breakpoints, tracing, singlestepping, etc. makes this **a lot** easier
- serial console does length-checking
- exploiting low-level network protocols like DHCP would hit the WINC, not the SAMG55
- there's a http server running on the SAMG55 with TLV- and JSON-parsing during configuration phase
- during config phase there's also the audio config protocol

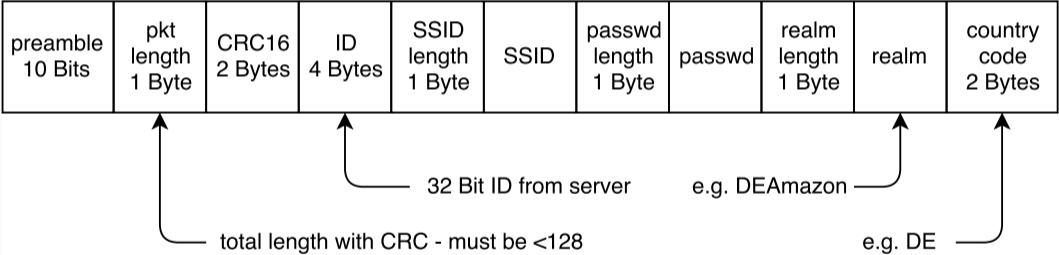
The audio configuration protocol



- used by iOS app - still in use?
- initial analysis by [jaygreco] - he provided me with some updates and sample data
- **FSK with 4 carriers** instead of ASK
- looks like ASK because of low-pass filtering
- carriers: 18.13, 18.62, 19.11 and 19.6 kHz

The audio configuration protocol - payload

payload:



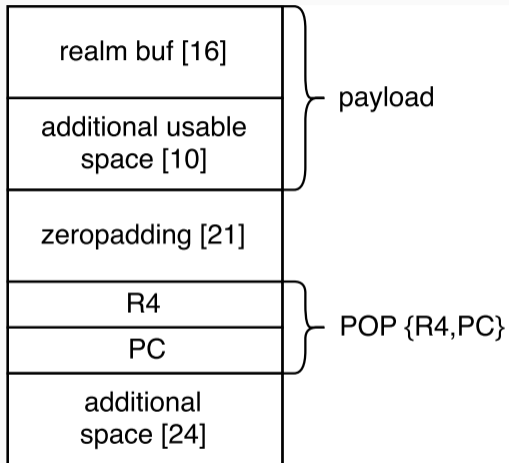
A closer look at the audio config handler

excerpt from `avocado_processHfaPacket` function:

```
ROM:004071E0      STR      R3, [SP,#0xF8+realm+0x18]
ROM:004071E2      LDR      R2, [SP,#0xF8+ssid_len]
ROM:004071E4      LDR      R3, [SP,#0xF8+password_len]
ROM:004071E6      ADD      R3, R2
ROM:004071E8      ADDS     R3, #7
ROM:004071EA      LDR      R2, [SP,#0xF8+buf_]
ROM:004071EC      ADD      R2, R3
ROM:004071EE      LDR      R3, [SP,#0xF8+realm+0x18]
ROM:004071F0      ADD      R1, SP, #0xF8+realm
ROM:004071F2      MOV      R0, R1          ; dst
ROM:004071F4      MOV      R1, R2          ; src
ROM:004071F6      MOV      R2, R3          ; n
ROM:004071F8      LDR      R3, =(memcpy+1)
ROM:004071FA      BLX     R3 ; memcpy      ; copy realm
```

- temporary fixed-len buffers created on stack
- credentials memcopied to these buffers
- **no length checks** in place
- trivial to exploit

Additional constraints

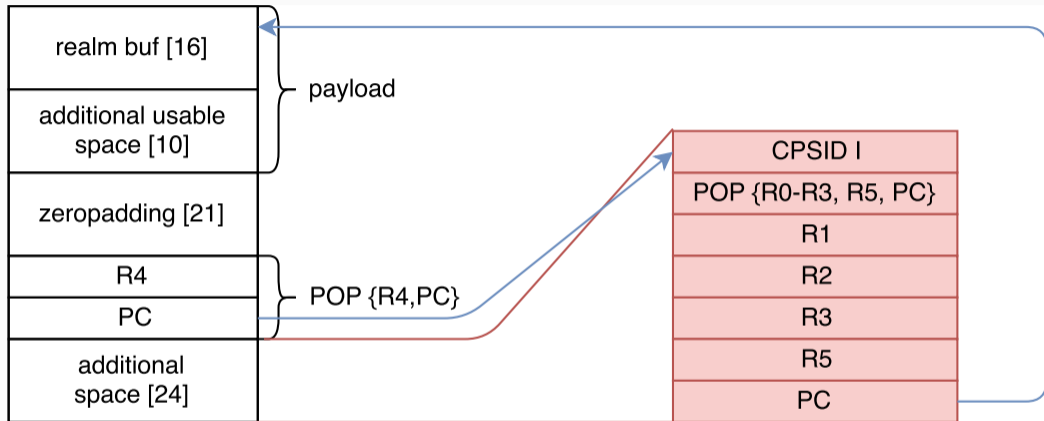


- additional space for payload in password & SSID buffers
- but: total length of audio packet needs to be **<128 Bytes**
- some values on stack after realm
- e.g. src & dst pointer, length values
- invalid memory access triggers exception handler
- **fill stack with zeroes** to avoid this

also needed:

- global IRQ disable
- watchdog servicing

Additional constraints



- immediate 32bit values are needed often (IO locations, functions, etc.)
- LDR takes 2+4 bytes (PC-relative load)
- putting them on the stack and popping them saves a few bytes

Dumping the flash

```
/* R1: src ptr, R2: uart base, R3: uart_write function
   R4: watchdog dst, R5: watchdog value */
MOVS    R0, R2
MOV     R2, #0x1000          /* chunk size */
loop:                                       /* send 1 chunk */
PUSH    {R0-R3}
BLX     R3                   /* uart_write(base,src,n) */
POP     {R0-R3}
ADDS    R1, R2               /* ptr += chunk */
STR     R5, [R4]            /* poke watchdog */
MOVS    R6, R1, LSR #16
CMP     R6, #0x48           /* end of flash? */
BNE    loop
done:                                       /* let the watchdog expire */
```


What now?

- eventually Amazon will probably fix this with a firmware update
- current buttons can only be updated by amazon if they can reach the server
- clearing security bit without erase doesn't work
- software-triggered full erase might work
- otherwise multi-stage loader needed to rewrite flash with custom firmware
- stuff I did so far: <https://github.com/znuh/dashbutton> (u can haz IDC file)
- I'm not really planning on doing some further work here
- if you want to carry on I'm happy to help

contact:

- DECT: hunz
- freenode: hunz

Thanks for your attention

```
Welcome to TaOS
```

```
> exit
```

```
There is no exit from here. You are stuck in a forever loop...MUAHAHAHA!
```

```
>
```


Appendix

MCU IOs (1)

1	VDDIO	3.3V
2	/RST	R/C
3	PB12/ERASE	GND
4	PA4	Wifi:RESETN(34)
5	PA3	NC
6	PA0 (TIOA0)	LED:R
7	PA1	button (TX11/U12)
8	PA5 (SPI2)	Flash:MISO
9	VDDCORE	1.18V
10	TEST	NC
11	PA7	U23: 32kHz clock
12	PA8	NC
13	GND	GND
14	PB15	NC
15	PB14	NC
16	PA31	R1531/BT:Reset? (D8)

MCU IOs (2)

17	PA6 (SPI2)	Flash:MOSI
18	PA16 (SPI2)	Flash:nCS
19	PA30	TX33
20	PA29	NC
21	PA28	TX32/Wifi:SD_CLK(19)
22	PA15 (SPI2)	Flash:SCK
23	PA23 (TIOA1)	LED:G
24	PA22	NC
25	PA21 (TIOA2)	LED:B
26	VDDUSB	3.3V
27	VDDIO	3.3V
28	ADVREF	3.3V
29	GND	GND
30	VDDOUT	1.18V
31	VDDIO	3.3V
32	VDDIO	3.3V

MCU IOs (3)

33	PA17	AD0 (Vbat)
34	PA18	BT: C8, B7
35	PA19	power supply latch
36	PA20	BT: A4
37	PB0	NC
38	PB1	BT: A3
39	PB2 (USART1)	RXD1/Bt:TXD (H1)
40	PB3 (USART1)	TXD1/Bt:RXD (J1)
41	PA14 (SPI5)	Wifi:SPI_SCK
42	PA13 (SPI5)	Wifi:SPI_RXD
43	PA12 (SPI5)	Wifi:SPI_TXD
44	PA11 (SPI5)	Wifi:SPI_SSN
45	VDDCORE	VDDCORE
46	PB10 (USART4)	TX16:UART_TX
47	PB11 (USART4)	TX15:UART_RX
48	PA10	TX26/MIC:CLK

MCU IOs (4)

49	PA9	TX13/MIC:DAT
50	PB5	NC
51	PA27	TX31/Wifi:SD_DAT3(12)
52	PA26	NC
53	GND	GND
54	PB6	SWDIO
55	PB7	SWCLK
56	PA25	NC
57	PB13	NC
58	PA24	NC
59	PB8/XOUT	vbat_adc_enable
60	PB9/XIN	Wifi:CHIP_EN
61	PA2	TPS61200 enable
62	PB4	Wifi:GPIO2/nIRQ(11)
63	JTAGSEL	NC
64	VDDIO	VDDIO

POST to /2/d - metrics

```
HTTPSCONN = 00000003 HTTPSTIMEOUT = 00000000 HTTPSREDSRECV = 00000003  
BLEPAIRED = 00000000 BLETIMEOUT = 00000000 BLEDISCONN = 00000000  
SSCREDDCODED = 00000001 WIFISCANFAIL = 00000000 WIFIAUTHFAIL = 00000000  
WIFICONNTIMEOUT = 00000000 WIFIDHCPTIMEOUT = 00000000 WIFIIIPCONFLICT = 00000000  
HTTPCDNSFAIL = 00000000 HTTPCCONNFAIL = 00000000 HTTPCTIMEOUT = 00000000  
HTTPCCONNABORT = 00000000 MALLOCFAIL = 00000000 DEVWAKEUP = 000000f3  
DEVDEREGISTERED = 00000005 DEVPOWERDOWN = 00000000 OTAINITIATED = 00000000  
OTACOMPLETED = 00000001 REGMODEENTERED = 00000011 REGMODEEXITED = 00000009  
REGMODETIMEOUT = 00000002 REGTOKSENDATTEMPT = 00000003 REGTOKSENDSUCC = 00000003  
REGTOKSENDFAIL = 00000000 ORDERSUCC = 00000000 ORDERFAIL = 0000002f  
FWRECVSUCC = 00000000 FWRECVFAIL = 00000000 FWINTEGFAIL = 00000000  
METRICSSENDSUCC = 00000030 METRICSSENDFAIL = 00000001 BATTOK = 000000ea  
BATTLOW = 00000007 BATTCRITICAL = 00000000
```