HACKING BITS

8-BIT DEVELOPMENT & REVERSE ENGINEERING

MYTECHNOTALENT.COM

KEVIN THOMAS

Hacking Bits

First edition

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

Contents

Foreword

To the Cyber Warriors of the 91st Cyber Brigade:

Welcome to *Hacking Bits* — a journey into the heart of embedded systems, low-level programming, and the art of reverse engineering. This book is crafted for you, the digital defenders and innovators of tomorrow, who stand at the frontline of cybersecurity in a world where every byte matters, every circuit tells a story, and every vulnerability is an opportunity to strengthen our digital fortresses.

Why 8-Bit?

The humble 8-bit microcontroller remains a cornerstone of modern technology. Whether in critical infrastructure, IoT devices, or legacy systems, the simplicity, efficiency, and resilience of 8-bit systems make them timeless. Understanding them isn't just nostalgia — it's strategic insight.

What We'll Explore Together

- **Assembler on Arduino Nano:** Master low-level programming on one of the most versatile microcontrollers.
- **Hardware Interaction:** Learn how software meets the physical world through GPIO pins, sensors, and actuators.

- **Reverse Engineering with Ghidra:** Disassemble, analyze, and understand compiled binaries to uncover hidden truths and vulnerabilities.

The Mission

This book isn't just about writing code or analyzing binaries — it's about cultivating a mindset. You'll learn to think like an engineer, a hacker, and a defender, all at once. Each chapter is designed to equip you with practical skills and real-world insights that directly translate to your mission as cybersecurity professionals.

Adapt. Innovate. Overcome.

In a world where threats evolve faster than defenses, the edge lies in adaptability. By the end of *Hacking Bits*, you'll not only understand 8-bit systems but also gain the confidence to analyze, break, and secure embedded systems in any context.

So, gear up and get ready. Whether you're debugging a circuit, deconstructing firmware, or crafting elegant Assembler code, remember: every bit counts, every byte tells a story, and you are the hero in this digital battlefield.

Onward, Warriors of the 91st. Let's build. Let's break. Let's secure.

— *Kevin Thomas*

1

Chapter 1: Ohm’s Law – The Foundation of Electronics

“Understanding the flow of electrons isn’t just science — it’s the language of control.”

Introduction to Ohm’s Law

At the core of every circuit, whether it's powering a multi-million-dollar defense system or blinking an LED on an Arduino Nano, lies Ohm's Law. Named after George Simon Ohm, this fundamental principle defines the relationship between Voltage (V), Current (I), and Resistance (R).

In this chapter, we'll break down Ohm's Law into actionable knowledge, ensuring you not only understand the theory but also know how to apply it to build, troubleshoot, and secure electronic systems.

What You'll Learn in This Chapter

1.The Formula Behind Ohm's Law:

- Understand the relationship: **V = I x R**
- Explore how voltage, current, and resistance interact in a circuit.

2.Real-World Application:

- How Ohm's Law applies to circuits on an Arduino Nano.
- Practical examples with resistors, LEDs, and power supplies.

3.Power Calculations:

- Learn how to calculate electrical power: **P = V x I**
- Understand energy consumption and efficiency in micro-controller circuits.

4.Debugging with Ohm's Law:

- Identify faulty components.
- Diagnose voltage drops and current bottlenecks.

1.1 The Formula Explained

Ohm's Law is deceptively simple yet profoundly powerful:

V = I x R

- **Voltage (V)**: The "pressure" pushing electrons through the circuit (measured in **Volts**, V).
- **Current (I)**: The flow of electrons (measured in **Amperes**, A).
- **Resistance (R)**: The opposition to the flow of current (measured in **Ohms**, Ω).

Analogy: Water Flow in a Pipe

- **Voltage** is the water pressure.
- **Current** is the amount of water flowing.
- **Resistance** is the width of the pipe.

If the pipe is narrow (**high resistance**), less water (**current**) will flow for the same pressure (**voltage**).

1.2 Ohm's Law in Action: Arduino Nano Example

Objective: Light up an LED safely using Ohm's Law.

Components:

- Arduino Nano
- LED
- Resistor (220Ω)
- Breadboard
- Jumper Wires

Step 1: Circuit Design

- Connect the LED to a **5V** pin on the Arduino Nano.
- Use a **220Ω resistor** in series with the LED.

Step 2: Calculate the Current

Using Ohm's Law:
I = V / R

- V = 5V (voltage from the Arduino pin)
- R = 220

I = 5 / 220 = 0.0227 A (22.7 mA)

This current is well within the safe operating range of an LED.

Step 3: Build and Test

- Upload a simple LED blinking code to your Arduino Nano.
- Verify the LED lights up without overheating.

1.3 Power Calculations

Formula:
P = V x I

Using our example:
P = 5 x 0.0227 = 0.1135W

This tells us the LED and resistor dissipate approximately **0.11W** of power.

Why It Matters:

Understanding power dissipation prevents overheating, damage to components, and energy waste.

1.4 Debugging with Ohm's Law

Ohm's Law is not just theoretical — it's your first tool in debugging circuits.

Common Issues and Ohm's Law Solutions:

- **LED Too Dim?** Check the resistor value — too high, and current drops.
- **Circuit Overheating?** Verify voltage and current match

component ratings.
- **No Light at All?** Test continuity and ensure no open circuits.

Challenge: Calculate the current and power for each LED using Ohm's Law.

Key Takeaways from Chapter 1

1. Ohm's Law connects **Voltage**, **Current**, and **Resistance**.
2. It's the foundation for designing and troubleshooting circuits.
3. Power calculations prevent overheating and component failure.
4. Practical understanding of Ohm's Law empowers you to approach hardware with confidence.

2

Chapter 2: The Binary Number System

In the realm of embedded systems, low-level programming, and reverse engineering, the **binary number system** serves as the foundation upon which everything else is built. It's the native language of computers, microcontrollers, and digital electronics. Every operation, whether simple arithmetic or complex cryptographic algorithms, ultimately reduces to manipulating binary digits—**bits**.

Understanding the binary system isn't just an academic exercise; it's a practical skill. Whether you're analyzing assembly code, reverse-engineering firmware, or debugging hardware interfaces, fluency in binary will allow you to decipher patterns, identify anomalies, and make precise decisions.

2.1 The Nature of Binary

At its core, the **binary number system** is a **base-2 numeral system**. Unlike the familiar **decimal system** (base-10) which uses ten digits (0-9), binary uses only two digits:

- **0 (zero)**
- **1 (one)**

Each digit in a binary number is called a **bit**—short for **binary digit**. A **bit** represents the smallest unit of data in a computer system and can exist in only one of two states: **on (1)** or **off (0)**, **true** or **false**, **high** or **low**.

This simplicity aligns perfectly with physical hardware, where transistors act as microscopic switches that can either allow or block electrical current, corresponding directly to 1 and 0.

2.2 Binary Representation of Numbers

Each digit in a binary number represents an increasing **power of 2**, starting from the rightmost digit (least significant bit, or **LSB**).

Example: Binary to Decimal Conversion

Let's take the binary number 1011 and convert it to decimal:

(1 x 2^3) + (0 x 2^2) + (1 x 2^1) + (1 x 2^0)

Breaking it down:

- 1 x 2^3 = 8
- 0 x 2^2 = 0

- 1 x 2^1 = 2
- 1 x 2^0 = 1

Add them together:

8 + 0 + 2 + 1 = 11

So, **1011 (binary) = 11 (decimal)**.

2.3 Decimal to Binary Conversion

To convert a **decimal number** into **binary**, repeatedly divide the decimal number by 2, recording the **remainder** each time. The remainders, when read in reverse order, represent the binary equivalent.

Example: Decimal to Binary Conversion

Convert **13 (decimal)** into binary:

1. 13 / 2 = 6 remainder **1**
2. 6 / 2 = 3 remainder **0**
3. 3 / 2 = 1 remainder **1**
4. 1 / 2 = 0 remainder **1**

Reading the remainders from bottom to top: **1101**

So, **13 (decimal) = 1101 (binary)**.

2.4 Binary Arithmetic

Addition

Binary addition follows four basic rules:

1. **0 + 0 = 0**
2. **0 + 1 = 1**
3. **1 + 0 = 1**
4. **1 + 1 = 10** (*carry the 1*)

Example: Binary Addition

Add **1011** and **1101**:

```
1011
+ 1101
  ----
 11000
```

- 1 + 1 = 10 → Write 0, carry 1
- 1 + 0 + 1 (carry) = 10 → Write 0, carry 1
- 0 + 1 + 1 (carry) = 10 → Write 0, carry 1
- 1 + 1 + 1 (carry) = 11 → Write 1, carry 1
- Bring down the final carry → 11000

So, **1011 + 1101 = 11000 (binary)**.

Subtraction

Binary subtraction also follows simple rules:

1. **0 - 0 = 0**
2. **1 - 0 = 1**
3. **1 - 1 = 0**
4. **0 - 1 = 1** (*borrow 1 from the next higher bit*)

Binary multiplication and division are extensions of decimal rules but adapted for base-2 arithmetic.

2.5 Binary and Hardware

At the hardware level:

- **Transistors** represent binary values as **high (1)** or **low (0)** voltages.
- **Registers** in the microcontroller hold binary data for computations and state storage.
- **Memory Addresses** are represented in binary, allowing precise access to data.

For example, setting **GPIO pins** on an 8-bit microcontroller often involves writing binary values to control registers:

```
DDRB = 0b11110000;
```

This configures the **upper 4 pins as outputs** and **lower 4 pins as inputs**.

2.6 Hexadecimal Representation

Writing long binary numbers can become tedious and error-prone. To simplify, we often use the **hexadecimal (base-16) system**, where each hex digit represents **4 bits**.

```
Binary | Hexadecimal
0000   | 0
0001   | 1
1010   | A
1111   | F
```

Example: Binary to Hex

Convert **10111011** (binary) to hex:

- Group into nibbles: **1011 1011**
- First nibble: **1011 = B**
- Second nibble: **1011 = B**

So, **10111011 (binary) = 0xBB (hex)**.

2.7 Practical Applications in Cybersecurity

- **Bitwise Operations:** Used extensively in encryption algorithms, checksums, and compression techniques.
- **Registers and Flags:** Hardware states are manipulated using binary flags.
- **Reverse Engineering:** Understanding binary data allows you to interpret raw machine code, firmware dumps, and

memory contents.

2.8 Conclusion

The **binary number system** is the foundation upon which digital systems are built. Every microcontroller operation, every hardware register configuration, and every network packet carries the essence of binary logic.

As you move forward, remember: **binary isn't just a language computers understand—it's the language we use to command them.**

3

Chapter 3: The Hexadecimal Number System

In the world of embedded systems, low-level programming, and reverse engineering, the **hexadecimal (hex) number system** plays a crucial role in bridging the gap between **binary** and **human readability**. While binary is the native language of computers, its long strings of 1s and 0s are cumbersome for humans to interpret. Hexadecimal offers a concise, efficient way to represent binary data, making it an indispensable tool for programmers, engineers, and cybersecurity professionals.

3.1 What is Hexadecimal?

The **hexadecimal number system** is a **base-16 numeral system**. It uses **16 distinct symbols** to represent values:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
```

- **0–9:** Represent their usual numeric values.
- **A–F:** Represent decimal values **10–15**.

Each **hexadecimal digit** represents **4 bits** (a nibble). This alignment with binary makes hex an ideal shorthand for representing binary numbers.

Comparison Between Decimal, Binary, and Hexadecimal

```
Decimal | Binary    | Hexadecimal
0       | 0000      | 0
1       | 0001      | 1
10      | 1010      | A
15      | 1111      | F
16      | 0001 0000 | 10
255     | 1111 1111 | FF
```

3.2 Why Use Hexadecimal?

1. Conciseness

Binary numbers quickly become unwieldy:

- **Binary:** 11111111 (8 bits)
- **Hexadecimal:** FF (2 hex digits)

Hexadecimal compresses binary into a more compact and human-friendly representation.

2. Alignment with Bytes

Each hexadecimal digit maps directly to **4 bits**. Since most computer systems operate in **8-bit chunks (1 byte)**, a byte can be represented with exactly **2 hexadecimal digits**.

3. Readability in Memory Dumps

When reverse engineering firmware or analyzing memory dumps, hex values provide a clean, readable format for data inspection.

4. Address Representation

Memory addresses in embedded systems are often displayed in hexadecimal because of their clarity and compactness.

3.3 Binary to Hexadecimal Conversion

Step 1: Group into Nibbles
Start from the **rightmost digit** and group the binary number into sets of **4 bits**.
Step 2: Convert Each Nibble to Hex

Use the binary-to-hex mapping table to convert each group.

Example: Binary to Hex

Convert 11011011 (binary) to hex:

1. Group into nibbles: 1101 1011
2. First nibble: 1101 → D
3. Second nibble: 1011 → B

So, **11011011 (binary) = 0xDB (hex)**.

3.4 Hexadecimal to Binary Conversion

Step 1: Convert Each Hex Digit to Binary

Use the hex-to-binary mapping table for each digit.

Example: Hex to Binary

Convert 0x3F (hex) to binary:

1. 3 → 0011
2. F → 1111

Combine them: **0011 1111**

So, **0x3F (hex) = 00111111 (binary)**.

3.5 Decimal to Hexadecimal Conversion

To convert a **decimal number** to **hexadecimal**:

1. Divide the decimal number by **16**.
2. Record the **remainder**.

3. Continue dividing until the quotient is **0**.
4. Write the remainders in **reverse order**.

Example: Decimal to Hex

Convert **255 (decimal)** to hexadecimal:

1. 255 \div 16 = 15 remainder **15 (F)**
2. 15 \div 16 = 0 remainder **15 (F)**

Write in reverse order: **FF**

So, **255 (decimal) = 0xFF (hex)**.

3.6 Hexadecimal Arithmetic

Addition

Hexadecimal addition follows decimal rules but carries over every **16**.

Example:

```
0x2A
+  0x1C
-------
   0x46
```

Subtraction

Hex subtraction uses borrowing when needed.

Example:

```
0x3F
-  0x1A
-------
   0x25
```

3.7 Hexadecimal in Embedded Systems

Hexadecimal is deeply embedded in the world of low-level programming:

1. Memory Addresses

Memory locations are almost always expressed in hex:

```
0x2000
0x3FFF
```

2. GPIO Register Configuration

When configuring GPIO pins, hexadecimal is often used for clarity:

```
ldi        r16, 0xFF ; set all pins high
out        DDRB, r16
```

3. Firmware Reverse Engineering

When analyzing firmware dumps in tools like **Ghidra** or **IDA Pro**, memory regions and instructions are displayed in hex.

4. Embedded Hardware Addresses

Registers in microcontrollers are typically assigned hexadecimal addresses.

3.8 Practical Example

Setting GPIO Pins Using Hexadecimal

To set **PB0–PB3** as output pins and keep **PB4–PB7** as inputs:

```
ldi        r16, 0x0F ; binary: 00001111
out        DDRB, r16 ; configure –PB0PB3 as output
```

Explanation:

- **0x0F (hex)** corresponds to **00001111 (binary)**.
- Lower 4 pins (PB0–PB3) are set as output (1).
- Upper 4 pins (PB4–PB7) remain as input (0).

3.9 Conclusion

The **hexadecimal number system** serves as a vital bridge between the raw binary data understood by machines and the human-readable representations needed by programmers and engineers. Its efficiency in representing data, ease of mapping to binary, and widespread use in hardware and firmware analysis make it an essential tool for every embedded systems developer and reverse engineer.

In the next chapter, we'll take these numerical foundations and explore **how GPIO pins interact with hardware on the Arduino Nano**.

4

Chapter 4: The ATmega328P Architecture

The **ATmega328P**, the microcontroller at the heart of the **Arduino Nano**, is a masterpiece of efficient design, balancing processing power, memory, and peripherals in a compact, low-power package. To fully utilize this device for embedded programming, reverse engineering, and cybersecurity applications, we must understand its architecture and memory layout.

This chapter dives into the **Harvard Architecture**, explains the difference between **Flash Memory** and **SRAM**, and resolves common confusion around memory addresses like **0x0000**.

4.1 Harvard Architecture

What is Harvard Architecture?

The **ATmega328P** microcontroller follows the **Harvard Architecture**, a design philosophy where:

- **Program Memory (Flash Memory)** and **Data Memory (SRAM)** are physically and logically separated.
- Separate **buses** are used to access **program instructions** and **data** simultaneously.

This is different from the **Von Neumann Architecture**, where program instructions and data share the **same memory space and bus**.

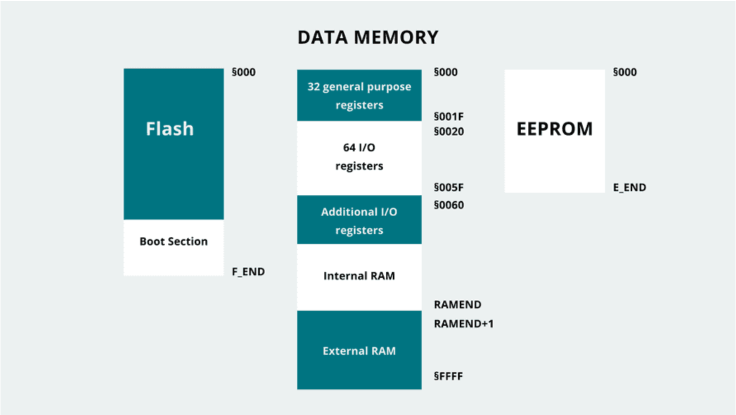
Advantages of Harvard Architecture

1. **Parallelism:** The microcontroller can **fetch instructions** from Flash Memory and **read/write data** from SRAM simultaneously, increasing efficiency.
2. **Memory Protection:** Programs and data are kept in separate spaces, reducing the risk of corruption.
3. **Optimized Instruction Execution:** Dedicated instruction and data buses allow faster processing.

Program Memory vs. Data Memory

- **Program Memory (Flash):** Stores **firmware** or **program code**. This memory is **non-volatile**, meaning it retains data even after a power cycle.
- **Data Memory (SRAM):** Stores **variables**, **stack data**, and **runtime state**. This memory is **volatile**, meaning it loses data when power is removed.

SOURCE - **AVR Memory Map**



4.2 ATmega328P Memory Spaces

1. Flash Memory (Program Space)

- **Size:** 32 KB
- **Type:** Non-volatile (retains data when powered off).
- **Purpose:** Stores the **program code** written in Assembly or C.
- **Access:** Via the **Program Counter (PC)**.

Key addresses:

- **0x0000:** Reset Vector (entry point when the microcontroller starts).
- **0x0002:** Interrupt Vectors (specific addresses for hardware interrupt routines).

Flash memory is **read-only during runtime**, except via self-programming routines (e.g., bootloader updates).

2. SRAM (Static Random Access Memory)

- **Size:** 2 KB
- **Type:** Volatile (cleared on reset or power-off).
- **Purpose:** Stores:
- General-purpose **registers** (R0–R31)
- **I/O Registers** (e.g., Timer, UART, ADC configurations)
- Program **stack** (used for function calls, return addresses, local variables)
- Dynamic **runtime variables**

Memory Map of SRAM:

```
Address Range  | Description
0–x00000x001F  | General-purpose registers –R0R31
0–x00200x005F  | I/O Registers
0–x00600x08FF  | SRAM Data Space
```

3. EEPROM (Electrically Erasable Programmable Read-Only Memory)

- **Size:** 1 KB
- **Type:** Non-volatile (retains data after power-off).
- **Purpose:** Stores persistent data (e.g., device configuration, calibration values).
- **Access:** Byte-wise read/write via special registers (EEAR, EEDR, EECR).

4.3 Why the Confusion?

When dealing with embedded systems, memory addresses can become confusing because the **same address can exist in both Flash and Data memory spaces**, but they refer to **different physical memory locations**.

Consider the address **0x0000**:

📌 **1. In Flash (Program Space)**

- **0x0000** points to the **Reset Vector**, which contains the address of the **first instruction** executed after reset.
- It is accessed using the **Program Counter (PC)**.

📌 **2. In Data Memory (SRAM Space)**

- **0x0000** refers to **General-Purpose Register R0**.
- It is accessed via **SRAM addressing**.

These two 0x0000 addresses coexist in different spaces, and the **CPU knows which one to use based on the instruction context**:

- **Flash:** Accessed with instructions like LPM (Load Program Memory).
- **SRAM:** Accessed with instructions like LD (Load from SRAM).

4.4 Example Code: Accessing Flash and SRAM

Accessing Flash (Program Space)

```
; load a byte from Flash memory at address 0x0000
ldi        r30, 0x00 ; Z-register (R30:R31) points to
Flash 0x0000
ldi        r31, 0x00
lpm        r16, Z    ; load byte from Flash into r16
```

- ldi r30, 0x00: Set the low byte of the Z-register to 0x00.
- ldi r31, 0x00: Set the high byte of the Z-register to 0x00.
- lpm: Load the byte at address 0x0000 in Flash into register r16.

Accessing SRAM (Data Space)

```
; write a value to SRAM address 0x0000 (R0)
ldi        r16, 0x42     ; Load 0x42 into r16
sts        0x0000, r16   ; Store r16 value into SRAM
address 0x0000
```

- ldi r16, 0x42: Load value 0x42 into register r16.
- sts 0x0000, r16: Store r16 into SRAM address 0x0000 (General-Purpose Register R0).

4.5 Memory Access Instructions

1. Flash Memory Instructions

- LPM (Load Program Memory)
- SPM (Store Program Memory, used by bootloaders)

2. SRAM Instructions

- LD (Load from SRAM)
- ST (Store to SRAM)
- LDS (Load Direct from SRAM)
- STS (Store Direct to SRAM)

3. EEPROM Instructions

- EERE (EEPROM Read Enable)
- EEWE (EEPROM Write Enable)

4.6.1 Introduction to AVR Registers

In AVR microcontrollers like the **ATmega328P**, the CPU uses **32 General-Purpose Registers** (r0 to r31) for arithmetic, logic, and data transfer operations. These registers are directly accessible by most AVR assembly instructions, enabling efficient and fast execution of operations.

Understanding the structure, access patterns, and specific uses of these registers is critical for writing optimized and maintainable AVR Assembly programs.

4.6.2 Overview of General-Purpose Registers

The 32 registers (r0–r31) are split into different groups based on functionality:

Register RangePurpose/Usage
r0 – r15 General-purpose registers (can be used for any task).
r16 – r23 Commonly used as **temporary registers** (temp).
r24 – r31 Can be used as **pointer registers** or for arithmetic operations.
r26 – r31 Special pointer registers (X, Y, Z).

These registers are mapped to the **Register File**, which is directly accessible in one clock cycle.

4.6.3 Register File in AVR Architecture

- **AVR Register File:** Located in the **I/O Memory Space**, starting at **address 0x00**.
- Each register is **8 bits wide**.
- The Register File provides **fast access** compared to SRAM or EEPROM.

Register Addressing
r0 0x00
r1 0x01
......

r31 0x1F

Direct Accessibility

- **r0 – r31** are directly accessible via most instructions.
- They can be used as **operands** in arithmetic (ADD, SUB), logical (AND, OR), and data transfer (MOV, LDI) instructions.

4.6.4 Special Roles of Specific Register Ranges

1. r0 – r15: General-Purpose Registers

- Can store temporary data, counters, flags, or intermediate results.
- Rarely used as temporary variables because r16–r31 are more optimized for temporary storage.

2. r16 – r23: Temporary Registers

- These registers are commonly used for temporary storage in assembly routines.
- Often used with LDI (Load Immediate) instructions to store constants or temporary calculations.
- **Why** r16–r23 **for temporary data?**
- They are directly accessible by LDI.
- Avoid conflicts with pointer or special-purpose registers.

3. r24 – r31: High Registers and Pointer Registers

- **r24–r27:** General-purpose but often used in pairs for arithmetic operations (e.g., r24:r25 for 16-bit operations).

- **r26–r31:** Used as **Pointer Registers** for indirect addressing:
- X: r26:r27
- Y: r28:r29
- Z: r30:r31

4.6.5 Best Practices for Using Registers

1. **Use** r16–r23 **for Temporary Data:** They are directly accessible and less likely to conflict with pointer operations.
2. **Reserve** r24–r31 **for Arithmetic and Pointers:** Use them for calculations and indirect addressing.
3. **Minimize Explicit Use of** r0 **and** r1**:** These may be overwritten by internal compiler routines.
4. **Pair Registers for 16-bit Operations:** Use register pairs (r24:r25) for multi-byte calculations.

4.7 Conclusion

Understanding the **Harvard Architecture** and the distinction between **Flash**, **SRAM**, and **EEPROM** is vital for efficient embedded programming and reverse engineering. Misunderstanding address spaces can lead to subtle bugs and wasted debugging time. As you dive deeper into low-level programming, this knowledge will serve as your foundation.

In the next chapter, we'll explore **how GPIO pins on the Arduino Nano interact with hardware peripherals and how memory access impacts them.**

5

Chapter 5: Tools

In this chapter, we'll set up the necessary tools for **ATmega328P development**, **Arduino Nano programming**, and **reverse engineering with Ghidra** across **macOS**, **Windows**, and **Linux**. By the end, you'll have a fully functional development environment ready to compile, upload, and analyze firmware.

5.1 Toolchain Overview

1. AVR Toolchain

- **avr-gcc:** Cross-compiler for AVR microcontrollers.
- **avrdude:** Tool for uploading firmware to AVR microcontrollers.
- **CMake:** Build system for organizing and managing projects.

2. Ghidra

- A powerful reverse engineering tool developed by the **NSA**, capable of analyzing compiled binaries, including AVR firmware.

3. Supporting Tools

- **Python:** Used for scripting and auxiliary tasks.
- **Serial Monitor:** For communicating with microcontrollers (e.g., minicom, Arduino IDE).

5.2 Installing AVR Toolchain

macOS

Open a **terminal** and follow these steps:

1. **Install Homebrew (if not already installed):**

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.s
```

2. Install AVR Toolchain:

```
brew tap osx-cross/avr
brew install avr-gcc
brew install avrdude
```

3. Install CMake:

```
brew install cmake
```

4. Verify Installation:

```
avr-gcc --version
avrdude -v
cmake --version
```

If the versions display correctly, the tools are installed.

Windows

1. Install AVR Toolchain:

- Download **WinAVR** or **Atmel AVR Toolchain** from the official Atmel site.
- Follow the installer instructions.

2. Install avrdude:

- Download **avrdude** from AVRDUDE Official Repository.
- Extract the files and add the bin folder to your **System PATH**.

3. Install CMake:

- Download CMake from CMake Official Site.
- Run the installer and ensure **"Add CMake to PATH"** is selected.

4. Verify Installation:

Open **Command Prompt** and run:

```
avr-gcc --version
avrdude -v
cmake --version
```

Linux (Debian/Ubuntu-based)

Open a **terminal** and follow these steps:

1. Update Your System:

```
sudo apt update
sudo apt upgrade
```

2. Install AVR Toolchain:

```
sudo apt install gcc-avr binutils-avr avr-libc avrdude
```

3. Install CMake:

```
sudo apt install cmake
```

4. Verify Installation:

```
avr-gcc --version
avrdude -v
cmake --version
```

Linux (Arch-based)

For Arch Linux users:

```
sudo pacman -S avr-gcc avr-libc avrdude cmake
```

Verify with:

```
avr-gcc --version
avrdude -v
cmake --version
```

5.3 Installing Ghidra

Ghidra is a sophisticated reverse engineering tool designed for analyzing binaries, including AVR firmware. It requires **Java 11 or higher**.

macOS, Windows, Linux

1. Install Java Development Kit (JDK):

Ensure Java is installed:

```
java -version
```

If not, download and install **OpenJDK 11+**:

- AdoptOpenJDK
- Select version 11 or higher.

Verify installation:

```
java -version
```

2. Download Ghidra

- Visit the **official Ghidra site:** https://ghidra-sre.org/
- Download the latest stable version.

3. Install Ghidra

macOS & Linux:

1.Extract the downloaded archive:

```
tar -xzf ghidra_<version>.zip
cd ghidra_<version>
```

2.Launch Ghidra:

```
./ghidraRun
```

Windows:

1. Extract the archive using tools like **7-Zip**.
2. Open the extracted folder.

3. Run:

```
ghidraRun.bat
```

4. Create a Desktop Shortcut (Optional)

macOS & Linux:

Create a symbolic link:

```
sudo ln -s $(pwd)/ghidraRun /usr/local/bin/ghidra
```

Now you can launch Ghidra with:

```
ghidra
```

Windows:

1. Right-click ghidraRun.bat.
2. Select **Create Shortcut**.
3. Move the shortcut to your Desktop.

5. Verify Ghidra Installation

Run Ghidra:

```
ghidra
```

Or on Windows, double-click ghidraRun.bat.

If the GUI launches successfully, your installation is complete.

5.5 Installing Visual Studio Code (VSCode) on macOS, Windows, and Linux

Visual Studio Code (VSCode) is a powerful, lightweight, and highly extensible code editor. It supports a wide range of programming languages, including **Assembly, C, and Python**, and integrates well with the AVR toolchain. In this section, we'll cover the installation process for **macOS**, **Windows**, and **Linux**, as well as essential extensions for embedded development and reverse engineering.

5.5.1 Why VSCode for Embedded Development?

- **Cross-Platform:** Runs on macOS, Windows, and Linux.
- **Integrated Terminal:** Direct access to system terminals for compiling and flashing firmware.
- **Extension Support:** Tools like **C/C++ IntelliSense**, **AVR Assembly Syntax Highlighting**, and **PlatformIO** enhance productivity.
- **Debugging Tools:** Supports advanced debugging with **GDB** and external debuggers.

5.5.2 Installing VSCode

macOS

1.**Download VSCode:**
Visit the official VSCode website and download the **macOS version**.
2.**Install VSCode:**

- Open the downloaded .zip file.
- Drag **Visual Studio Code.app** to the **Applications** folder.

3.**Add VSCode to PATH (Optional):**
Open **Terminal** and run:

```
export PATH="$PATH:/Applications/Visual Studio
Code.app/Contents/Resources/app/bin"
```

Verify with:

```
code --version
```

Windows

1.**Download VSCode:**
Go to the official VSCode website and download the **Windows Installer** (.exe).

2 Run the Installer:

- Select **"Add to PATH"** during installation.

- Enable **"Register Code as an editor for supported file types."**

3 Verify Installation:
Open **Command Prompt** and type:

```
code --version
```

4.**Launch VSCode:**
Open from the **Start Menu** or run code in **Command Prompt**.

Linux

Debian/Ubuntu-based Distros:

1.**Add Microsoft Repository:**

```
sudo apt update
sudo apt install wget gpg
wget -qO-
https://packages.microsoft.com/keys/microsoft.asc |
gpg --dearmor > packages.microsoft.gpg
sudo install -o root -g root -m 644
packages.microsoft.gpg /usr/share/keyrings/
sudo sh -c 'echo "deb [arch=amd64
signed-by=/usr/share/keyrings/packages.microsoft.gpg]
https://packages.microsoft.com/repos/vscode stable
main" > /etc/apt/sources.list.d/vscode.list'
sudo apt install apt-transport-https
sudo apt update
sudo apt install code
```

2.**Verify Installation:**

```
code --version
```

3.**Launch VSCode:**

Run from the terminal:

```
code
```

Arch-based Distros:

```
sudo pacman -S code
```

5.6 Testing the Toolchain

Simple “Blink” Firmware

1.Create a new file named blink.c:

```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
    DDRB |= (1 << PB5); // Set PB5 (LED on Nano) as
    output

    while (1) {
        PORTB ^= (1 << PB5); // Toggle LED
        _delay_ms(500);
    }
}
```

2.Compile:

```
avr-gcc -mmcu=atmega328p -Os -o blink.elf blink.c
avr-objcopy -O ihex blink.elf blink.hex
```

3.Upload to Arduino Nano:

```
avrdude -c arduino -p m328p -P /dev/ttyUSB0 -b 115200
-U flash:w:blink.hex:i
```

4.Verify the LED on pin **D13** blinks.

5.7 Conclusion

Your development environment is now ready for both **embedded programming** and **reverse engineering**. In the next chapter, we'll dive into **GPIO pin programming on the ATmega328P** to interact with hardware peripherals.

6

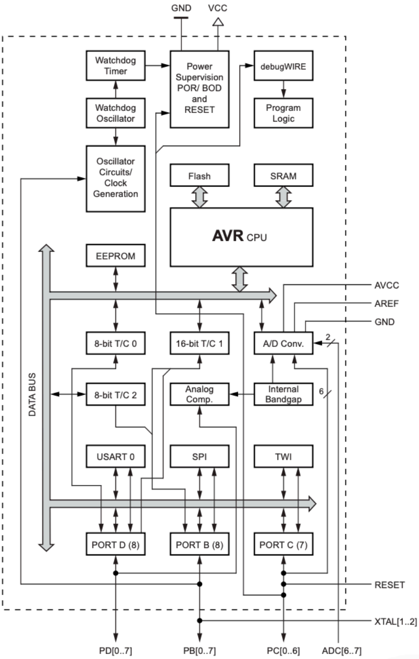
Chapter 6: ATmega328P Datasheet

The **ATmega328P** microcontroller, the brain behind the **Arduino Nano**, is a powerful yet compact 8-bit MCU designed for efficiency, versatility, and precision. In this chapter, we will explore its **architecture**, **memory layout**, and **key registers** as outlined in the **datasheet**. This information is vital for programming, debugging, and reverse engineering firmware.

6.1 Block Diagram Overview

2.1 Block Diagram

Figure 2-1. Block Diagram



The **Block Diagram** of the ATmega328P (Figure 2-1) provides a bird's-eye view of the microcontroller's architecture. It illustrates how different components interact, highlighting the **Harvard Architecture** design, where program and data memory are accessed via separate buses.

Key Components in the Block Diagram

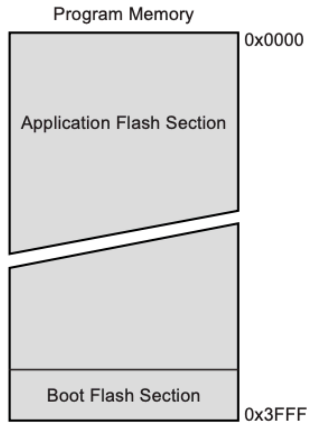
1. **AVR CPU:** The heart of the microcontroller, responsible for executing instructions and managing peripherals.
2. **Program Memory (Flash):** Non-volatile memory for

firmware storage.
3. **Data Memory (SRAM):** Volatile memory for runtime data and stack operations.
4. **EEPROM:** Non-volatile memory for configuration and persistent data.
5. **Watchdog Timer:** Ensures the system can recover from unresponsive states.
6. **USART, SPI, TWI Interfaces:** Communication protocols for external peripherals.
7. **GPIO Ports (PORTB, PORTC, PORTD):** Digital input/output ports.
8. **Timers (8-bit and 16-bit):** Time-based operations for delays, PWM, and counters.
9. **Analog-to-Digital Converter (ADC):** Converts analog signals to digital.
10. **Power Supervision and Reset Logic:** Ensures proper startup and operation under varying power conditions.

The diagram also highlights the **DATA BUS**, which serves as the backbone for communication between the CPU and peripherals.

6.2 Program Memory (Flash)

Figure 7-1. Program Memory Map ATmega328P



The **Program Memory Map** (Figure 7-1) outlines the structure of the **Flash Memory** in the ATmega328P. Flash memory is used for **storing firmware code** and is **non-volatile**, meaning it retains its contents after power-off.

Flash Memory Overview

- **Total Size:** 32 KB
- **Address Range:** 0x0000 – 0x3FFF

Memory Segments:

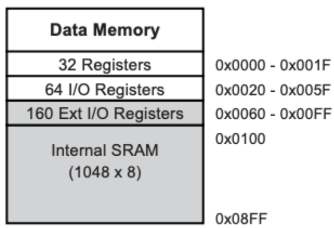
1. **Application Flash Section:** Stores the main program code.
2. **Boot Flash Section:** Reserved for bootloader code, allowing self-programming and firmware updates.

Key Points:

- At **0x0000**, the **Reset Vector** is located. This is the address where the CPU starts executing after reset.
- Interrupt vectors are mapped sequentially after the reset vector.

6.3 Data Memory (SRAM)

Figure 7-2. Data Memory Map



The **Data Memory Map** (Figure 7-2) provides insight into how **SRAM** is organized. SRAM is used for **storing variables**, **register states**, and the **stack** during runtime.

Memory Segments:

1.**General Purpose Registers (0x0000 – 0x001F)**

- 32 registers (R0–R31) are directly accessible for arithmetic, logic, and data transfer instructions.

2.**I/O Registers (0x0020 – 0x005F)**

- 64 registers control peripherals like timers, GPIO ports, and ADC.

3.**Extended I/O Registers (0x0060 – 0x00FF)**

- 160 registers provide additional functionality for complex peripherals.

4.**Internal SRAM (0x0100 – 0x08FF)**

- 1 KB of SRAM is used for storing variables and the program stack.

Key Notes:

- SRAM is **volatile** and is cleared after each power cycle.
- The stack grows **downwards** from the highest available SRAM address.

6.4 EEPROM (Electrically Erasable Programmable Read-Only Memory)

- **Size:** 1 KB
- **Type:** Non-volatile
- **Purpose:** Store calibration constants, configuration data, and small amounts of persistent information.
- **Access:** Requires specific registers (EEAR, EEDR, EECR) and follows a **write cycle delay**.

Accessing EEPROM in Assembly:

- EEAR: EEPROM Address Register
- EEDR: EEPROM Data Register
- EECR: EEPROM Control Register

6.5 Register Summary

30. Register Summary (Continued)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0x68)	PCICR	–	–	–	–	–	PCIE2	PCIE1	PCIE0	
(0x67)	Reserved	–	–	–	–	–	–	–	–	
(0x66)	OSCCAL	Oscillator calibration register								32
(0x65)	Reserved	–	–	–	–	–	–	–	–	
(0x64)	PRR	PRTWI	PRTIM2	PRTIM0	–	PRTIM1	PRSPI	PRUSAR0	PRADC	36
(0x63)	Reserved	–	–	–	–	–	–	–	–	
(0x62)	Reserved	–	–	–	–	–	–	–	–	
(0x61)	CLKPR	CLKPCE	–	–	–	CLKPS3	CLKPS2	CLKPS1	CLKPS0	33
(0x60)	WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	47
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C	10
0x3E (0x5E)	SPH	–	–	–	–	–	(SP10)	SP9	SP8	13
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	13
0x3C (0x5C)	Reserved	–	–	–	–	–	–	–	–	
0x3B (0x5B)	Reserved	–	–	–	–	–	–	–	–	
0x3A (0x5A)	Reserved	–	–	–	–	–	–	–	–	
0x39 (0x59)	Reserved	–	–	–	–	–	–	–	–	
0x38 (0x58)	Reserved	–	–	–	–	–	–	–	–	
0x37 (0x57)	SPMCSR	SPMIE	(RWWSB)	–	(RWWSRE)	BLBSET	PGWRT	PGERS	SELFPRGN	239
0x36 (0x56)	Reserved	–	–	–	–	–	–	–	–	
0x35 (0x55)	MCUCR	–	BODS	BODSE	PUD	–	–	IVSEL	IVCE	38/52/72
0x34 (0x54)	MCUSR	–	–	–	–	WDRF	BORF	EXTRF	PORF	46
0x33 (0x53)	SMCR	–	–	–	–	SM2	SM1	SM0	SE	35
0x32 (0x52)	Reserved	–	–	–	–	–	–	–	–	
0x31 (0x51)	Reserved	–	–	–	–	–	–	–	–	
0x30 (0x50)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	203
0x2F (0x4F)	Reserved	–	–	–	–	–	–	–	–	
0x2E (0x4E)	SPDR	SPI data register								142
0x2D (0x4D)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X	141
0x2C (0x4C)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	140
0x2B (0x4B)	GPIOR2	General purpose I/O register 2								23
0x2A (0x4A)	GPIOR1	General purpose I/O register 1								23
0x29 (0x49)	Reserved	–	–	–	–	–	–	–	–	
0x28 (0x48)	OCR0B	Timer/Counter0 output compare register B								
0x27 (0x47)	OCR0A	Timer/Counter0 output compare register A								
0x26 (0x46)	TCNT0	Timer/Counter0 (8-bit)								
0x25 (0x45)	TCCR0B	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	
0x24 (0x44)	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	
0x23 (0x43)	GTCCR	TSM	–	–	–	–	–	PSRASY	PSRSYNC	115/134

The **Register Summary Table** (Figure 30) provides an overview of the key registers in the ATmega328P, including their addresses, bit assignments, and functionality.

Key Register Highlights:

1. Status Register (SREG) – 0x5F

- Controls and reflects CPU status flags (I, T, H, S, V, N, Z, C).

2. Stack Pointer (SPH, SPL) – 0x3E, 0x3D

- Points to the current position of the stack.

3. GPIO Registers:

- **PORTB, PORTC, PORTD:** Control digital I/O pins.
- **DDRB, DDRC, DDRD:** Configure I/O direction (input/output).
- **PINB, PINC, PIND:** Read the state of input pins.

4. Timer Registers:

- **TCCR0A, TCCR0B:** Timer/Counter Control for Timer0.
- **OCR0A, OCR0B:** Output Compare Registers.

5. USART Registers:

- **UDR0:** USART Data Register.
- **UCSR0A, UCSR0B, UCSR0C:** USART Control and Status Registers.

6.6 Interrupt Vectors

The ATmega328P supports multiple interrupt sources, each linked to a specific **vector address** in **Flash memory**.

Interrupt Vector Table Overview:

- **RESET:** Address 0x0000
- **External Interrupt 0 (INT0):** 0x0002
- **Timer Overflow (TIMER0_OVF):** 0x000A

Interrupts allow the CPU to respond to hardware events asynchronously, improving efficiency and responsiveness

6.7 Peripherals

1. Timers

- **Timer0 (8-bit):** Basic time-keeping, delays, PWM.
- **Timer1 (16-bit):** Precise timing, waveform generation.
- **Timer2 (8-bit):** Additional timing and PWM capabilities.

2. Communication Interfaces

- **USART:** Serial communication with external devices.
- **SPI:** High-speed synchronous communication.
- **TWI (I^2C):** Communication with sensors and peripherals.

3. ADC (Analog-to-Digital Converter)

- **10-bit resolution**
- **8 channels** for analog input.

6.8 Conclusion

The ATmega328P microcontroller is a sophisticated system with distinct memory spaces and powerful peripherals. Understanding its **block diagram**, **memory maps**, and **registers** is crucial for effective programming and reverse engineering. In the next chapter, we will dive into **GPIO Programming with Assembly on the Arduino Nano**.

7

Chapter 7: Blink Driver in C

7.1 Introduction

In this chapter, we will transition from **AVR Assembly** to **AVR C programming** to blink an **LED connected to PB5 (Pin 13)** on the **ATmega328P microcontroller (Arduino Nano)**. Programming in **C** for microcontrollers provides a higher level of abstraction while maintaining fine control over the hardware.

By the end of this chapter, you will:

- Understand how to configure **microcontroller pins in C**.
- Learn how to **control an LED using C code**.
- Implement **software delays using the AVR libc library** (<util/delay.h>).

7.2 Overview of the Program

This program will:

1. **Configure PB5 as an output pin** using the DDRB register.
2. **Toggle the LED ON and OFF** using the PORTB register.
3. Use _delay_ms from **AVR libc** to implement a **1-second delay** between toggles.
4. Loop indefinitely to keep the LED blinking.

This example uses:

- **AVR-GCC:** Compiler to generate machine code.
- **AVRDUDE:** Tool to upload the compiled code to the microcontroller.

7.3 Program Listing

```
//////////////////////////////////////////////////////////////////
// Project: ATmega328P Blink Driver
//////////////////////////////////////////////////////////////////
// Author: Kevin Thomas
// E-Mail: ket189@pitt.edu
// Version: 1.0
// Date: 12/26/24
// Target Device: ATmega328P (Arduino Nano)
// Clock Frequency: 16 MHz
// Toolchain: AVR-GCC, AVRDUDE
// License: Apache License 2.0
```

```
// Description: This program toggles the onboard LED
connected to PB5
//              (Pin 13) on the Arduino Nano at
1-second intervals
//              using C. The delay is implemented
using the AVR libc
//              library.
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// INCLUDES
/////////////////////////////////////////////////////////////////
#include <avr/io.h>
#include <util/delay.h>

/////////////////////////////////////////////////////////////////
// DEFINES, MACROS, CONSTANTS
/////////////////////////////////////////////////////////////////
#ifndef F_CPU
#define F_CPU 16000000UL          // define clk freq
(16 MHz)
#endif

/////////////////////////////////////////////////////////////////
// FUNCTIONS
/////////////////////////////////////////////////////////////////
int main(void) {
  DDRB |= (1 << PB5);             // set PB5 (D13) as
  output

  while (1) {
    PORTB |= (1 << PB5);          // LED on
    _delay_ms(1000);              // 1s delay
    PORTB &= ~(1 << PB5);         // LED off
    _delay_ms(1000);              // 1s delay
  }
```

```
  return 0;
}
```

7.4 Understanding the Code

7.4.1 Includes

```
#include <avr/io.h>
#include <util/delay.h>
```

- <avr/io.h>**:** Provides definitions for the **microcontroller's hardware registers**.
- Example: DDRB, PORTB, and PB5 are defined here.
- <util/delay.h>**:** Part of **AVR libc** and provides **_delay_ms()‘** for time delays.
- Relies on the **F_CPU** macro to calculate timing correctly.

7.4.2 Main Function

```
int main(void) {
```

The main function is the **entry point** of the program.

- In embedded systems, the main function typically **does not return**, but we include return 0; for compliance with standard C conventions.

7.4.3 Configuring PB5 as an Output Pin

```
DDRB |= (1 << PB5);
```

- DDRB**:** The **Data Direction Register for PORTB**.
- 0: Configure pin as **input**.
- 1: Configure pin as **output**.
- (1 « PB5)**:** The expression sets **Bit 5** of DDRB to 1.
- |= **(OR Equal Operator):** Ensures other bits in DDRB remain unchanged.
- **Result:** PB5 (D13) is configured as an **output pin**.

7.4.4 Turning the LED ON

```
PORTB |= (1 << PB5);
```

- PORTB**:** The **Data Register for PORTB**.
- 0: Drive pin **LOW** (0V).
- 1: Drive pin **HIGH** (5V).
- (1 « PB5)**:** Sets **Bit 5** in PORTB to 1.
- |= **(OR Equal Operator):** Only modifies PB5 without affecting other bits.
- **Result:** PB5 (D13) is set **HIGH**, and the LED turns **ON**.

7.4.5 Delay Using _delay_ms()

```
_delay_ms(1000);
```

- **_delay_ms:** Blocks execution for a specified number of milliseconds.
- 1000**:** Specifies a **1-second delay**.

Important Note:

- The F_CPU macro (e.g., #define F_CPU 16000000UL) must be defined during compilation to ensure accurate timing.

7.4.6 Turning the LED OFF

```
PORTB &= ~(1 << PB5);
```

- ~(1 « PB5)**:** Inverts the **PB5** bit, clearing it while preserving others.
- &= **(AND Equal Operator):** Clears PB5 without affecting other bits.
- **Result:** PB5 (D13) is set **LOW**, and the LED turns **OFF**.

7.4.7 Infinite Loop

```
while (1) { ... }
```

7.4.8 Program Termination

```
return 0;
```

- Included for **standard C compliance**.
- Embedded systems typically do not return from main.

7.5 Program Flow Summary

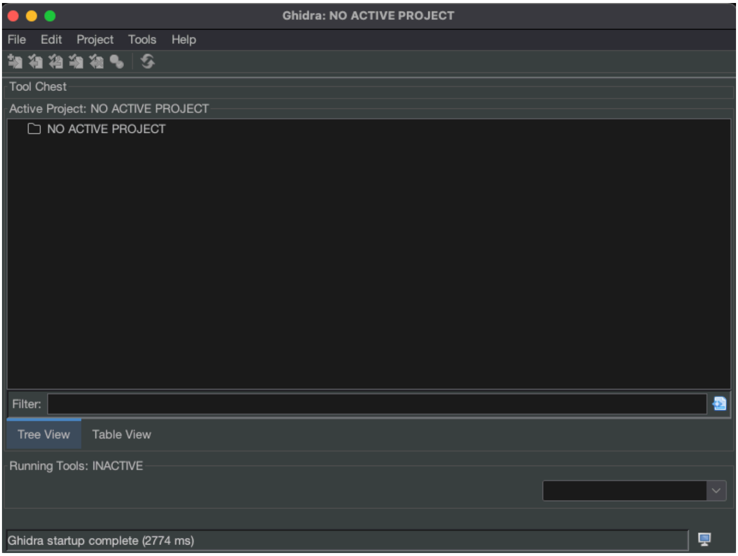
1. **Initialization:** PB5 is set as an **output pin**.
2. **Loop Starts:**

- Turn LED **ON**.
- Wait for **1 second**.
- Turn LED **OFF**.
- Wait for **1 second**.

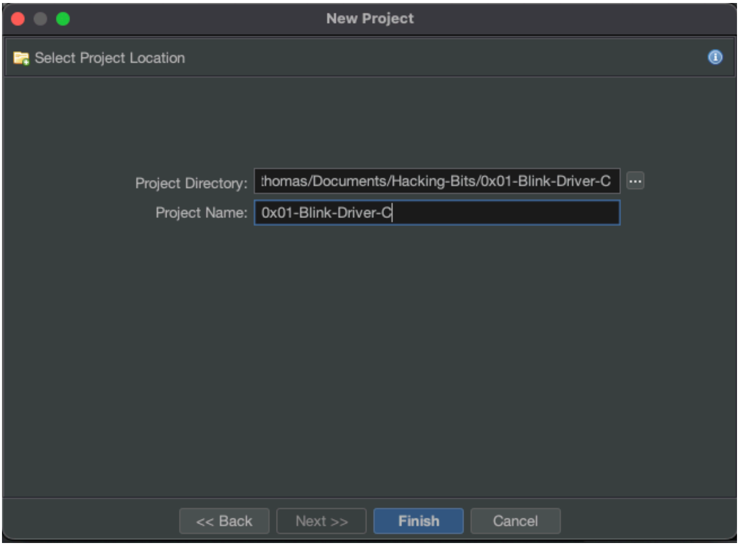
Repeat Indefinitely: The while(1) loop ensures the blinking continues.

7.6 Reverse Engineering in Ghidra

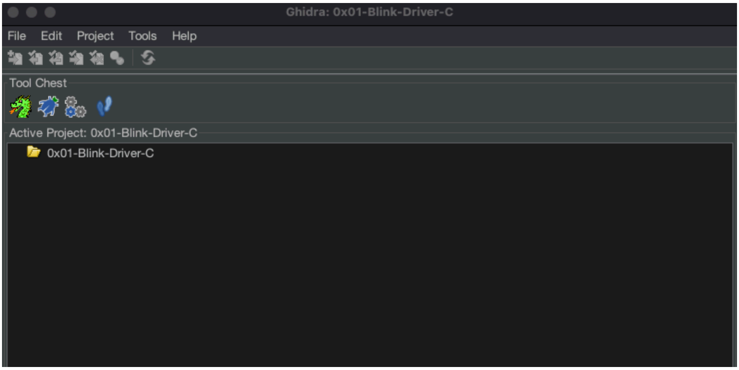
open Ghidra



File - New Project - Non-Shared Project - Next »

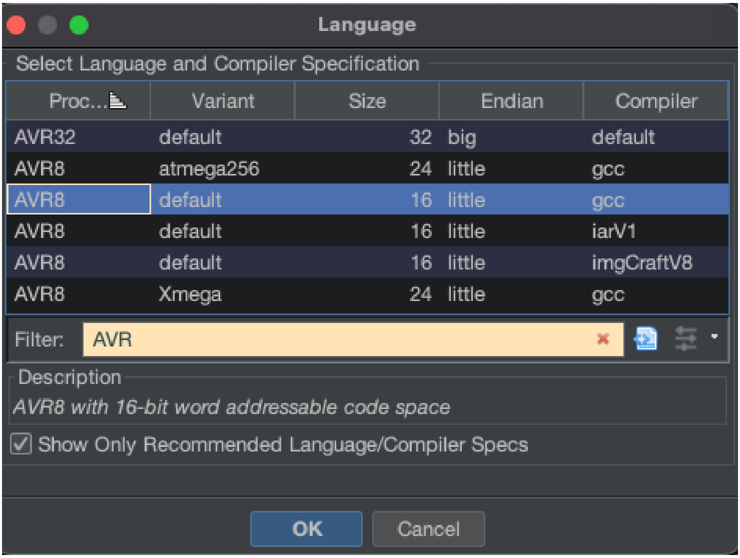


Finish

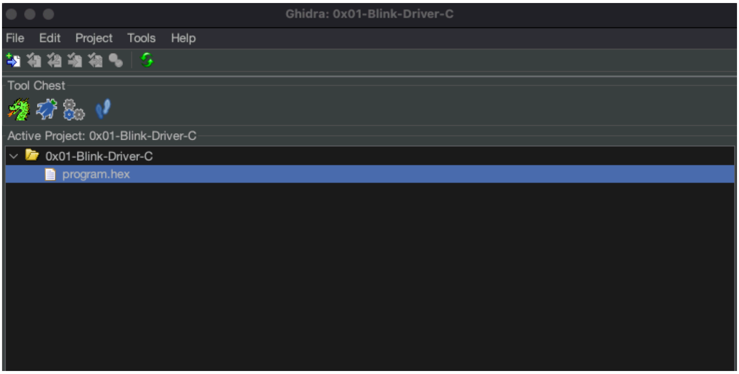


Drag **program.hex** into folder

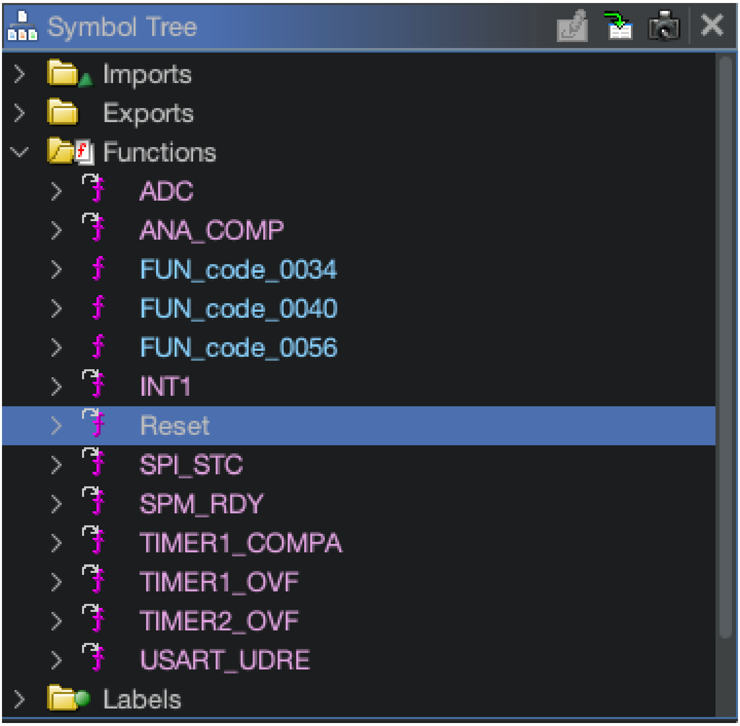
Filter: AVR - AVR8 default 16 little gcc



double-click on program.hex then auto-analyze



click on Function and select Reset



This is the Reset Handler that the bootloader will load immediately on boot. Below is the Assembler output. We then follow the **jmp** to **FUN_code_0034** which will then lead us to the call for the **main** function.

```
                    //
                    // code
                    // Generated by Intel Hex
                    // code:0000-code:0057.1
                    //

                    **************************************************
                    *                 THUNK FUNCTION                 *
                    **************************************************
                    thunk undefined Reset()
                      Thunked-Function: FUN_code_0034
         undefined    R24:1        <RETURN>
                    INT0 (code:0000+2)                  XREF[2,1… Entry Point(*),
                    Reset                                         Reset:003e(T),
                                                                  Reset:003e(j),
                                                                  Entry Point(*)
   code:0000 0c 94      jmp       FUN_code_0034
             34 00
```

Here we see the Assembler view and then the Decompile view (pseudo-code in C).

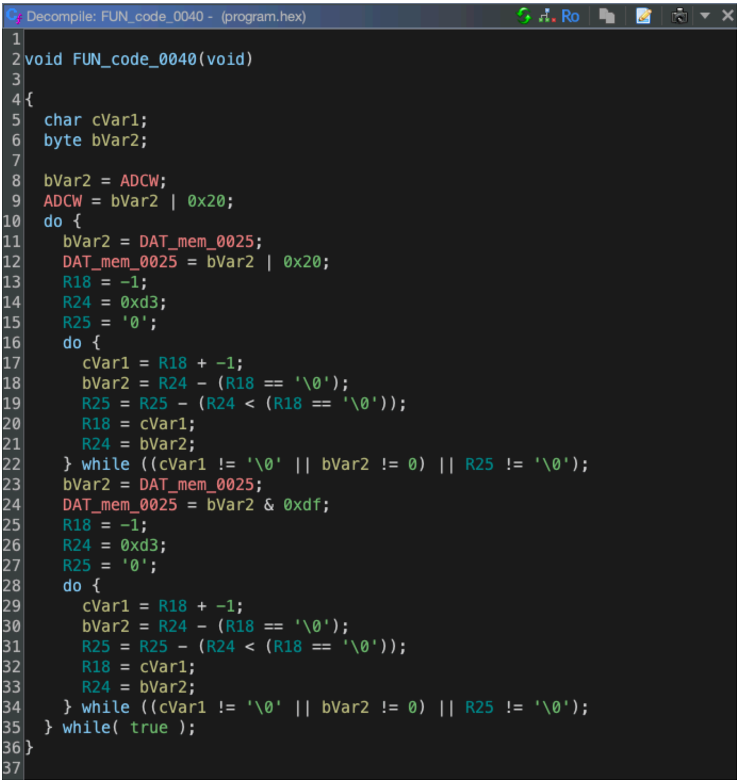
```
                    **************************************************
                    *                    FUNCTION                    *
                    **************************************************
                    undefined FUN_code_0034()
         undefined    R24:1        <RETURN>
                    FUN_code_0034                   XREF[1]:   Reset:0000(T),
                                                               Reset:0000(j)
   code:0034 11 24      eor       R1,R1
   code:0035 1f be      out       SREG,R1
   code:0036 cf ef      ser       Ylo
   code:0037 d8 e0      ldi       Yhi,0x8
   code:0038 de bf      out       SPH,Yhi
   code:0039 cd bf      out       SPL,Ylo
   code:003a 0e 94      call      FUN_code_0040              undefined FUN_code_0040()
             40 00                                           = ??
   code:003c 0c 94      jmp       FUN_code_0056              undefined FUN_code_0056()
             56 00
```

Decompile: FUN_code_0034 - (program.hex)

```

void FUN_code_0034(void)

{
  R1 = 0;
  SREG = 0;
  Ylo = 0xff;
  Yhi = 8;
  DAT_mem_08fe = 0x3c;
  FUN_code_0040();
  FUN_code_0056();
  return;
}

```

We then double-click on **FUN_code_0040** which will be our **main** function. The top is the Assembler view and the bottom is the Decompile view.

```
                    ************************************************
                    *                   FUNCTION                   *
                    ************************************************
                    undefined FUN_code_0040()
     undefined        R24:1        <RETURN>
                    FUN_code_0040                          XREF[1]:   FUN_code_0034:003a(c)
code:0040 25 9a         sbi        ADCW,0x5                                 = ??

                    LAB_code_0041                          XREF[1]:   code:0055(j)
code:0041 2d 9a         sbi        DAT_mem_0025,0x5                         = ??
code:0042 2f ef         ser        R18
code:0043 83 ed         ldi        R24,0xd3
code:0044 90 e3         ldi        R25,0x30

                    LAB_code_0045                          XREF[1]:   code:0048(j)
code:0045 21 50         subi       R18,0x1
code:0046 80 40         sbci       R24,0x0
code:0047 90 40         sbci       R25,0x0
code:0048 e1 f7         brbc       LAB_code_0045,Zflg
code:0049 00 c0         rjmp       LAB_code_004a

                    LAB_code_004a                          XREF[1]:   code:0049(j)
code:004a 00 00         nop
code:004b 2d 98         cbi        DAT_mem_0025,0x5                         = ??
code:004c 2f ef         ser        R18
code:004d 83 ed         ldi        R24,0xd3
code:004e 90 e3         ldi        R25,0x30

                    LAB_code_004f                          XREF[1]:   code:0052(j)
code:004f 21 50         subi       R18,0x1
code:0050 80 40         sbci       R24,0x0
code:0051 90 40         sbci       R25,0x0
code:0052 e1 f7         brbc       LAB_code_004f,Zflg
code:0053 00 c0         rjmp       LAB_code_0054

                    LAB_code_0054                          XREF[1]:   code:0053(j)
code:0054 00 00         nop
code:0055 eb cf         rjmp       LAB_code_0041
```




```
void FUN_code_0040(void)

{
  char cVar1;
  byte bVar2;

  bVar2 = ADCW;
  ADCW = bVar2 | 0x20;
  do {
    bVar2 = DAT_mem_0025;
    DAT_mem_0025 = bVar2 | 0x20;
    R18 = -1;
    R24 = 0xd3;
    R25 = '0';
    do {
      cVar1 = R18 + -1;
      bVar2 = R24 - (R18 == '\0');
      R25 = R25 - (R24 < (R18 == '\0'));
      R18 = cVar1;
      R24 = bVar2;
    } while ((cVar1 != '\0' || bVar2 != 0) || R25 != '\0');
    bVar2 = DAT_mem_0025;
    DAT_mem_0025 = bVar2 & 0xdf;
    R18 = -1;
    R24 = 0xd3;
    R25 = '0';
    do {
      cVar1 = R18 + -1;
      bVar2 = R24 - (R18 == '\0');
      R25 = R25 - (R24 < (R18 == '\0'));
      R18 = cVar1;
      R24 = bVar2;
    } while ((cVar1 != '\0' || bVar2 != 0) || R25 != '\0');
  } while( true );
}
```

Let's take a step back and re-examine our source code in C.

```
int main(void) {
  DDRB |= (1 << PB5);               // set PB5 (D13) as
  output

  while (1) {
    PORTB |= (1 << PB5);            // LED on
    _delay_ms(1000);                // 1s delay
    PORTB &= ~(1 << PB5);           // LED off
    _delay_ms(1000);                // 1s delay
```

```
  }

  return 0;
}
```

This declares the entry point of the function. In AVR assembly, **main()** is often referred to as the starting address **FUN_code_0040**. It also indicates that the return value, if any, will be stored in register **R24**. For more info on calling conventions in AVR visit https://gcc.gnu.org/wiki/avr-gcc.

```
                    ******************************************************
                    *                        FUNCTION                    *
                    ******************************************************
                    undefined FUN_code_0040()
      undefined       R24:1         <RETURN>
```

We then set **DDRB** to configure **PB5** as an output.

```
DDRB |= (1 << PB5);              // set PB5 (D13) as
output
```

sbi: ***Set Bit in I/O Register***. This instruction modifies the Data Direction Register for **PORTB** (register **DDRB**).

ADCW, **0x5** sets bit 5 of **DDRB**, enabling **PB5** as an output pin.

```
code:0040 25 9a        sbi       ADCW,0x5
code:0041 2d 9a        sbi       DAT_mem_0025,0x5
```

We then turn the LED on.

```
PORTB |= (1 << PB5);          // LED on
```

ser R18: Set all bits of register **R18** to 1.

ldi R24, 0xd3 and **ldi R25, 0x30**: Load immediate values into registers **R24** and **R25**. These represent delay constants for the **_delay_ms()** function (explained later).

```
code:0042 2f ef      ser      R18
code:0043 83 ed      ldi      R24,0xd3
code:0044 90 e3      ldi      R25,0x30
```

We then call the 1000 ms delay (1 second).

```
_delay_ms(1000);              // 1s delay
```

These instructions implement a delay loop using registers **R18**, **R24**, and **R25**:

subi R18, 0x1: Subtract 1 from **R18**.

sbci R24, 0x0: Subtract with carry from **R24**.

sbci R25, 0x0: Subtract with carry from **R25**.

brbc LAB_code_0045, Zflg: Branch back to delay loop until the zero flag (**Zflg**) is set.

We then turn off the LED.

```
PORTB &= ~(1 << PB5);         // LED off
```

cbi DAT_mem_0025, 0x5: ***Clear Bit in I/O Register***. Clears bit 5 of **PORTB**, turning off the LED.

```
code:004b 2d 98        cbi       DAT_mem_0025,0x5
```

rjmp LAB_code_0041: Jumps back to the start of the infinite loop (**while (1)**).

```
code:0053 00 c0        rjmp      LAB_code_0054
                       LAB_code_0054                             XREF[1]:   code:0053
code:0054 00 00        nop
code:0055 eb cf        rjmp      LAB_code_0041
```

8

Chapter 8: Blink Driver in Assembler

Understanding Assembler of an architecture means you can use tools like Ghidra to Reverse Engineer literally anything!

We are going to learn AVR Assembler for the 8-bit microcontroller of the ATmega128P Arduino Nano from scratch.

There are two documents that you MUST, yes I am yelling; you MUST use if you are to Reverse Engineer and that is a **datasheet** and an **instruction set manual**.

To begin with Assembler, we will create a, “blinky”, program in pure AVR Assembler. This is the, “hello world”, of Embedded Systems!

Here is the complete code to which I simply want you to do the following.

1. read it, one breath at a time, re-read it, one breath at a time
2. open up the AVR Instruction Set Manual and PHYSICALLY

LOOK UP EACH INSTRUCTION so you have an understanding of every machine instruction

```
;
==================================================================
; Project: ATmega328P Blink Driver
;
==================================================================
; Author: Kevin Thomas
; E-Mail: ket189@pitt.edu
; Version: 1.0
; Date: 12/26/24
; Target Device: ATmega328P (Arduino Nano)
; Clock Frequency: 16 MHz
; Toolchain: AVR-AS, AVRDUDE
; License: Apache License 2.0
; Description: This program toggles the onboard LED
connected to PB5
;              (Pin 13) on the Arduino Nano at
1-second intervals
;              using pure AVR Assembly. The delay is
implemented
;              using nested loops calibrated for a 16
MHz
;              clock frequency.
;
==================================================================

;
==================================================================
; SYMBOLIC DEFINITIONS
;
==================================================================
.equ     DDRB, 0x04               ; Data Direction
Register for PORTB
.equ     PORTB, 0x05              ; PORTB Data
```

```
Register
.equ     PB5, 5                    ; Pin 5 of PORTB
(D13 on Nano)

;
==========================================================================
; PROGRAM ENTRY POINT
;
==========================================================================
.global  program                   ; global label;
make avail external
.section .text                     ; start of the
.text (code) section

;
==========================================================================
; PROGRAM LOOP
;
==========================================================================
; Description: Main program loop which executes all
subroutines and
;              then repeads indefinately.
;
--------------------------------------------------------------------------
; Instructions: AVR Instruction Set Manual
;               6.87 RCALL – Relative Call to
Subroutine
;               6.90 RJMP – Relative Jump
;
==========================================================================
program:
  rcall  config_pins               ; config pins
program_loop:
  rcall  led_on                    ; turn LED on
  rcall  delay_1s                  ; wait 1 second
  rcall  led_off                   ; turn LED off
  rcall  delay_1s                  ; wait 1 second
```

```
  rjmp   program_loop             ; infinite loop

;
=============================================================================
; SUBROUTINE: config_pins
;
=============================================================================
; Description: Main configuration of pins on the
ATmega128P Arduino
;              Nano.
;
-----------------------------------------------------------------------------
; Instructions: AVR Instruction Set Manual
;               6.95 SBI - Set Bit in I/O Register
;               6.88 RET - Return from Subroutine
;
=============================================================================
config_pins:
  sbi    DDRB, PB5                ; set PB5 as output
  ret                             ; return from
  subroutine

;
=============================================================================
; SUBROUTINE: led_on
;
=============================================================================
; Description: Sets PB5 high to turn on the LED.
;
-----------------------------------------------------------------------------
; Instructions: AVR Instruction Set Manual
;               6.95 SBI - Set Bit in I/O Register
;               6.88 RET - Return from Subroutine
;
=============================================================================
led_on:
  sbi    PORTB, PB5               ; set PB5 high
```

```
  ret                             ; return from
  subroutine

;
==============================================================
; SUBROUTINE: led_off
;
==============================================================
; Description: Clears PB5 to turn off the LED.
;
--------------------------------------------------------------
; Instructions: AVR Instruction Set Manual
;               6.33 CBI – Clear Bit in I/O Register
;               6.88 RET – Return from Subroutine
;
==============================================================
led_off:
  cbi    PORTB, PB5               ; set PB5 low
  ret                             ; return from
  subroutine

;
==============================================================
; SUBROUTINE: delay_1s
;
==============================================================
; Description: A one-second delay.
;              - CPU Clock: 16 MHz
;              - 1 clock cycle = 62.5 ns
;              - total cycles for 1 second =
16,000,000
;              - nested loops create approximate
1-second delay
;
--------------------------------------------------------------
; Instructions: AVR Instruction Set Manual
;               6.69 LDI – Load Immediate
```

```
;                  6.81 NOP - No Operation
;                  6.49 DEC - Decrement
;                  6.23 BRNE - Branch if Not Equal
;                  6.88 RET - Return from Subroutine
;
=================================================================
delay_1s:
  ldi    r16, 250                  ; outer loop counter
.outer_loop:
  ldi    r17, 250                  ; middle loop
  counter
.middle_loop:
  ldi    r18, 64                   ; inner loop counter
.inner_loop:
  nop                              ; 1 cycle delay
  dec    r18                       ; decrement inner
  loop counter
  brne   .inner_loop               ; repeat if not
  zero else 2 cycles
  dec    r17                       ; decrement middle
  loop counter
  brne   .middle_loop              ; repeat if not zero
  dec    r16                       ; decrement outer
  loop counter
  brne   .outer_loop               ; repeat if not zero
  ret                              ; return from
  subroutine
```

8.1 Introduction

In this chapter, we will explore how to control an **LED connected to PB5 (Pin 13)** on an **ATmega328P microcontroller** using **AVR Assembly language**. By the end of this chapter, you will:

- Understand how to configure **input/output pins** on the microcontroller.
- Learn how to **control the state of an LED** using assembly instructions.
- Build a **software-based delay** using nested loops.
- Gain familiarity with **AVR Assembly instructions** like SBI, CBI, LDI, DEC, and BRNE.

This example runs on the **Arduino Nano**, which uses an **ATmega328P microcontroller** with a **16 MHz clock frequency**.

8.2 Overview of the Program

The goal of this program is to:

1. **Set PB5 as an output pin.**
2. **Turn the LED ON.**
3. **Wait for 1 second.**
4. **Turn the LED OFF.**
5. **Wait for another second.**
6. **Repeat indefinitely.**

This behavior is achieved through:

- **Pin Configuration:** Configuring PB5 as an output pin.
- **LED Control:** Turning the LED ON and OFF.
- **Delay Implementation:** Using nested loops for a software delay.

8.3 Hardware Overview

Pin Definitions

- **PB5 (Pin 13 on Arduino Nano):** Connected to the onboard LED.

Registers Used

1. **DDRB (Data Direction Register for PORTB):**

- Controls whether a pin is configured as **input (0)** or **output (1)**.

2. **PORTB (PORTB Data Register):**

- Controls the **HIGH/LOW state** of pins configured as output.

8.4 Symbolic Definitions

```
.equ     DDRB, 0x04                  ; Data Direction
Register for PORTB
.equ     PORTB, 0x05                 ; PORTB Data
Register
.equ     PB5, 5                      ; Pin 5 of PORTB
(D13 on Nano)
```

- .equ: Defines symbolic constants.
- DDRB: Address 0x04 configures PORTB pins as **input/out-**

put.

- PORTB**:** Address 0x05 controls the **ON/OFF state** of PORTB pins.
- PB5**:** Represents **bit 5** of PORTB, corresponding to **Pin 13**.

8.5 Program Entry Point

```
.global  program                          ; make label
'program' accessible
.section .text                            ; start of the code
section
```

- .global program**:** Makes the program label accessible to the linker.
- .section .text**:** Specifies the **code section** where the program instructions reside.

8.6 Main Program Loop

```
program:
  rcall  config_pins                      ; configure pins
  for LED control
program_loop:
  rcall  led_on                           ; turn LED on
  rcall  delay_1s                         ; wait 1 second
  rcall  led_off                          ; turn LED off
  rcall  delay_1s                         ; wait 1 second
  rjmp   program_loop                     ; repeat forever
```

Explanation:

1. rcall config_pins: Call the config_pins subroutine to set up PB5 as an output pin.
2. rcall led_on: Call the led_on subroutine to turn the LED ON.
3. rcall delay_1s: Call the delay_1s subroutine for a 1-second delay.
4. rcall led_off: Call the led_off subroutine to turn the LED OFF.
5. rjmp program_loop: Infinite loop to repeat the process.

- rcall: Calls a subroutine while saving the return address on the stack.
- rjmp: Unconditional jump back to program_loop.

8.7 Subroutine: config_pins

```
config_pins:
  sbi    DDRB, PB5                       ; set PB5 as output
  ret                                    ; return from
  subroutine
```

Explanation:

1.sbi DDRB, PB5:

- Sets **bit 5** in the **DDRB register**, configuring PB5 as an **output pin**.
- sbi: Set Bit in I/O Register. Sets a specific bit to 1.

2.ret: Return from the subroutine.

8.8 Subroutine: led_on

```
led_on:
  sbi    PORTB, PB5                ; set PB5 high (LED
  ON)
  ret                              ; return from
  subroutine
```

Explanation:

1.sbi PORTB, PB5**:**

- Sets **bit 5** in **PORTB**, turning the LED **ON**.

2.ret**:** Return from the subroutine.

8.9 Subroutine: led_off

```
led_off:
  cbi    PORTB, PB5                ; set PB5 low (LED
  OFF)
  ret                              ; return from
  subroutine
```

Explanation:

1.cbi PORTB, PB5**:**

- Clears **bit 5** in **PORTB**, turning the LED **OFF**.
- cbi**:** Clear Bit in I/O Register. Clears a specific bit to 0.

2.ret: Return from the subroutine.

8.10 Subroutine: delay_1s

```
delay_1s:
  ldi    r16, 250                     ; outer loop counter
.outer_loop:
  ldi    r17, 250                     ; middle loop
  counter
.middle_loop:
  ldi    r18, 64                      ; inner loop counter
.inner_loop:
  nop                                 ; 1 cycle delay
  dec    r18                          ; decrement inner
  loop counter
  brne   .inner_loop                  ; repeat if not zero
  dec    r17                          ; decrement middle
  loop counter
  brne   .middle_loop                 ; repeat if not zero
  dec    r16                          ; decrement outer
  loop counter
  brne   .outer_loop                  ; repeat if not zero
  ret                                 ; return from
  subroutine
```

Explanation:

1. ldi r16, 250: Load 250 into **r16** (outer loop counter).
2. ldi r17, 250: Load 250 into **r17** (middle loop counter).
3. ldi r18, 64: Load 64 into **r18** (inner loop counter).
4. nop: No operation (1 cycle delay).
5. dec: Decrease the counter.
6. brne: Repeat the loop if the counter is not zero.

The nested loops create an approximate **1-second delay** at **16 MHz**.

8.11 Summary

- **Pin Configuration:** PB5 set as an output pin.
- **LED Control:** Turned ON and OFF using sbi and cbi.
- **Delay:** Implemented via nested loops.

Key Instructions:

- sbi: Set bit in an I/O register.
- cbi: Clear bit in an I/O register.
- ldi: Load immediate value into a register.
- dec: Decrement a register value.
- brne: Branch if not equal.
- rjmp: Unconditional jump.

9

Chapter 9: IO Driver in C

9.1 Introduction

In this chapter, we will work with an external LED and button.

By the end of this chapter, you will:
- Understand how to configure microcontroller pins in C.
- Learn how to control an LED using C code.
- Implement software delays using the AVR libc library.

9.2 Overview of the Program

This program will:
- Configure PB5 as an output pin using the DDRB register.
- Toggle the LED ON and OFF using the PORTB register.
- Use '_delay_ms' from AVR libc to implement a 1-second delay between toggles.
- Loop indefinitely to keep the LED blinking.

This example uses:
- AVR-GCC: Compiler to generate machine code.
- AVRDUDE: Tool to upload the compiled code to the microcontroller.

9.3 Program Listing

```
/////////////////////////////////////////////////////////////////
// Project: ATmega328P Basic IO Driver
/////////////////////////////////////////////////////////////////
// Author: Kevin Thomas
// E-Mail: ket189@pitt.edu
// Version: 1.0
// Date: 12/27/24
// Target Device: ATmega328P (Arduino Nano)
// Clock Frequency: 16 MHz
// Toolchain: AVR-GCC, AVRDUDE
// License: Apache License 2.0
// Description: This program uses a button connected
to PD2 to ctrl
//              an external LED connected to PD5.
When the button is
//              pressed, the LED illuminates;
otherwise, it remains
//              off.
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// INCLUDES
/////////////////////////////////////////////////////////////////
#include <avr/io.h>

/////////////////////////////////////////////////////////////////
// DEFINES, MACROS, CONSTANTS
/////////////////////////////////////////////////////////////////
```

```
#ifndef F_CPU
#define F_CPU 16000000UL          // define clk freq
(16 MHz)
#endif

//////////////////////////////////////////////////////////////////////
// FUNCTIONS
//////////////////////////////////////////////////////////////////////
int main(void) {
  DDRD &= ~(1 << PD2);            // set PD2 (D1) as
  input
  PORTD |= (1 << PD2);            // enable pull-up
  resistor on PD2
  DDRD |= (1 << PD5);             // set PD5 (D5) as
  output

  while (1) {
    if (!(PIND & (1 << PD2))) {   // if button
    pressed (PD2 low)
      PORTD |= (1 << PD5);        // LED on
    } else {                      // if button not
    pressed (PD2 high)
      PORTD &= ~(1 << PD5);       // LED off
    }
  }

  return 0;
}
```

9.4 Understanding the Code

9.4.1 Includes

```
#include <avr/io.h>
```

9.4.2 Main Function

```
int main(void) {
  DDRD &= ~(1 << PD2);            // set PD2 (D1) as
  input
  PORTD |= (1 << PD2);            // enable pull-up
  resistor on PD2
  DDRD |= (1 << PD5);             // set PD5 (D5) as
  output

  while (1) {
    if (!(PIND & (1 << PD2))) {   // if button
    pressed (PD2 low)
      PORTD |= (1 << PD5);        // LED on
    } else {                      // if button not
    pressed (PD2 high)
      PORTD &= ~(1 << PD5);       // LED off
    }
  }

  return 0;
}
```

- Initializes pins using config_pins.
- Enters an infinite loop to check the button state and control the LED.

9.4.3 LED Control Functions & Checking Button State

```
if (!(PIND & (1 << PD2))) {   // if button pressed
(PD2 low)
      PORTD |= (1 << PD5);        // LED on
    } else {                      // if button not
    pressed (PD2 high)
      PORTD &= ~(1 << PD5);       // LED off
    }
```

- **Turning LED ON**: Sets the bit for LED_PIN in PORTD.
- **Turning LED OFF**: Clears the bit for LED_PIN in PORTD.

Read Button State: Uses PIND to check the state of BUTTON_PIN.

- **Condition**: If the button is pressed (low), call led_on; otherwise, call led_off.

9.5 Program Flow Summary

1. **Initialization**:

- Set up button pin as input with pull-up resistor.
- Configure led pin as output.

1. **Main Loop**:

- Continuously read the button state.
- Turn the LED ON or OFF based on the button's state.

This program demonstrates basic I/O handling and introduces key concepts for working with AVR microcontrollers in C.

Section 9.6: Reverse Engineering the Main Function

Objective: Understanding the behavior of the main program and its initialization process, focusing on control flow, register utilization, and peripheral interactions.

1. Initialization of Registers and Peripherals

The Reset function (or the entry point) performs hardware initialization and stack pointer setup:

- **Code Analysis**:
- eor R1, R1: Clears register R1 by XOR-ing it with itself.
- out SREG, R1: Resets the Status Register.
- ldi Ylo, 0xFF and ldi Yhi, 0x08: Sets the stack pointer to the top of the SRAM. These values represent the high and low bytes of the SRAM end address.
- out SPH, Yhi and out SPL, Ylo: Updates the stack pointer registers.
- **Explanation**:

These instructions prepare the system for proper operation by resetting critical registers and initializing the stack pointer, a standard AVR startup sequence.

```
                                //
                                // code
                                // Generated by Intel Hex
                                // code:0000-code:004a.1
                                //

                                ************************************************
                                *                THUNK FUNCTION                *
                                ************************************************
                                thunk undefined Reset()
                                  Thunked-Function: FUN_code_0034
            undefined             R24:1          <RETURN>
                                INT0 (code:0000+2)                     XREF[2,1… Entry Point(*),
                                Reset                                            Reset:003e(T),
                                                                                 Reset:003e(j),
                                                                                 Entry Point(*)
       code:0000 0c 94          jmp         FUN_code_0034
                 34 00
```

```
                         ************************************************
                         *                   FUNCTION                   *
                         ************************************************
                         undefined FUN_code_0034()
          undefined        R24:1         <RETURN>
                         FUN_code_0034                        XREF[1]:   Reset:0000(T),
                                                                         Reset:0000(j)
code:0034 11 24        eor       R1,R1
code:0035 1f be        out       SREG,R1
code:0036 cf ef        ser       Ylo
code:0037 d8 e0        ldi       Yhi,0x8
code:0038 de bf        out       SPH,Yhi
code:0039 cd bf        out       SPL,Ylo
code:003a 0e 94        call      FUN_code_0040                          undefined FUN_code_0040()
          40 00                                                         = ??
code:003c 0c 94        jmp       FUN_code_0049                          undefined FUN_code_0049()
          49 00
```

Decompile: Reset - (program.hex)

```

void Reset(void)

{
  R1 = 0;
  SREG = 0;
  Ylo = 0xff;
  Yhi = 8;
  DAT_mem_08fe = 0x3c;
  FUN_code_0040();
  FUN_code_0049();
  return;
}

```

Decompile: FUN_code_0034 - (program.hex)

```
void FUN_code_0034(void)

{
  R1 = 0;
  SREG = 0;
  Ylo = 0xff;
  Yhi = 8;
  DAT_mem_08fe = 0x3c;
  FUN_code_0040();
  FUN_code_0049();
  return;
}

```

2. Function Analysis: FUN_code_0040

Purpose: Configures the USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) and processes data in a loop.

- **Key Code Observations**:
- **Register Usage**:
- UCSRB and UCSRA are manipulated to enable and configure USART operations.
- UBRRL checks data availability via polling.
- Bits are set/cleared using sbi and cbi to enable specific USART modes or transmit data.
- **Logical Flow**:
- Polling is implemented using:

```
                     ************************************************
                     *                    FUNCTION                  *
                     ************************************************
                     undefined FUN_code_0040()
      undefined        R24:1          <RETURN>
                     FUN_code_0040                        XREF[1]:   FUN_code_0034:003a(c)
code:0040 52 98         cbi        UCSR0B,0x2                              = ??
code:0041 5a 9a         sbi        UCSR0A,0x2                              = ??
code:0042 55 9a         sbi        UCSR0B,0x5                              = ??

                     LAB_code_0043                        XREF[2]:   code:0046(j),
                                                                     code:0048(j)
code:0043 4a 99         sbic       UBRR0L,0x2                              = ??
code:0044 02 c0         rjmp       LAB_code_0047
code:0045 5d 9a         sbi        UCSR0A,0x5                              = ??
code:0046 fc cf         rjmp       LAB_code_0043

                     LAB_code_0047                        XREF[1]:   code:0044(j)
code:0047 5d 98         cbi        UCSR0A,0x5                              = ??
code:0048 fa cf         rjmp       LAB_code_0043
```

Here, the program waits until the bit 0x2 in UBRRL is set, indicating data readiness.

- After data is processed, the control returns to check for new data.
- **Explanation**:
- This function demonstrates how USART is configured and used for basic serial communication. The main loop continuously monitors the data register for new input and processes it accordingly.

3. LED Control Logic in Main Program

Functionality:

- The main program toggles LEDs based on the state of PD2, an input pin connected to a button.
- The logic is implemented using bit manipulation:
- PIND & (1 « PD2) checks the button's state.
- PORTD |= (1 « PD5) turns the LED on.

- PORTD &= ~(1 « PD5) turns the LED off.

Observations:

- **Efficiency**:
- Minimal use of conditional branching (if statements).
- Direct register manipulation ensures fast response to button presses.
- **Port Setup**:
- DDRD &= ~(1 « PD2) configures PD2 as an input.
- PORTD |= (1 « PD2) enables the pull-up resistor for PD2.

Decompile: FUN_code_0040 - (program.hex)

```c

void FUN_code_0040(void)

{
  byte bVar1;

  bVar1 = UCSR0B;
  UCSR0B = bVar1 & 0xfb;
  bVar1 = UCSR0A;
  UCSR0A = bVar1 | 4;
  bVar1 = UCSR0B;
  UCSR0B = bVar1 | 0x20;
  do {
    while (bVar1 = UBRR0L, (bVar1 & 4) != 0) {
      bVar1 = UCSR0A;
      UCSR0A = bVar1 & 0xdf;
    }
    bVar1 = UCSR0A;
    UCSR0A = bVar1 | 0x20;
  } while( true );
}

```

4. Reverse Engineering Observations

- **Flow Analysis**:
- The program initializes hardware and enters an infinite loop,

reflecting typical embedded system design.
- Peripheral configurations are handled before entering the main logic.
- **Code Simplicity**:
- The structure mirrors the C code closely, with direct register assignments replacing high-level constructs.
- The translated assembly adheres to AVR standards, ensuring compatibility.

5. USART Functionality and LED Indication

- The USART loop (FUN_code_0040) could be paired with LED indicators:
- **Concept**:
- Turn on PD5 when data is being processed.
- Turn off PD5 when no data is available.
- **Integration**:
- Combine USART logic with LED control to provide visual feedback for serial communication activity.

6. Improvements and Error Handling

- Add timeouts or error flags in the USART loop to handle edge cases (e.g., no data or buffer overflows).
- Implement a secondary LED (e.g., PD6) for error states.

10

Chapter 10: STUXNET

10.1 Introduction

In this chapter, we introduce **STUXNET**, an advanced embedded application that controls a servo motor and LED indicators using an ATmega328P microcontroller. By the end of this chapter, you will:

- Understand how to generate a ~50Hz PWM signal using Timer1.
- Learn how to sweep a servo motor from 0° to 180° and back in 1° increments.
- Configure and control external LEDs based on compile-time delay settings.
- Gain insight into how compile-time conditions can dictate peripheral behavior, mimicking reactor control logic similar to that used at the Natanz Nuclear Facility.

10.2 Overview of the Program

This program performs the following tasks:

- Configures Timer1 in Fast PWM mode on PB1 (Arduino Pin 9) to produce a ~50Hz signal.
- Drives a servo motor by sweeping its position from 0° to 180° and then back to 0°.
- Controls two LEDs on PD5 and PD6 using compile-time defined constants for sweep delays.
- Uses conditional compilation to determine LED behavior: if both **SWEEP_DELAY_UP** and **SWEEP_DELAY_DOWN** equal 5, LED_D5 is turned off and LED_D6 is turned on; otherwise, the opposite LED configuration is used.
- Enters an infinite loop to continuously update the servo position and maintain the LED state.

This example uses:

- **AVR-GCC** as the compiler.
- **AVRDUDE** for programming the ATmega328P.

10.3 Program Listing

```
////////////////////////////////////////////////////////////////////////////
// Project: ATmega328P STUXNET
////////////////////////////////////////////////////////////////////////////
// Author: Kevin Thomas
// E-Mail: ket189@pitt.edu
// Version: 1.0
```

```
// Date: 12/27/24
// Target Device: ATmega328P (Arduino Nano)
// Clock Frequency: 16 MHz
// Toolchain: AVR-GCC, AVRDUDE
// License: Apache License 2.0
// Description: This program generates a ~50Hz PWM
signal on PB1
//                (Pin 9) to control a servo. The servo
sweeps from
//                0° to 180° and back to 0° in 1°
steps. Additionally,
//                it controls LEDs on D5 and D6 based
on sweep delays.
//                If both SWEEP_DELAY_UP and
SWEEP_DELAY_DOWN are 5,
//                it turns off LED at D6 and turns on
LED at D5.
//                Otherwise, it turns off LED at D5 and
turns on LED at
//                D6. This program mimics the basic
functionality of
//                the reactors at the Natanz Nuclear
Facility.
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// INCLUDES
//////////////////////////////////////////////////////////////////
#include <avr/io.h>
#include <util/delay.h>

//////////////////////////////////////////////////////////////////
// DEFINES, MACROS, CONSTANTS
//////////////////////////////////////////////////////////////////
#ifndef F_CPU
#define F_CPU 16000000UL          // define clk freq
(16 MHz)
```

```
#endif
#define SERVO_MIN 2000              // minimum pulse
width (1ms)
#define SERVO_MAX 4000              // maximum pulse
width (2ms)
#define SERVO_STEP 11               // step size each
degree (11 ticks)
#define SWEEP_DELAY_UP 5            // delay in ms for
upward sweep
#define SWEEP_DELAY_DOWN 5          // delay in ms for
downward sweep
#define LED_D5 PD5                  // Arduino Digital
Pin 5
#define LED_D6 PD6                  // Arduino Digital
Pin 6

/////////////////////////////////////////////////////////////////
// FUNCTION DECLARATIONS
/////////////////////////////////////////////////////////////////
static void timer1_init(void);
static void servo_write(uint8_t pos);

/////////////////////////////////////////////////////////////////
// FUNCTIONS
/////////////////////////////////////////////////////////////////
int main(void) {
  timer1_init();                    // initialize
  Timer1 for PWM

  // control LEDs based on sweep delays
  #if (SWEEP_DELAY_UP == 5) && (SWEEP_DELAY_DOWN == 5)
    PORTD &= ~(1 << LED_D5);        // LED off at D5
    PORTD |= (1 << LED_D6);         // LED on at D6
  #else
    PORTD &= ~(1 << LED_D6);        // LED off at D6
    PORTD |= (1 << LED_D5);         // LED on at D5
  #endif
```

```
  while (1) {
    // sweep up from 0° to 180°
    for (uint8_t pos = 0; pos <= 180; pos++) {
      servo_write(pos);           // set servo
      position
      _delay_ms(SWEEP_DELAY_UP);
    }

    // sweep down from 180° to 0°
    for (uint8_t pos = 180; pos > 0; pos--) {
      servo_write(pos);           // set servo
      position
      _delay_ms(SWEEP_DELAY_DOWN);
    }
  }

  return 0;
}

/////////////////////////////////////////////////////////////////
// FUNCTION: timer1_init
/////////////////////////////////////////////////////////////////
// Description: Configures Timer1 for Fast PWM mode
with a frequency
//              of ~50Hz and sets PB1 (Pin 9) as the
output.
//              Also configures LEDs on D5 and D6 as
outputs.
/////////////////////////////////////////////////////////////////
static void timer1_init(void) {
  // set PB1 (OC1A) as output and set PD5 and PD6 as
  outputs
  DDRB |= (1 << PB1);
  DDRD |= (1 << LED_D5) | (1 << LED_D6);
```

```
  // configure Timer1 for Fast PWM, TOP=ICR1,
  non-inverting mode
  TCCR1A = (1 << COM1A1) | (1 << WGM11);
  TCCR1B = (1 << WGM13) | (1 << WGM12) | (1 << CS11);

  // set ICR1 to 39999 for 20ms period
  ICR1 = 39999;
}

////////////////////////////////////////////////////////////////////////
// FUNCTION: servo_write
////////////////////////////////////////////////////////////////////////
// Description: Sets the servo position by updating
OCR1A with the
//              pulse width corresponding to the
position in degrees
//              (0° to 180°).
////////////////////////////////////////////////////////////////////////
static void servo_write(uint8_t pos) {
  // calculate pulse width (ticks) for the given
  position
  uint16_t pulse = SERVO_MIN + ((uint16_t)pos *
  SERVO_STEP);
  if (pulse > SERVO_MAX)
    pulse = SERVO_MAX;
  OCR1A = pulse;
}
```

10.4 Understanding the Code

10.4.1 Includes

- **#include <avr/io.h>**: Provides register definitions for the ATmega328P.

- **#include <util/delay.h>**: Enables use of the _delay_ms() function for software delays.

10.4.2 Main Function

- **Timer1 Initialization**: The function timer1_init() configures Timer1 in Fast PWM mode with TOP set to ICR1 (39999), which creates a PWM period of 20ms (~50Hz). This PWM signal controls the servo on PB1 (Pin 9).
- **Servo Control**: The servo_write() function computes the PWM pulse width from a given servo angle. It uses a linear relationship with defined constants for minimum pulse width, maximum pulse width, and step size.

10.4.3 PWM and Servo Control

- **LED Initialization**: The leds_init() function sets PD5 and PD6 as outputs and turns them off initially.
- **Conditional LED Setting**: Using preprocessor directives, the code checks if both SWEEP_DELAY_UP and SWEEP_DELAY_DOWN are 5. If true, it turns off the LED on D5 and turns on the LED on D6; otherwise, it does the opposite. This mechanism simulates a status indicator similar to reactor control systems.

10.4.4 LED Control

- The main loop continuously sweeps the servo from 0° to 180° (up) and from 180° back to 0° (down), using _delay_ms() with the respective delay values.
- This loop ensures smooth, continuous servo motion while

the LED indicator remains constant based on the preset delays.

10.5 Program Flow Summary

- **Initialization**: The system sets up Timer1 for PWM and configures the LED pins.
- **LED Configuration**: Depending on the defined sweep delays, a compile-time condition sets the LED state.
- **Servo Sweep**: The main loop performs a bidirectional sweep of the servo motor, with defined delays for upward and downward movement.
- **Peripheral Interactions**: Direct register manipulations ensure efficient control of both the servo and the LEDs.

10.6 Reverse Engineering the Main Function

Objective: To analyze how the system initializes hardware, generates PWM signals, and controls LEDs.

- **Initialization Phase**: The microcontroller's peripherals are configured in the timer1_init() and leds_init() functions. The Timer1 setup involves setting up Fast PWM mode and establishing a 20ms period.
- **LED Logic**: Compile-time checks determine the LED state, providing visual feedback based on the system's operating parameters.
- **Control Flow**: The infinite loop in the main function implements the servo sweep in two phases (up and down) with precise timing delays. This reflects a robust embedded system design where hardware and software are tightly

integrated.

Here we start at our Reset handler.

```
                    //
                    // code
                    // Generated by Intel Hex
                    // code:0000-code:0076.1
                    //

                    ************************************************
                    *                 THUNK FUNCTION               *
                    ************************************************
                    thunk undefined Reset()
                      Thunked-Function: FUN_code_0034
        undefined   <UNASSIGNE…<RETURN>
                    INT0 (code:0000+2)              XREF[2,1… Entry Point(*),
                    Reset                                     Reset:003e(T),
                                                              Reset:003e(j),
                                                              Entry Point(*)
   code:0000 0c 94       jmp      FUN_code_0034
             34 00
```

Here is the decompiled code.

```
Decompile: Reset - (program.hex)

void Reset(void)

{
  R1 = 0;
  SREG = 0;
  Ylo = 0xff;
  Yhi = 8;
  DAT_mem_08fe = 0x3c;
  FUN_code_0040();
  FUN_code_0075();
  return;
}

```

We know FUN_code_0040() is our main.

```
                    **************************************************
                    *                    FUNCTION                    *
                    **************************************************
                    undefined FUN_code_0040()
       undefined      ⚠<UNASSIGNE…<RETURN>
                    FUN_code_0040                       XREF[1]:   FUN_code_0034:003a(c)
code:0040 21 9a        sbi       ADCW,0x1                                 = ??
code:0041 8a b1        in        R24,UCSR0B                               = ??
code:0042 80 66        ori       R24,0x60
code:0043 8a b9        out       UCSR0B,R24                               = ??
code:0044 82 e8        ldi       R24,0x82
code:0045 80 93        sts       ICR3L,R24                                = ??
          80 00
code:0047 8a e1        ldi       R24,0x1a
code:0048 80 93        sts       ICR3H,R24                                = ??
          81 00
code:004a 8f e3        ldi       R24,0x3f
code:004b 9c e9        ldi       R25,0x9c
code:004c 90 93        sts       OCR3AH,R25                               = ??
          87 00
code:004e 80 93        sts       OCR3AL,R24                               = ??
          86 00
code:0050 5d 98        cbi       UCSR0A,0x5                               = ??
code:0051 5e 9a        sbi       UCSR0A,0x6                               = ??

                    LAB_code_0052                       XREF[1]:   code:0074(j)
code:0052 80 ed        ldi       R24,0xd0
code:0053 97 e0        ldi       R25,0x7

                    LAB_code_0054                       XREF[1]:   code:0062(j)
code:0054 90 93        sts       TCNT3H,R25                               = ??
          89 00
code:0056 80 93        sts       TCNT3L,R24                               = ??
          88 00
code:0058 ef e1        ldi       Zlo,0x1f
code:0059 fe e4        ldi       Zhi,0x4e

                    LAB_code_005a                       XREF[1]:   code:005b(j)
code:005a 31 97        sbiw      Z,0x1
code:005b f1 f7        brbc      LAB_code_005a,Zflg
code:005c 00 c0        rjmp      LAB_code_005d
```

```
                    LAB_code_005d                       XREF[1]:   code:005c(j)
code:005d 00 00        nop
code:005e 0b 96        adiw      R25R24,0xb
code:005f 87 39        cpi       R24,0x97
code:0060 ff e0        ldi       Zhi,0xf
code:0061 9f 07        cpc       R25,Zhi
code:0062 89 f7        brbc      LAB_code_0054,Zflg
code:0063 8c e8        ldi       R24,0x8c
code:0064 9f e0        ldi       R25,0xf

                    LAB_code_0065                       XREF[1]:   code:0073(j)
code:0065 90 93        sts       TCNT3H,R25                               = ??
          89 00
code:0067 80 93        sts       TCNT3L,R24                               = ??
          88 00
code:0069 ef e1        ldi       Zlo,0x1f
code:006a fe e4        ldi       Zhi,0x4e

                    LAB_code_006b                       XREF[1]:   code:006c(j)
code:006b 31 97        sbiw      Z,0x1
code:006c f1 f7        brbc      LAB_code_006b,Zflg
code:006d 00 c0        rjmp      LAB_code_006e

                    LAB_code_006e                       XREF[1]:   code:006d(j)
code:006e 00 00        nop
code:006f 0b 97        sbiw      R25R24,0xb
code:0070 80 3d        cpi       R24,0xd0
code:0071 f7 e0        ldi       Zhi,0x7
code:0072 9f 07        cpc       R25,Zhi
code:0073 89 f7        brbc      LAB_code_0065,Zflg
code:0074 dd cf        rjmp      LAB_code_0052
```

Here is the decompiled view.

```
Decompile: FUN_code_0040 - (program.hex)

void FUN_code_0040(void)

{
  byte bVar1;

  bVar1 = ADCW;
  ADCW = bVar1 | 2;
  R25R24._0_1_ = UCSR0B;
  R25R24._0_1_ = (byte)R25R24 | 0x60;
  UCSR0B = (byte)R25R24;
  ICR3L = 0x82;
  ICR3H = 0x1a;
  OCR3AH = 0x9c;
  OCR3AL = 0x3f;
  bVar1 = UCSR0A;
  UCSR0A = bVar1 & 0xdf;
  bVar1 = UCSR0A;
  UCSR0A = bVar1 | 0x40;
  do {
    R25R24 = 2000;
    do {
      TCNT3H = R25R24._1_1_;
      TCNT3L = (byte)R25R24;
      Z = 19999;
      do {
        Z = Z + -1;
      } while (Z != 0);
      R25R24 = R25R24 + 0xb;
    } while ((byte)R25R24 != 0x97 || R25R24._1_1_ != (char)(((byte)R25R24 < 0x97) + '\x0f'));
    R25R24 = 0xf8c;
    do {
      TCNT3H = R25R24._1_1_;
      TCNT3L = (byte)R25R24;
      Z = 19999;
      do {
        Z = Z + -1;
      } while (Z != 0);
      R25R24 = R25R24 + -0xb;
    } while ((byte)R25R24 != 0xd0 || R25R24._1_1_ != (char)(((byte)R25R24 < 0xd0) + '\a'));
  } while( true );
}
```

LET'S HACK! We see 19999 in both loops we know that this is the SWEEP_DELAY_UP and SWEEP_DELAY_DOWN so if we shorten these values we can cause the reactor to twist back and forth so fast it will burn out the core and keep the green light on so the Engineers have NO IDEA!

```
                     LAB_code_0054                         XREF[1]:   code:0062(j)
    code:0054 90 93       sts        TCNT3H,R25                              = ??
              89 00
    code:0056 80 93       sts        TCNT3L,R24                              = ??
              88 00
    code:0058 ef e1       ldi        Zlo,0x1f
    code:0059 fe e4       ldi        Zhi,0x4e
```

The above is the first value where we must lower 0x43 to something say like 0x5.

```
                     LAB_code_0065                          XREF[1]:   code:0073(j)
code:0065 90 93        sts        TCNT3H,R25                              = ??
          89 00
code:0067 80 93        sts        TCNT3L,R24                              = ??
          88 00
code:0069 ef e1        ldi        Zlo,0x1f
code:006a fe e4        ldi        Zhi,0x4e
```

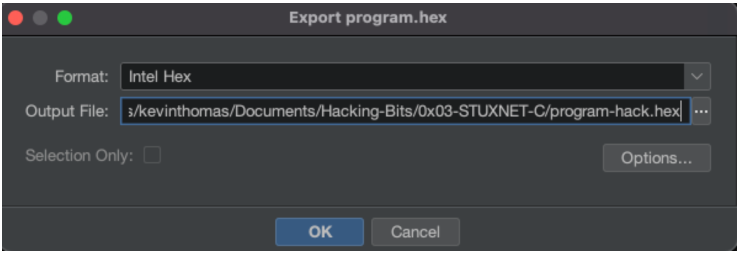
We will do the same above.

Right-click on each value in Ghidra and select patch instruction and change it to 0x5 for both.

```
                     LAB_code_0054                          XREF[1]:   code:0062(j)
code:0054 90 93        sts        TCNT3H,R25                              = ??
          89 00
code:0056 80 93        sts        TCNT3L,R24                              = ??
          88 00
code:0058 ef e1        ldi        Zlo,0x1f
code:0059 f5 e0        ldi        Zhi,0x5
```

```
                     LAB_code_0065                          XREF[1]:   code:0073(j)
code:0065 90 93        sts        TCNT3H,R25                              = ??
          89 00
code:0067 80 93        sts        TCNT3L,R24                              = ??
          88 00
code:0069 ef e1        ldi        Zlo,0x1f
code:006a f5 e0        ldi        Zhi,0x5
```

Now click File - Export Program ...



Rename to program-hack.hex and click OK.

Then in the folder run make and SEE THE HACK!

This concludes our book and course on Embedded Engineering! I hope you take this as a base knowledge and continue on your Reverse Engineering path!