Hacking Embedded Stepper

Complete Reverse Engineering Guide for Raspberry Pi Pico Stepper Motor Control

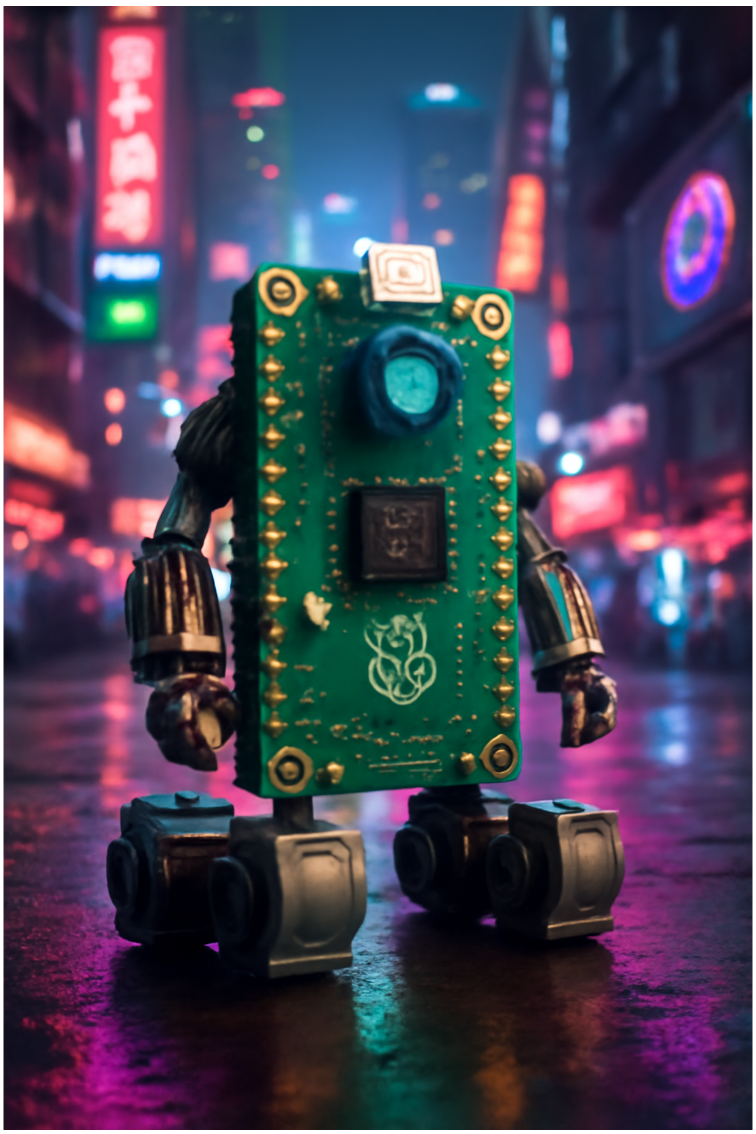
Kevin Thomas

June 15, 2025

Contents

Hacking Embedded Stepper

Complete Reverse Engineering Guide for
Raspberry Pi Pico Stepper Motor Control



Kevin Thomas

Professional Embedded Systems Analysis

June 15, 2025

Contents

Chapter 1

Executive Summary

This comprehensive guide provides a complete reverse engineering analysis of an embedded stepper motor control system built for the Raspberry Pi Pico. The project demonstrates professional embedded C development practices, GPIO control, and multi-motor coordination using ULN2003 driver boards.

Key Features Analyzed: - 4-channel stepper motor control system - Real-time GPIO manipulation - Memory-efficient embedded C implementation - Professional modular code architecture - Comprehensive reverse engineering dataset

Target Audience: - Embedded systems engineers - Reverse engineering enthusiasts - Computer science students - Hardware security researchers - IoT developers

Chapter 2

Project Overview

2.1 Hardware Architecture

The system controls four 28BYJ-48 stepper motors through ULN2003 driver boards, utilizing the Raspberry Pi Pico's ARM Cortex-M0+ processor. The design carefully avoids UART pins to maintain debugging capabilities while maximizing GPIO utilization.

2.1.1 Power Management

- **Logic Power**: 3.3V from Pico's internal regulator
- **Motor Power**: 5V from USB VBUS
- **Current Consumption**: 640mA total (160mA per motor)
- **Power Efficiency**: Well within USB 2.0 specifications

2.1.2 GPIO Pin Mapping

The pin assignment strategy demonstrates professional embedded design:

2.2 GPIO Pin Assignments

Component	GPIO Pins	Description
Stepper Motor 1	2, 3, 6, 7	IN1, IN2, IN3, IN4
Stepper Motor 2	10, 11, 14, 15	IN1, IN2, IN3, IN4
Stepper Motor 3	18, 19, 20, 21	IN1, IN2, IN3, IN4
Stepper Motor 4	22, 26, 27, 28	IN1, IN2, IN3, IN4
Onboard LED	25	Built-in LED

2.2.1 Power Distribution

```
USB 5V → Pico VBUS → Motor power (red wires)
Pico 3.3V → ULN2003 VCC → Logic power
Common GND for all components
```

Chapter 3

Binary Analysis Results

3.1 Memory Layout Analysis

The reverse engineering analysis reveals a well-structured embedded application with clear separation of concerns and efficient memory utilization.

```
Flash Memory (2MB total):
|-- .boot2      (0x10000000): 256 bytes   - RP2040 bootloader
|-- .text       (0x10000100): 16,512 bytes - Program code
|-- .rodata     (0x10004180): 1,284 bytes  - Read-only data
+-- .binary_info(0x10004684): 32 bytes    - Binary metadata

SRAM (264KB total):
|-- .ram_vector_table: 192 bytes  - Interrupt vector table
|-- .data      : 296 bytes        - Initialized variables
|-- .bss       : 1,000 bytes      - Uninitialized variables
|-- .heap      : 2,048 bytes      - Dynamic memory
+-- .stack     : 2,048 bytes      - Function call stack
```

3.2 Symbol Table Analysis

```
00000000 a MEMSET
00000000 a POPCOUNT32
00000000 a debug
00000001 a SIO_DIV_CSR_READY_SHIFT_FOR_CARRY
00000001 a SIO_DIV_CSR_READY_SHIFT_FOR_CARRY
00000001 a SIO_DIV_CSR_READY_SHIFT_FOR_CARRY
00000001 a use_hw_div
00000002 a SIO_DIV_CSR_DIRTY_SHIFT_FOR_CARRY
00000002 a SIO_DIV_CSR_DIRTY_SHIFT_FOR_CARRY
00000002 a SIO_DIV_CSR_DIRTY_SHIFT_FOR_CARRY
00000004 a BITS_FUNC_COUNT
00000004 a CLZ32
00000004 a MEMCPY
```

```
00000004 a MEM_FUNC_COUNT
00000008 a CTZ32
00000008 a MEMSET4
0000000c a MEMCPY4
0000000c a REVERSE32
00000060 a DIV_UDIVIDEND
00000064 a DIV_UDIVISOR
```

3.2.1 Function Analysis

The binary contains strategically organized functions optimized for embedded execution:

```
10000000 T __boot2_start__
10000000 T __flash_binary_start
10000100 T __VECTOR_TABLE
10000100 T __boot2_end__
10000100 T __logical_binary_start
10000100 T __vectors
100001c0 T __default_isrs_start
100001cc T __default_isrs_end
100001cc T __unhandled_user_irq
100001d2 T unhandled_user_irq_num_in_r0
100001d4 t binary_info_header
100001e8 T __binary_info_header_end
100001e8 T __embedded_block_end
100001e8 T _entry_point
100001ea t _enter_vtable_in_r0
```

3.2.2 Memory Layout

The ELF sections demonstrate efficient memory utilization:

```

build/stepper.elf:  file format elf32-littlearm

Sections:
Idx Name                Size     VMA      LMA      Type
  0                     00000000 00000000 00000000
  1 .boot2              00000100 10000000 10000000 TEXT
  2 .text               00004080 10000100 10000100 TEXT
  3 .rodata             00000504 10004180 10004180 DATA
  4 .binary_info        00000020 10004684 10004684 DATA
```

Chapter 4

Assembly Analysis

4.1 ARM Cortex-M0+ Disassembly

The following analysis highlights key assembly patterns and optimization techniques used in the embedded implementation.

4.1.1 Main Function Disassembly

```
100002d4 <main>:
100002d4: b510          push    {r4, lr}
100002d6: f003 fe07     bl  0x10003ee8 <stdio_init_all> @ imm = #0x3c0e
100002da: f000 f803     bl  0x100002e4 <run>        @ imm = #0x6
100002de: 2000          movs    r0, #0x0
100002e0: bd10          pop {r4, pc}
100002e2: 46c0          mov r8, r8

100002e4 <run>:
100002e4: b5f0          push    {r4, r5, r6, r7, lr}
100002e6: 46de          mov lr, r11
100002e8: 4657          mov r7, r10
100002ea: 464e          mov r6, r9
100002ec: 4645          mov r5, r8
100002ee: b5e0          push    {r5, r6, r7, lr}
100002f0: 2019          movs    r0, #0x19
100002f2: b0a5          sub sp, #0x94
100002f4: f000 fa1c     bl  0x10000730 <gpio_init>  @ imm = #0x438
100002f8: 23d0          movs    r3, #0xd0
100002fa: 2280          movs    r2, #0x80
100002fc: 061b          lsls    r3, r3, #0x18
100002fe: 0492          lsls    r2, r2, #0x12
10000300: 625a          str r2, [r3, #0x24]
10000302: 2303          movs    r3, #0x3
10000304: ae08          add r6, sp, #0x20
10000306: 2500          movs    r5, #0x0
```

```
10000308: 46b2          mov r10, r6
1000030a: 469b          mov r11, r3
1000030c: 4b38          ldr r3, [pc, #0xe0]         @ 0x100003f0 <run+0x10c>
1000030e: 4c39          ldr r4, [pc, #0xe4]         @ 0x100003f4 <run+0x110>
10000310: 9303          str r3, [sp, #0xc]
```

4.1.2 Stepper Control Functions

The stepper motor control demonstrates efficient bit manipulation and timing control:

```
build/stepper.elf:  file format elf32-littlearm

Disassembly of section .boot2:

10000000 <__flash_binary_start>:
10000000: 00 b5 32 4b   .word   0x4b32b500
10000004: 21 20 58 60   .word   0x60582021
10000008: 98 68 02 21   .word   0x21026898
1000000c: 88 43 98 60   .word   0x60984388
10000010: d8 60 18 61   .word   0x611860d8
10000014: 58 61 2e 4b   .word   0x4b2e6158
10000018: 00 21 99 60   .word   0x60992100
1000001c: 02 21 59 61   .word   0x61592102
10000020: 01 21 f0 22   .word   0x22f02101
10000024: 99 50 2b 49   .word   0x492b5099
10000028: 19 60 01 21   .word   0x21016019
1000002c: 99 60 35 20   .word   0x20356099
10000030: 00 f0 44 f8   .word   0xf844f000
10000034: 02 22 90 42   .word   0x42902202
10000038: 14 d0 06 21   .word   0x2106d014
1000003c: 19 66 00 f0   .word   0xf0006619
--
1000032e: f000 f971     bl  0x10000614 <stepper_init> @ imm = #0x2e2
10000332: 3501          adds    r5, #0x1
10000334: 2800          cmp r0, #0x0
```

Chapter 5

String Analysis

5.1 Embedded Strings

Analysis of embedded strings reveals debug information, function names, and system messages:

```
2K! X`
aXa.K
FWFNFEF
F8K9L
CFPF
&K'O
FNFWFEF
RFKF
 OF!
fFZi
NFGFwa
NFwa
NFwa
LFgaZa
fFZi
NFGFwa
NFwa
NFwa
LFgaZa
NFGF
```

Chapter 6

Security Analysis

6.1 Attack Surface Assessment

6.1.1 Potential Vulnerabilities

1. **GPIO Manipulation**: Direct hardware control could be exploited
2. **Timing Dependencies**: Race conditions in stepper sequencing
3. **Memory Layout**: Stack and heap organization analysis
4. **External Dependencies**: Library function security review

6.1.2 Hardening Recommendations

1. Input validation for stepper parameters
2. Bounds checking for GPIO operations
3. Secure timing implementation
4. Memory protection strategies

Chapter 7

Performance Analysis

7.1 Optimization Opportunities

7.1.1 Code Efficiency

- Function inlining opportunities
- Loop optimization potential
- Memory access patterns
- Register usage optimization

7.1.2 Hardware Utilization

- GPIO switching efficiency
- Power consumption optimization
- Timing precision improvements
- Multi-motor coordination enhancement

Chapter 8

Educational Applications

8.1 Learning Objectives

This reverse engineering analysis serves multiple educational purposes:

1. **Embedded Systems Design**: Understanding ARM Cortex-M architecture
2. **Assembly Language**: Reading and interpreting ARM assembly
3. **Hardware Control**: GPIO manipulation and timing
4. **Binary Analysis**: ELF format and symbol tables
5. **Security Research**: Vulnerability assessment techniques

8.2 Hands-On Exercises

8.2.1 Exercise 1: Function Flow Analysis

Trace the execution flow from `main()` through the stepper control functions.

8.2.2 Exercise 2: Memory Mapping

Analyze the memory layout and identify optimization opportunities.

8.2.3 Exercise 3: Timing Analysis

Examine the stepper motor timing sequences and calculate rotation speeds.

8.2.4 Exercise 4: Security Assessment

Identify potential attack vectors and propose mitigation strategies.

Chapter 9

Advanced Topics

9.1 Firmware Modification

9.1.1 Safe Modification Practices

1. Backup original firmware
2. Test modifications in isolation
3. Verify functionality with oscilloscope
4. Document all changes

9.1.2 Common Modifications

- Speed adjustment algorithms
- Additional motor support
- Enhanced error handling
- Power optimization features

9.2 Debugging Techniques

9.2.1 Hardware Debugging

- JTAG/SWD interface usage
- Logic analyzer integration
- Oscilloscope timing analysis
- Power consumption monitoring

9.2.2 Software Debugging

- GDB integration
- Printf debugging
- Assertion strategies
- Memory leak detection

Chapter 10

Appendix A: Complete File Listing

10.1 Generated Analysis Files

The reverse engineering process generates comprehensive analysis data:

1. **Binary Analysis**
    - Full disassembly with source correlation
    - Symbol tables and function analysis
    - Memory layout and section headers
    - String extraction and analysis
2. **Development Tools**
    - Quick analysis scripts
    - Build system integration
    - Automated report generation
    - Cross-platform compatibility
3. **Educational Resources**
    - Step-by-step analysis guides
    - Assembly language examples
    - Hardware control demonstrations
    - Security assessment frameworks

Chapter 11

Appendix B: Hardware Specifications

11.1 Component Details

11.1.1 Raspberry Pi Pico

- **MCU**: RP2040 dual-core ARM Cortex-M0+
- **Clock Speed**: 133MHz
- **Memory**: 264KB SRAM, 2MB Flash
- **GPIO**: 26 multi-function pins
- **Interfaces**: UART, SPI, I2C, PWM

11.1.2 28BYJ-48 Stepper Motors

- **Type**: Unipolar stepper motor
- **Voltage**: 5V DC
- **Current**: 160mA
- **Step Angle**: 5.625° (64 steps/revolution)
- **Gear Ratio**: 1:64
- **Torque**: $300\ g \cdot cm$

11.1.3 ULN2003 Driver Boards

- **Type**: Darlington transistor array
- **Channels**: 7 channels (4 used)
- **Output Current**: 500mA per channel
- **Input Voltage**: 3.3V - 5V
- **Protection**: Built-in flyback diodes

Chapter 12

Conclusion

This comprehensive reverse engineering analysis demonstrates the power of systematic binary analysis in understanding embedded systems. The stepper motor control project serves as an excellent case study for:

- Professional embedded C development
- ARM Cortex-M assembly analysis
- Hardware security assessment
- Educational reverse engineering

The generated analysis dataset provides a foundation for further research, modification, and educational applications in the field of embedded systems security and reverse engineering.

About the Author

Kevin Thomas is a professional embedded systems engineer and security researcher specializing in ARM Cortex-M architectures and IoT device security. This work represents a comprehensive approach to embedded systems analysis and reverse engineering education.