Embedded Systems Reverse Engineering

// WEEK 06

Static Variables in Embedded Systems:
Debugging and Hacking Static Variables
w/ GPIO Input Basics

George Mason University

RP2350 // ARM Cortex-M33

Static vs Regular Vars

Persistence across loop iterations

Regular (auto)

Loop 1: 42 -> 43 -> destroy

Loop 2: 42 -> 43 -> destroy

Loop 3: 42 -> 43 -> destroy

Always prints: 42

Static

Loop 1: 42 -> 43 (kept!)

Loop 2: 43 -> 44 (kept!)

Loop 3: 44 -> 45 (kept!)

Keeps incrementing: 42,43,44...

Declaration Syntax

```
uint8_t reg = 42;
```

Recreated each iteration

```
static uint8_t s = 42;
```

Persists for program life

uint8_t Overflow

255 + 1 = 0 (wraps around!)

Binary: 11111111 + 1 = 100000000 (9 bits)

Only 8 bits kept: 00000000 = 0

Memory Layout

Where variables live in RAM

RP2350 SRAM Map

STACK (grows down)
Local/auto variables

0x20082000

(free space)

HEAP (grows up)
malloc / free

.bss section
Uninit static/global

.data section
Initialized static/global

0x20000000

Static vars: .data (init)
Static vars: .bss (uninit)

Variable Storage

Type	Location
Automatic	**Stack**
Static	**.data / .bss**
Global	**.data / .bss**
Dynamic	**Heap**

Static vars are NOT on heap!
Fixed location, set at compile time. Lives entire program.

Example:
static_fav_num @ 0x200005A8

Key Insight

Stack vars:
Created + destroyed
each function call

Static vars:
Fixed RAM address
persist entire runtime

GPIO Input Basics

Reading buttons with the RP2350

OUTPUT (before)

```
Pico GPIO 16          --> LED
```

We CONTROL the LED
gpio_put(pin, value)

INPUT (new!)

```
Pico GPIO 15          <-- BTN
```

We READ button state
gpio_get(pin)

Floating Input

No connection =
RANDOM values!

Read 1: HIGH Read 2: LOW
Read 3: HIGH Read 4: HIGH ???

Pull Resistors

Type	Default	Pressed
Pull-Up	HIGH(1)	LOW(0)
Pull-Down	LOW(0)	HIGH(1)

Pico 2 has internal pull resistors!

GPIO Input Functions

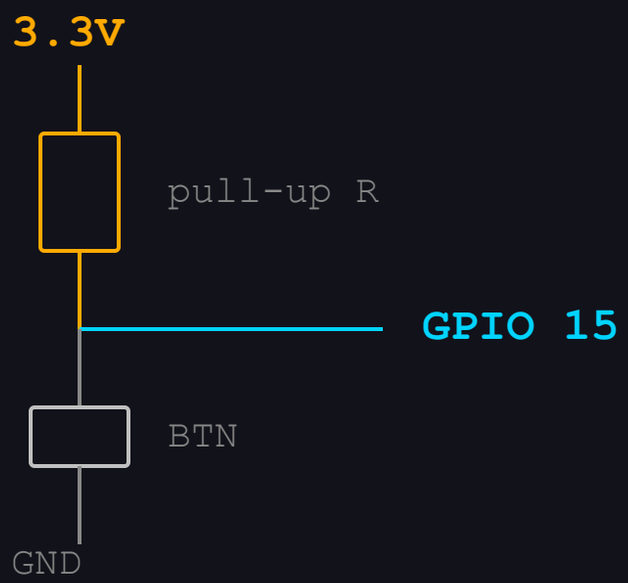
`gpio_init(pin)`	Initialize pin
`gpio_set_dir(pin, GPIO_IN)`	Set as input
`gpio_pull_up(pin)`	Enable pull-up
`gpio_get(pin)`	Read state (0 or 1)

No external resistor needed -- internal pull-up!

Pull-Up Resistor

Internal pull-up on GPIO 15

Circuit



Button Logic

State	GPIO	LED
Released	HIGH (1)	OFF
Pressed	LOW (0)	ON

Inverted! Pull-up = backwards

Ternary Operator

```
gpio_put(LED, pressed?0:1);
```

pressed=1 -> LED OFF (inverted!)

Hardware Wiring

Pico 2		Components
GPIO 15	-->	Button (one leg)
GPIO 16	-->	LED + resistor
GND	-->	Button (other leg)
BOOTSEL		Hold to flash UF2

Key Point

No external resistor needed!

Internal pull-up handles it all

Source Code

0x0014_static-variables.c

main()

```c
int main(void) {
  stdio_init_all();
  // GPIO setup
  gpio_init(15);
  gpio_set_dir(15, GPIO_IN);
  gpio_pull_up(15);
  gpio_init(16);
  gpio_set_dir(16, GPIO_OUT);
  while (true) {
    uint8_t reg = 42;
    static uint8_t s = 42;
    printf(... reg, s);
    reg++; s++;
  }
}
```

GPIO Setup

Pin 15: Input

Pull-up enabled

Pin 16: Output

LED control

Serial Output

```
reg: 42
s: 42
reg: 42
s: 43
```

reg always 42, s grows

Button Logic

```
pressed = gpio_get(15);
```

Key Behaviors

reg: always 42	s: 42,43,44...	wraps at 255->0

Compiler Optimizations

What the compiler does to your code

Optimization 1: Dead Code Removal

Your code:

```
uint8_t reg = 42;
reg++;
// No lasting effect!
```

Compiler output:

```
movs r1, #42
// reg++ is GONE
// Uses constant 42 directly
```

Optimization 2: Function Inlining

Your code:

```
gpio_pull_up(15);
// Simple wrapper func
```

Compiler output:

```
gpio_set_pulls(15,1,0);
// Inlined to real func
```

Optimization 3: Scheduling

Compiler reorders instructions
to avoid pipeline stalls:

Load SIO base | **Increment s** | **Read GPIO**

Assembly Analysis

Key instructions in main() loop

Loop Body (0x10000264)

```
movs r1, #0x2a              ; reg=42
bl __wrap_printf            ; print reg
ldrb r1, [r4]               ; load static
bl __wrap_printf            ; print s
mov.w r1, #0xd0000000       ; SIO base
ldrb r3, [r4]               ; reload s
adds r3, #1                 ; s++
strb r3, [r4]               ; store s
ldr r3, [r1, #4]            ; read GPIO
ubfx r3, r3, #15, #1        ; bit 15
eor.w r3, r3, #1            ; invert
mcrr 0, 4, r2, r3, cr0      ; GPIO out
b.n 0x10000264              ; loop
```

Key Registers

r1: printf arg

r3: temp / static

r4: 0x200005a8
(static var addr)

r2: LED pin (16)

Key Patterns

ldrb/adds/strb
Load-increment-store
(static var update)

ubfx #15, #1
Extract GPIO bit 15

eor.w #1
Inverts (? 0 : 1)

Infinite Loop

```
b.n 0x10000264
; while(true)
```

No pop/bx lr

main() never returns

GDB: Static Variable

Finding static_fav_num in RAM

Literal Pool Lookup

```
ldr r4, [pc, #44]      @ 0x10000290
```

Examine literal pool:

```
x/1wx 0x10000290 = 0x200005A8
```

r4 now holds the RAM address of static_fav_num!

Read Value

```
x/1db 0x200005a8 = 42
```

After one loop iteration:

```
x/1db 0x200005a8 = 43
```

It incremented! Persists in RAM.

Disasm Gotcha

x/i 0x10000290 shows:

```
lsls r0, r5, #22
```

GDB decodes DATA as code!

Bytes A8 05 00 20 = 0x200005A8
Use x/wx not x/i for data

GPIO Input Register

Read button state in GDB:

```
p/x (*(uint*)0xd0000004 >> 15) & 1
```

Returns 1: not pressed (pull-up)
Returns 0: button pressed

Hacking the Binary

Two patches with a hex editor

File Offset Formula

offset = address - 0x10000000

Example: 0x10000264 -> 0x264

Hack 1: Change 42 to 43

Target: movs r1, #0x2a

at address 0x10000264 (offset 0x264)

Before: 2A 21 movs r1, #42

After: 2B 21 movs r1, #43

Thumb encoding: imm8 byte first, opcode 0x21 second

Hack 2: Invert Button Logic

Target: eor.w r3, r3, #1

at address 0x10000286 (offset 0x286)

Before: 83 F0 01 03

After: 83 F0 00 03

Byte at offset 0x288: **01** -> **00**

XOR with 0 = no invert

LED now ON by default, OFF when pressed

Static Vars & GPIO Input

Static variables, GPIO input, hacking

Static vs Auto

Aspect	Auto	Static
Where	Stack	.data
Life	Scope	Forever
Init	Every	Once
Keeps?	No	Yes
Optimized?	Often	In RAM

Compiler may remove auto vars

GPIO Input Setup

1. gpio_init(pin)
2. gpio_set_dir(pin, GPIO_IN)
3. gpio_pull_up(pin)
4. gpio_get(pin)

Pull-up: released=HIGH
pressed=LOW (inverted!)
Internal R, no hardware

Key Instructions

ubfx r3,r3,#15,#1
Extract single GPIO bit

eor.w r3,r3,#1
XOR to invert logic
b.n 0x10000264

Hacking Workflow

1. Analyze in GDB
2. Calculate offset
3. Patch .bin in HxD
4. uf2conv.py + flash

Takeaways

42 -> 43 (1 byte)

XOR 1->0 (invert)

offset = addr - base