Microprocessors and Digital Systems

Semester 2, 2022

Dr Mark Broadmeadow

Tarang Janawalkar

Contents

Part I

Microcontrollers

Chapter 1

Microcontroller Fundamentals

1.1 Architecture of a Computer

Definition 1.1.1 (Computer)**.** A computer is a digital electronic machine that can be programmed to carry out sequences of arithmetic or logical operations (computation) automatically.

Definition 1.1.2 (Control unit)**.** The control unit interprets the instructions and decides what actions to take.

Definition 1.1.3 (Arithmetic logic unit)**.** The arithmetic logic unit (ALU) performs computations required by the control unit.

1.2 Microprocessors & Microcontrollers

While a microcontroller puts the CPU and all peripherals onto the same chip, a microprocessor houses a more powerful CPU on a single chip that connects to external peripherals. The peripherals include memory, I/O, and control units.
The QUTy uses a microcontroller called ATtiny1626, that are within a family of microcontrollers called AVRs.

1.3 ATtiny1626 Microcontroller

The ATtiny1626 microcontroller has the following features:

- CPU: AVR Core (AVRxt variant)
- Memory:
  - Flash memory (16KB) used to store program instructions in memory
  - SRAM (2KB) used to store data in memory
  - EEPROM (256B)
- Peripherals: Implemented in hardware (part of the chip) in order to offload complexity

1.3.1 Flash Memory

- Non-volatile — memory is not lost when power is removed
- Inexpensive
- Slower than SRAM
- Can only erase in large chunks
- Typically used to store programme data
- Generally read-only. Programmed via an external tool, which is loaded once and remains static during the lifetime of the program
- Writing is slow

1.3.2 SRAM

- Volatile — memory is lost when power is removed
- Expensive
- Faster than flash memory and is used to store variables and temporary data
- Can access individual bytes (large chunk erases are not required)

1.3.3 EEPROM

- Older technology
- Expensive
- Non-volatile
- Can erase individual bytes

1.4 AVR Core

Definition 1.4.1 (Computer programme)**.** A computer programme is a sequence or set of instructions in a programming language for a computer to execute.

The main function of the AVR Core Central Processing Unit (CPU) is to ensure correct program execution. The CPU must, therefore, be able to access memory, perform calculations, control peripherals, and handle interrupts. Some key characteristics of the AVR Core are:

- 8-bit Reduced Instruction Set Computer (RISC)
- 32 working registers (R0 to R31)
- Program Counter (PC) — location in memory where the program is stored

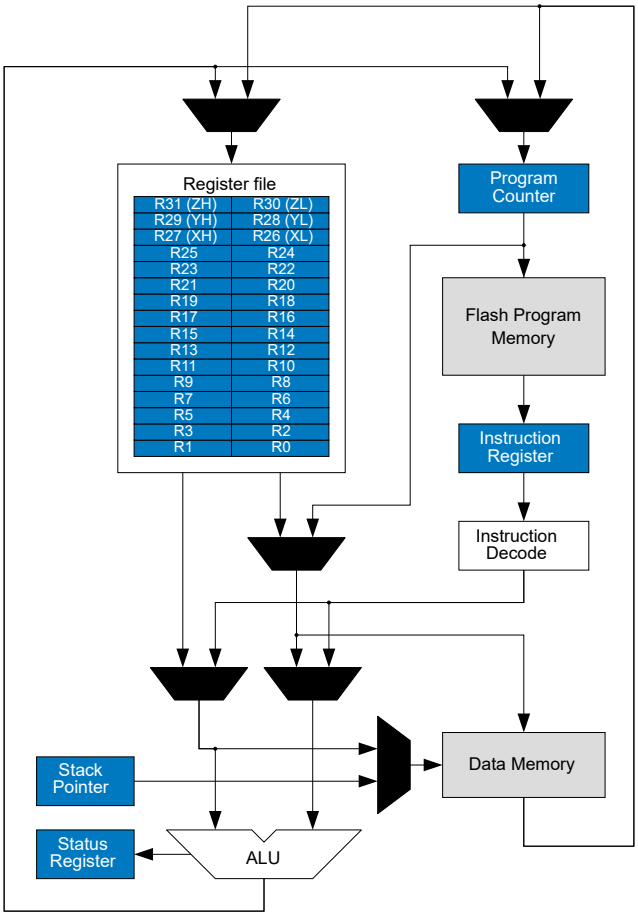
- Status Register (SREG) — stores key information from calculations performed by the ALU (i.e., whether a result is negative)
- Stack Pointer — temporary data that doesn't fit into the registers
- 8-bit core — all data, registers, and operations, operate within 8-bits

1.5 Programme Execution

At the time of reset, PC = 0. The following steps are then performed:

1. Fetch instruction (from memory)
2. Decode instruction (decode binary instruction)
3. Execute instruction:
    - Execute an operation
    - Store data in data memory, the ALU, a register, or update the stack pointer
4. Store result
5. Update PC (move to next instruction or if instruction is longer than 1 word, increment twice. The program can also move to another point in the program that has an address k, through jumps.)

This is illustrated in the following figure:



1.6 Instructions

- The CPU understands and can execute a limited set of instructions — ~88 unique instructions for the ATtiny1626
- Instructions are encoded in programme memory as opcodes. Most instructions are two bytes long, but some instructions are four bytes long
- The AVR Instruction Set Manual describes all of the available instructions, and how they are translated into opcodes
- Instructions fall loosely into five categories:
  - Arithmetic and logic — ALU
  - Change of flow — jumping to different sections of the code or making decisions
  - Data transfer — moving data in/out of registers, into the data space, or into RAM
  - Bit and bit-test — looking at data in registers (which bits are set or not set)

– Control — changing what the CPU is doing

1.7 Interacting with memory and peripherals

- The CPU interacts with both memory and peripherals via the data space
- From the perspective of the CPU, the data space is single large array of locations that can be read from, or written to, using an address
- We control peripherals by reading from, and writing to, their registers
- Each peripheral register is assigned a unique address in the data space
- When a peripheral is accessed in this manner we refer to it as being memory mapped, as we access them as if they were normal memory
- Different devices, peripherals and memory can be included in a memory map (and sometimes a device can be accessed at multiple different addresses)

1.8 Memory map

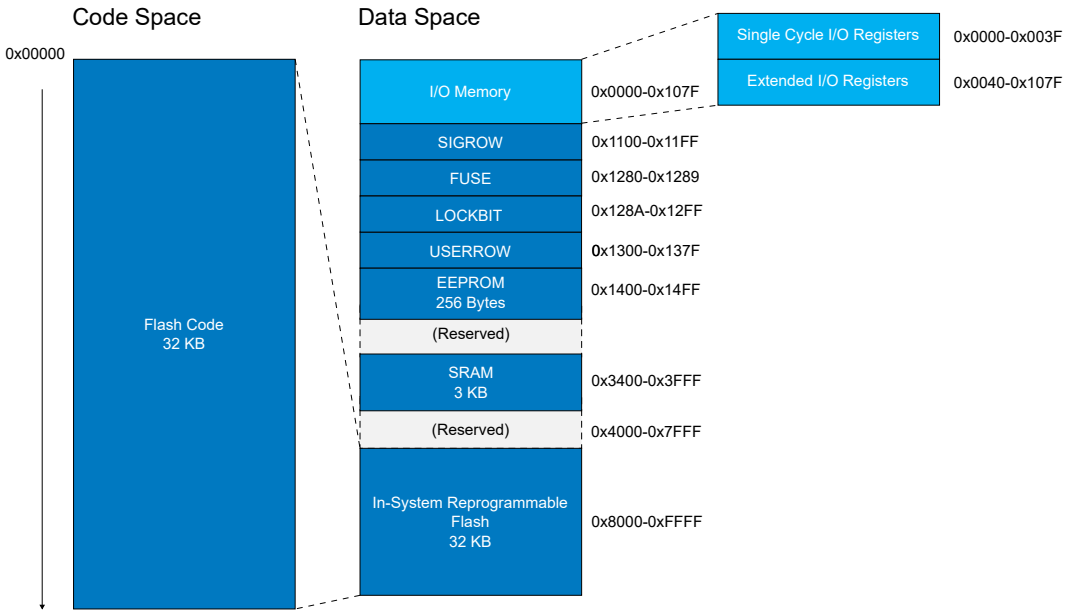


Figure 1.1: ATtiny1626 memory map.

1.9 Assembly code

- The opcodes placed into programme memory are called machine code (i.e., code the machine operates on directly)
- We don't write machine code directly as it is:
  - Not human readable
  - Prone to errors (swapping a single bit can completely change the operation)
- Instead we can write instructions directly in assembly code
- We use instruction mnemonics to specify each instruction: `ldi`, `add`, `sts`, `jmp`, …
- An assembler takes assembly code and translates it into opcodes that can be loaded into programme memory

Chapter 2

Digital Representations and Operations

2.1 Digital Systems

A **bit**[1] is the most basic unit of information in a digital system. A bit encodes a logical state with one of two possible values (i.e., binary). These states are often referred to as:

- true, false
- high, low (voltage states)
- on, off (logical states)
- set, reset
- 1, 0

A sequence of *eight* bits is known as a **byte**, and it is the most common representation of data in digital systems. A sequence of *four* bits is known as a **nibble**.
A sequence of n bits can represent up to 2^n states.

2.2 Representation

2.2.1 Binary Representation

The **binary system** is a base-2 system that uses a sequence of bits to represent a number. Bits are written left-to-right from **most significant** to **least significant** bit.
The first bit is the "most significant" bit because it is associated with the highest value in the sequence (coefficient of the highest power of two).

- The **least significant bit** (LSB) is at bit index 0.

[1] The term *bit* comes from **b**inary dig**it**.

- The **most significant bit** (MSB) is at bit index $n-1$ in an n-bit sequence.

$0000_2 = 0$	$0100_2 = 4$	$1000_2 = 8$	$1100_2 = 12$
$0001_2 = 1$	$0101_2 = 5$	$1001_2 = 9$	$1101_2 = 13$
$0010_2 = 2$	$0110_2 = 6$	$1010_2 = 10$	$1110_2 = 14$
$0011_2 = 3$	$0111_2 = 7$	$1011_2 = 11$	$1111_2 = 15$

The subscript 2 indicates that the number is represented using a base-2 system.

2.2.2 Hexadecimal Representation

The **hexadecimal system** (hex) is a base-16 system. As we need 16 digits in this system, we use the letters A to F to represent digits 10 to 15.
Hex is a convenient notation when working with digital systems as each hex digit maps to a nibble.

$0_{16} = 0000_2$	$4_{16} = 0100_2$	$8_{16} = 1000_2$	$C_{16} = 1100_2$
$1_{16} = 0001_2$	$5_{16} = 0101_2$	$9_{16} = 1001_2$	$D_{16} = 1101_2$
$2_{16} = 0010_2$	$6_{16} = 0110_2$	$A_{16} = 1010_2$	$E_{16} = 1110_2$
$3_{16} = 0011_2$	$7_{16} = 0111_2$	$B_{16} = 1011_2$	$F_{16} = 1111_2$

2.2.3 Numeric Literals

When a fixed value is declared directly in a program, it is referred to as a **literal**. Generally, numeric literals can be expressed as either binary, decimal, or hexadecimal, so we use prefixes to denote various bases. Typically,

- **Binary** notation requires the prefix `0b`
- **Decimal** notation does not require prefixes
- **Hexadecimal** notation requires the prefix `0x`

For example, `0x80 = 0b10000000 = 128`.

2.3 Unsigned Integers

The **unsigned integers** represent the set of counting (natural) numbers, starting at 0. In the **decimal system** (base-10), the unsigned integers are encoded using a sequence of decimal digits (0–9).
The decimal system is a **positional numeral system**, where the contribution of each digit is determined by its position. For example,

$$\begin{aligned} 278_{10} &= 2 \times 10^2 &&+ 7 \times 10^1 &&+ 8 \times 10^0 \\ &= 2 \times 100 &&+ 7 \times 10 &&+ 8 \times 1 \\ &= 200 &&+ 70 &&+ 8 \end{aligned}$$

In the **binary system** (base-2) the unsigned integers are encoded using a sequence of binary digits (0–1) in the same manner. For example,

$$\begin{aligned} 10101_2 &= 1 \times 2^4 & + 0 \times 2^3 & & + 1 \times 2^2 & & + 0 \times 2^1 & & + 1 \times 2^0 \\ &= 1 \times 16 & + 0 \times 8 & & + 1 \times 4 & & + 0 \times 2 & & + 1 \times 1 \\ &= 16 & + 0 & & + 4 & & + 0 & & + 1 \\ &= 21_{10} \end{aligned}$$

The range of values an n-bit binary number can hold when encoding an unsigned integer is 0 to $2^n - 1$.

No. of Bits	Range
8	0–255
16	0–65 535
32	0–4 294 967 295
64	0–18 446 744 073 709 551 615

Table 2.1: Range of available values in binary representations.

2.4 Signed Integers

Signed integers are used to represent integers that can be positive or negative. The following representations allow us to encode negative integers using a sequence of binary bits:

- Sign-magnitude
- One's complement
- Two's complement (most common)

2.4.1 Sign-Magnitude

In sign-magnitude representation, the most significant bit encodes the sign of the integer. In an 8-bit sequence, the remaining 7-bits are used to encode the value of the bit.

- If the sign bit is 0, the remaining bits represent a positive value,
- If the sign bit is 1, the remaining bits represent a negative value.

As the sign bit consumes one bit from the sequence, the range of values that can be represented by an n-bit sign-magnitude encoded bit sequence is:

$$-(2^{n-1} - 1) \text{ to } 2^{n-1} - 1$$

For 8-bit sequences, this range is: -127 to 127.
However, there are some issues with this representation.

1. There are two ways to represent zero: `0b10000000 = 0`, or `0b00000000 = -0`.
2. Arithmetic and comparison requires inspecting the sign bit
3. The range is reduced by 1 (due to the redundant zero representation)

2.4.2 One's Complement

In one's complement representation, a negative number is represented by inverting the bits of a positive number (i.e., $0 \to 1$ and $1 \to 0$).
The range of values are still the same:

$$-\left(2^{n-1}-1\right) \text{ to } 2^{n-1}-1$$

however, this representation tackles the second problem in the previous representation as addition is performed via standard binary addition with *end-around carry* (carry bit is added onto result).

$$a - b = a + (\sim b) + C.$$

2.4.3 Two's Complement

In two's complement representation, the most significant bit encodes a negative weighting of -2^{n-1}.
For example, in 8-bit sequences, index-7 represents a value of -128.
The two's complement is calculated by adding 1 to the one's complement.
The range of values are:

$$-2^{n-1} \text{ to } 2^{n-1}-1$$

This representation is more efficient than the previous because `0` has a single representation and subtraction is performed by adding the two's complement of the subtrahend.

$$a - b = a + (\sim b + 1)\,.$$

2.5 Logical Operators

2.5.1 Boolean Functions

A Boolean function is a function whose arguments and results assume values from a two-element set, (usually {0, 1} or `{false, true}`).
These functions are also referred to as *logical functions* when they operate on bits. The most common logical functions available to microprocessors and most programming languages are:

- Negation: `NOT`
- Conjunction: `AND`
- Disjunction: `OR`
- Exclusive disjunction: `XOR`

By convention, we map a bit value of `0` to `false`, and a bit value of `1` to `true`.

2.5.2 Negation

`NOT` is a unary operator that is used to **invert** a bit. It is typically expressed as:

- `NOT` a
- $\sim a$
- $\overline{a}$

Truth table:

a	`NOT` a
0	1
1	0

2.5.3 Conjunction

`AND` is a binary operator whose output is true if **both** inputs are **true**. It is typically expressed as:

- a `AND` b
- a `&` b
- $a \cdot b$
- $a \wedge b$

Truth table:

a	b	a `AND` b
0	0	0
0	1	0
1	0	0
1	1	1

2.5.4 Disjunction

`OR` is a binary operator whose output is true if **either** input is **true**. It is typically expressed as:

- a `OR` b
- a `|` b
- $a + b$
- $a \vee b$

Truth table:

a	b	a `OR` b
0	0	0
0	1	0
1	0	1
1	1	1

2.5.5 Exclusive Disjunction

`XOR` (Exclusive `OR`) is a binary operator whose output is true if **only one** input is **true**. It is typically expressed as:

- a `XOR` b
- $a \hat{} b$
- $a \oplus b$

Truth table:

a	b	a `XOR` b
0	0	0
0	1	1
1	0	1
1	1	0

2.5.6 Bitwise Operations

When applying logical operators to a sequence of bits, the operation is performed in a **bitwise** manner. The result of each operation is stored in the corresponding bit index also.

2.6 Bit Manipulation

Often we need to modify individual bits within a byte, **without** modifying other bits. This is accomplished by performing a bitwise operation on the byte using a **bit mask** or **bit field**.
These operations can:

- **Set** specific bits (change value to `1`)
- **Clear** specific bits (change value to `0`)
- **Toggle** specific bits (change values from $0 \rightarrow 1$, or $1 \rightarrow 0$)

2.6.1 Setting Bits

To **set** a bit, we take the bitwise `OR` of the byte, with a bit mask that has a **1** in each position where the bit should be set.

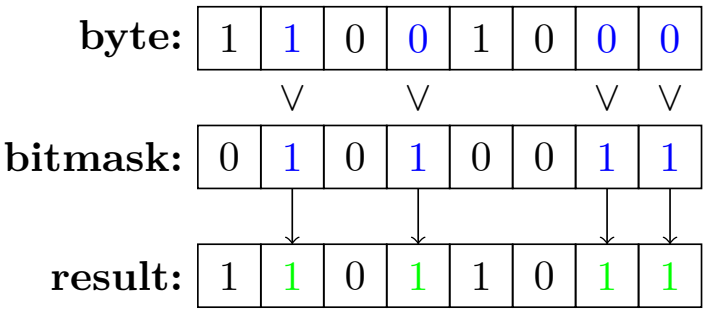


Figure 2.1: Setting bits using the logical or.

2.6.2 Clearing Bits

To **clear** a bit, we take the bitwise `AND` of the byte, with a bit mask that has a **0** in each position where the bit should be cleared.

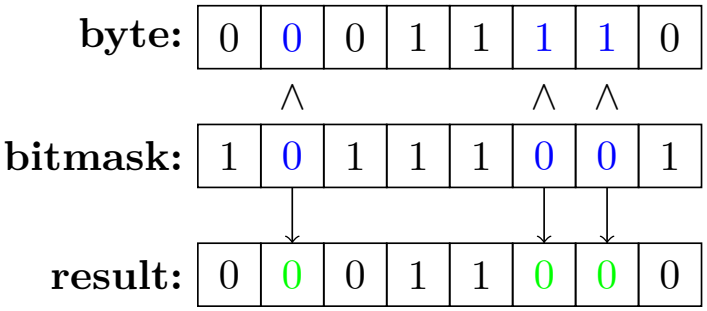


Figure 2.2: Clearing bits using the logical and.

2.6.3 Toggling Bits

To **toggle** a bit, we take the bitwise `XOR` of the byte, with a bit mask that has a **1** in each position where the bit should be toggled.

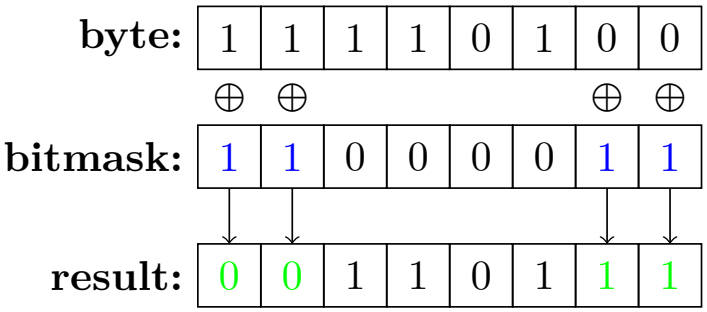


Figure 2.3: Toggling bits using the logical exclusive or.

Other bitwise operations act on the entire byte.

- One's complement (bitwise `NOT`)
- Two's complement (bitwise `NOT` + 1)
- Shifts
  - Logical
  - Arithmetic (for signed integers)
- Rotations

2.6.4 One's Complement

The one's complement of a byte inverts every bit in the operand. This is done by taking the bitwise `NOT` of the byte.
Similarly, we can subtract the byte from `0xFF` to get the one's complement.

2.6.5 Two's Complement

The two's complement of a byte is the one's complement of the byte plus one. Therefore, we can apply take the bitwise `NOT` of the byte, and then add one to it.

2.6.6 Shifts

Shifts are used to move bits within a byte. In many programming languages this is represented by two greater than `>>` or two less than `<<` characters.

$$a \gg s$$

shifts the bits in a by s places to the right while adding `0`'s to the MSB.

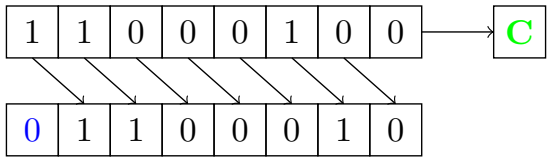


Figure 2.4: Right shift using `lsr` in AVR Assembly.

Similarly

$$a \ll s$$

shifts the bits in a by s places to the left while adding `0`'s to the LSB.

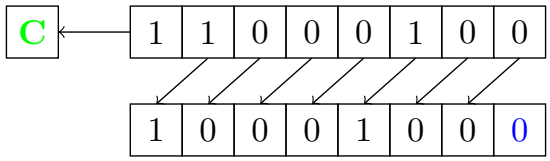


Figure 2.5: Left shift using `lsl` in AVR Assembly.

When using signed integers, the arithmetic shift is used to preserve the value of the sign bit when shifting.

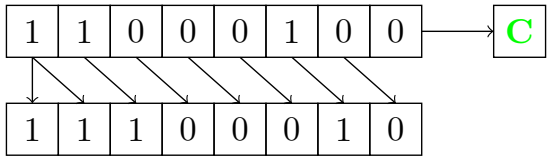


Figure 2.6: Arithmetic right shift using `asr` in AVR Assembly.

Left shifts are used to multiply numbers by 2, whereas right shifts are used to divide numbers by 2 (with truncation).

2.6.7 Rotations

Rotatations are used to shift bits with a carry from the previous instruction.

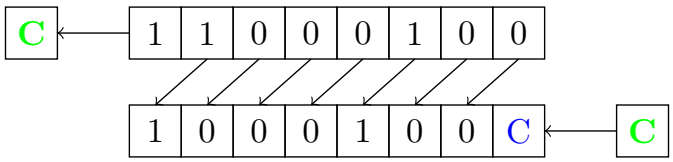


Figure 2.7: Rotate left using `rol` in AVR Assembly.

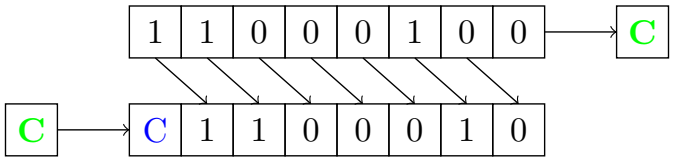


Figure 2.8: Rotate right using `ror` in AVR Assembly.

Here the blue bit is carried from the previous instruction, and the carry bit is updated to the value of the bit that was shifted out. Rotations are used to perform multi-byte shifts and arithmetic operations.

2.7 Arithmetic Operations

2.7.1 Addition

Addition is performed using the same process as decimal addition except we only use two digits, 0 and 1.

1. `0b0 + 0b0 = 0b0`
2. `0b0 + 0b1 = 0b1`
3. `0b1 + 0b1 = 0b10`

When adding two 1's, we carry the result into the next bit position as we would with a 10 in decimal addition. In AVR Assembly, we can use the `add` instruction to add two bytes. The following example adds two bytes.

```
1  ; Accumulator
2  ldi r16, 0
3
4  ; First number
5  ldi r17, 29
6  add r16, 17 ; R16 <- R16 + R17 = 0 + 29 = 29
7
8  ; Second number
9  ldi r17, 118
10 add r16, 17 ; R16 <- R16 + R17 = 29 + 118 = 147
```

Below is a graphical illustration of the above code.

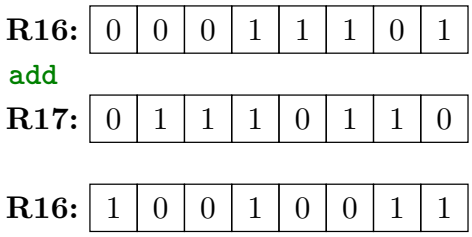


Figure 2.9: Overflow addition using `add` in AVR Assembly.

2.7.2 Overflows

When the sum of two 8-bit numbers is greater than 8-bit (255), an **overflow** occurs. Here we must utilise a second register to store the high byte so that the result is represented as a 16-bit number. To avoid loss of information, a **carry bit** is used to indicate when an overflow has occurred. This carry bit can be added to the high byte in the event that an overflow occurs. This is because the carry bit is 0 when the sum is less than 256, and 1 when the sum is greater than 255.
The following example shows how to use the `adc` instruction to carry the carry bit when an overflow occurs.

```
; Low byte
ldi r30, 0
; High byte
ldi r31, 0

; Empty byte for adding carry bit
ldi r29, 0

; First number
ldi r16, 0b11111111
; Add to low byte
add r30, r16 ; R30 <- R30 + R16 = 0 + 255 = 255, C <- 0
; Add to high byte
adc r31, r29 ; R31 <- R31 + R29 + C = 0 + 0 + 0 = 0

; Second number
ldi r16, 0b00000001
; Add to low byte
add r30, r16 ; R30 <- R30 + R16 = 255 + 1 = 0, C <- 1
; Add to high byte
adc r31, r29 ; R31 <- R31 + R29 + C = 0 + 0 + 1 = 1
```

Therefore the final result is: `R31:R30 = 0b00000001:0b00000001 = 256`. Below is a graphical representation of the above code.

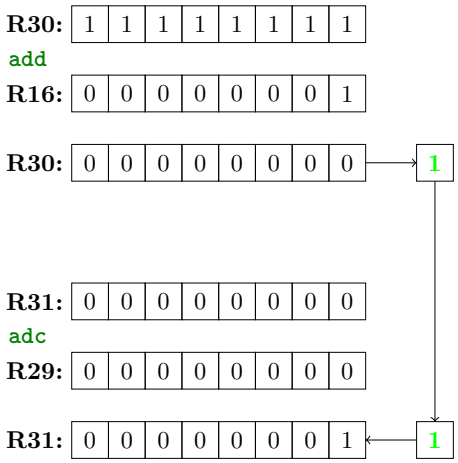


Figure 2.10: Overflow addition using `adc` in AVR Assembly.

2.7.3 Subtraction

Subtraction is performed using the same process as binary addition, with the subtrahend in two's complement form. In the case of overflows, the carry bit is discarded.

2.7.4 Multiplication

Multiplication is understood as the sum of a set of partial products, similar to the process used in decimal multiplication. Here each digit of the multiplier is multiplied to the multipicand and each partial product is added to the result.
Given an m-bit and an n-bit number, the product is at most $(m+n)$-bits wide.

$$\begin{aligned}
13 \times 43 &= 00001101_2 \times 00101011_2 \\
&= \quad 00001101_2 \times 1_2 \\
&\quad + \; 00001101_2 \times 10_2 \\
&\quad + \; 00001101_2 \times 1000_2 \\
&\quad + \; 00001101_2 \times 100000_2 \\
&= \quad 00001101_2 \\
&\quad + \; 00011010_2 \\
&\quad + \; 01101000_2 \\
&\quad + \; 110100000_2 \\
&= 1000101111
\end{aligned}$$

Using AVR assembly, we can use the `mul` instruction to perform multiplication.

```
; First number
ldi r16, 13
```

```
; Second number
ldi r17, 43

; Multiply
mul r16, r17 ; R1:R0 <- 0b00000010:0b00101111 = 2:47
```

The result is stored in the register pair `R1:R0`.

2.7.5 Division

Division, square roots and many other functions are very expensive to implement in hardware, and thus are typically not found in conventional ALUs, but rather implemented in software.
However, there are other techniques that can be used to implement division in hardware. By representing the divisor in reciprocal form, we can try to represent the number as the sum of powers of 2.
For example, the divisor 6.4 can be represented as:

$$\frac{1}{6.4} = \frac{10}{64} = 10 \times 2^{-6}$$

so that dividing an integer n by 6.4 is approximately equivalent to:

$$\frac{n}{6.4} \approx (n \times 10) \gg 6$$

When the divisor is not exactly representable as a power of 2 we can use fractional exponents to represent the divisor, however this requires a floating point system implementation which is not provided on the AVR.

Chapter 3

Microcontroller Interfacing

3.1 Logic Levels

3.1.1 Discretisation

The process of discretisation translates a continuous signal into a discrete signal (bits). As an example, we can translate **voltage levels** on microcontroller pins into digital **logic levels**.

3.1.2 Logic Levels

For digital input/output (IO), conventionally:

- The voltage level of the positive power supply represents a **logical 1**, or the **high state**, and
- 0 V (ground) represents a **logical 0**, or the **low state**.

The QUTy is supplied 3.3 V so that when a digital output is high, the voltage present on the corresponding pin will be around 3.3 V. Because voltage is a continuous quantity, we must discretise the full range of voltages into logical levels using **thresholds**.

- A voltage **above** the input **high threshold** t_H is considered **high**.
- A voltage **below** the input **low threshold** t_L is considered **low**.

The interpretation of a voltage between these states is determined by **hysteresis**.

3.1.3 Hysteresis

Hysteresis refers to the property of a system whose state is **dependent** on its **history**. In electronic circuits, this avoids ambiguity in determining the state of an input as it switches between voltage levels.

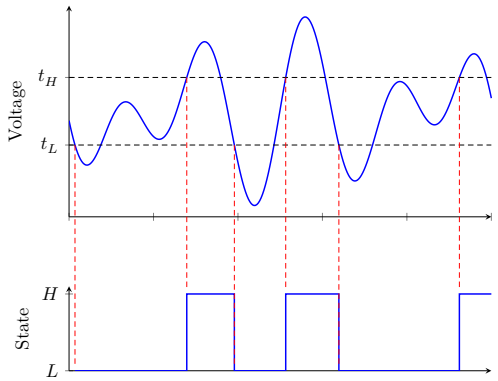


Figure 3.1: Example of hysteresis.

Given a transition:

- If an input is currently in the **low state**, it has not transitioned to the **high state** until the voltage crosses the **high input voltage** threshold.
- If an input is currently in the **high state**, it has not transitioned to the **low state** until the voltage crosses the **low input voltage** threshold.

It is therefore always preferrable to drive a digital input to an unambiguous voltage level.

3.2 Electrical Quantities

3.2.1 Voltage

Voltage v measures the electrical *potential difference* between two points in a circuit, measured in **Volts** (V).

- Voltage is measured across a circuit element, or between two points in a circuit, most commonly with respect to a 0 V reference (ground).
- It represents the **potential** of the electrical system to do **work**.

3.2.2 Current

Current i measures the *rate of flow of electrical charge* through a circuit, measured in **Amperes** (A).

- Current is measured through a circuit element.

3.2.3 Power

Power p is the rate of energy transferred per unit time, measured in **Watts** (W). Power can be determined through the equation

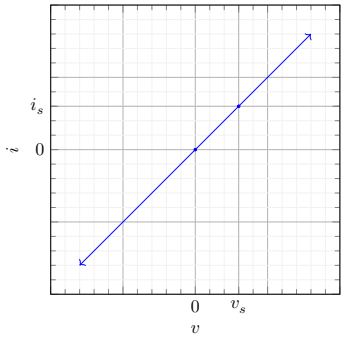
$$p = vi.$$

3.2.4 Resistance

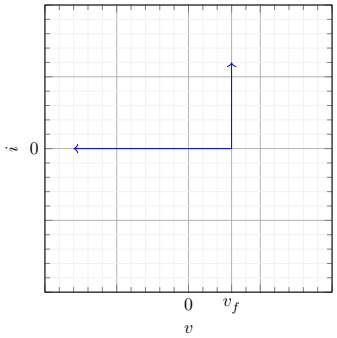
Resistance R is a property of a material to *resist the flow of current*, measured in **Ohms** (Ω). Ohm's law states that the voltage across a component is proportional to the current that flows through it:

$$v = iR.$$

Note that not all circuit elements are resistive (or Ohmic), such that they do not follow Ohm's law, this can be seen in diodes.



(a) VI curve for Ohmic components.



(b) VI curve for diodes.

Figure 3.2: Voltage-current characteristic curves for various components.

Although the wires used to connect a circuit are resistive, we usually assume that they are ideal, that is, they have zero resistance.

3.3 Electrical Components

3.3.1 Resistors

A **resistor** is a circuit element that is designed to have a specific resistance R.

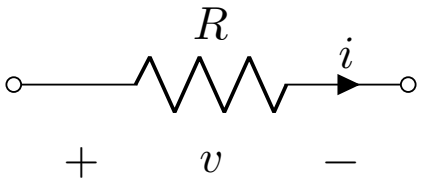


Figure 3.3: Resistor circuit symbol.

3.3.2 Switches

A **switch** is used to connect and disconnect different elements in a circuit. It can be **open** or **closed**.

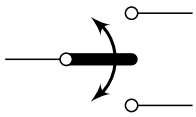
- In the **open** state, the switch **will not conduct**[1] current
- In the **closed** state, the switch **will conduct** current

Switches can take a variety of forms:

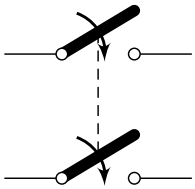
- **Poles** — the number of circuits the switch can control.
- **Throw** — the number of output connections each pole can connect its input to.
- Momentary or toggle action
- Different form factors, e.g., push button, slide, toggle, etc.

Switches are typically for user input.

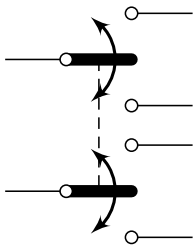
(a) Single pole single throw switch.



(b) Single pole double throw switch.



(c) Double pole single throw switch.



(d) Double pole double throw switch.

Figure 3.4: Various types of switches.

3.3.3 Diodes

A **diode** is a semiconductor device that conducts current in only one direction: from the **anode** to the **cathode**.

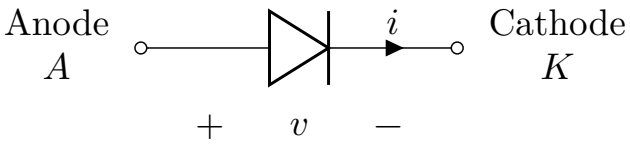


Figure 3.5: Diode symbol.

Diodes are a non-Ohmic device:

[1] Conductance is a measure of the ability for electric charge to flow in a certain path.

- When **forward biased**, a diode **does** conduct current, and the anode-cathode voltage is equal to the diodes **forward voltage**.
- When **reverse biased**, a diode **does not** conduct current, and the cathode-anode voltage is equal to the **applied voltage**.

(a) Forward biased diode. $v_{AK} = v_f$ and $i > 0$.

(b) Reverse biased diode. $v_{KA} > 0$ and $i = 0$.

Figure 3.6: Diodes in forward and reverse bias.

A diode is only forward biased when the applied anode-cathode voltage **exceeds** the forward voltage v_f. A typical forward voltage v_f for a silicon diode is in the range 0.6 V to 0.7 V, whereas for Light Emitting Diodes (LEDs), v_f ranges between 2 V to 3 V.

3.3.4 Integrated Circuit

An **integrated circuit** (IC) is a set of electronic circuits (typically) implemented on a single piece of semiconductor material, usually silicon. ICs comprise of hundreds to many thousands of transistors, resistors and capacitors; all implemented on silicon. ICs are **packaged**, and connections to the internal circuitry are exposed via **pins**.
In general, the specific implementation of the IC is not important, but rather the **function of the device** and how it **interfaces** with the rest of the circuit. Hence ICs can be treated as a functional **black box**.
For digital ICs:

- **Input pins** are typically **high-impedance**, and they appear as an open circuit.
- **Output pins** are typically **low-impedance**, and will actively drive the voltage on a pin and any connected circuitry to a **high** or **low** state. They can also drive connected loads.

3.4 Digital Outputs

Digital output interfaces are designed to be able to drive connected circuitry to one of states, high, or low, however, the appropriate technique is **context specific**. When referring to digital outputs, we will refer to the states of a net. A **net** is defined as the common point of connection of multiple circuit components.
In this section we will consider:

- What kind of load the output drives?
- Could more than one device be attempting to actively drive the net to a specific logic level?

3.4.1 Push-Pull Outputs

A push-pull digital output is the most common form of output used in digital outputs. The **output driver** A *drives* the **output state** Y to:

- **HIGH** by connecting the output net to the supply voltage +V.
- **LOW** by connecting the output net to the ground voltage GND (0 V).

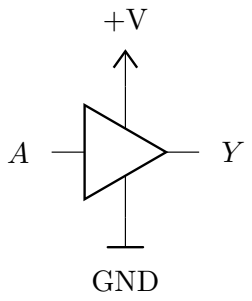


Figure 3.7: Push-pull output.

Hence the output state Y is determined by the logic level of the output driver A.

$$Y = A.$$

A	Y
LOW	LOW
HIGH	HIGH

Table 3.1: Truth table for a push-pull digital output.

The push-pull output Y can both source and sink current from the connected net.

3.4.2 High-Impedance Outputs

In many instances, a digital output is required to be placed in a high-impedance (HiZ) state. This is accomplished by using an **output enable** (OE) signal.

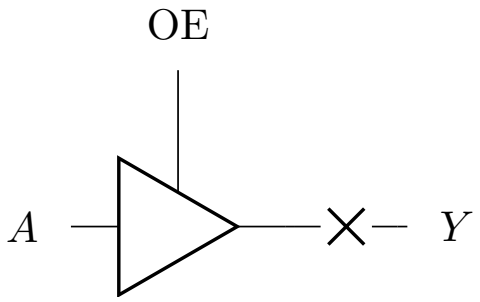


Figure 3.8: High-impedance output.

- When the OE signal is **HIGH**, the output state Y is determined by the output driver A.
- When the OE signal is **LOW**, the output state Y is in a **high-impedance** state.

A	**OE**	Y
LOW	LOW	HiZ
HIGH	LOW	HiZ
LOW	HIGH	LOW
HIGH	HIGH	HIGH

Table 3.2: Truth table for a push-pull digital output.

When the output is in **HiZ state**:

- The output is an effective **open circuit**, meaning it has **no effect** on the rest of the circuit.
- The voltage on the output net is determined by the **other circuitry** connected to the net.

HiZ outputs are typically used when multiple need to signal over the same wire(s).

3.4.3 Pull-up and Pull-down Resistors

When **no devices** are actively driving a net (e.g., all connected outputs are in the HiZ state), the state of the net is not well-defined. Hence we can use a **pull-up** or **pull-down** resistor to ensure that the state of the pin is always **well-defined**.

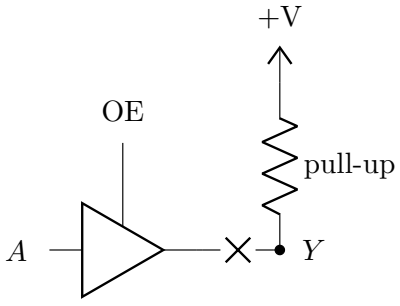


Figure 3.9: Pull-up resistor.

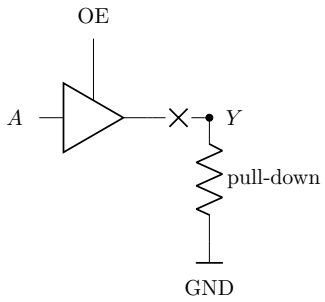


Figure 3.10: Pull-down resistor.

- When **no circuitry** is actively driving the net, the resistor will passively pull the voltage to either the voltage supply, or ground.
- When **another device** actively drives the net, the active device defines the voltage of the net. Hence the current from the resistor is simply sourced or sunk by the **active device**.

The resistors used as pull-up and pull-down resistors are typically in the kΩ range.

3.4.4 Open-Drain Outputs

Multiple push-pull outputs should never be connected to the same net as when one output is driven HIGH and another is driven LOW, an effective short circuit is created and one or more devices may be damaged. While push-pull outputs with an output enable may be used, the timing must be carefully managed.
Hence a more robust solution is to use open-drain outputs.

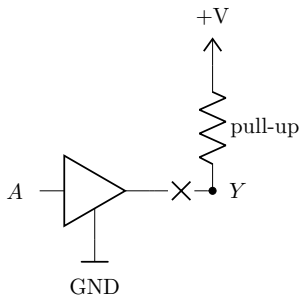


Figure 3.11: Open-drain output.

An open-drain output is either:

- In the **high-impedance** state, where the pull-up resistor is used to pull the net to the **high state** when the net is **not driven low**.
- **Connected to ground**, when the net is **driven low**.

3.5 Microcontroller Pins

Microcontrollers are interfaced via their exposed pins. These pins are the only means to access inputs and outputs, and they are used to interface with other electronic circuits in order to achieve a required functionality. Pins can be used for:

- General purpose input and output (GPIO) — pin represents a digital state
- Peripheral functions
- Other functions (power supply, reset input, clock input, etc.)

Pins are typically organised into groups of related IO banks, referred to as **ports** on the AVR microcontroller.
These ports and pins are assigned an alphanumeric identifier, (e.g., PB7 for pin 7 on port B).

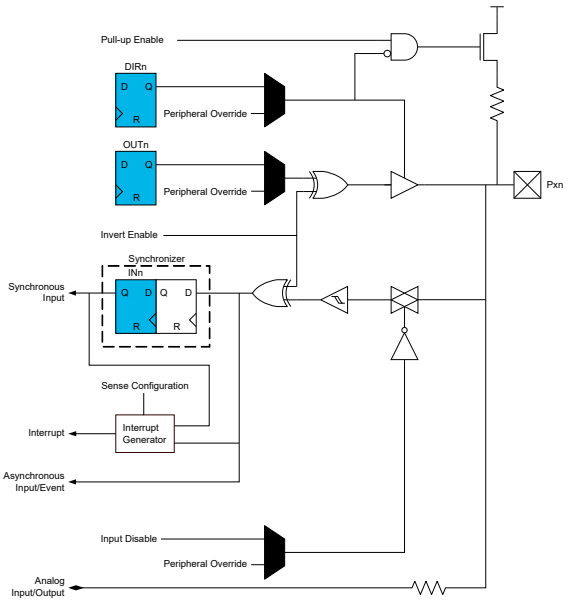


Figure 3.12: ATtiny1626 PORT block diagram.

To summarise this diagram:

- The data direction register (DIR) controls the push-pull output enable.
- The output driver register (OUT) drives the output state.
- The input register (IN) reads the output state.
- The internal pull-up register enabled through software.
- The physical voltage on the pin can be routed to an analogue to digital converter (ADC)
- Other peripheral functions can override port pin configurations and the output state.

3.5.1 Configuring an Output in Assembly

1. Place the port pin in a **safe initial state** by writing to the OUT register (HIGH or LOW depending on the context).
2. Configure the port pin as an output by **setting** the corresponding bits in the DIR register.
3. Set the desired pin state by writing to the OUT register.

```
; Load macros for easy access to port data space addresses.
\#include <avr/io.h>

; Bitmask for pin 5
ldi r16, PIN5_bm

; Set initial safe state
sts PORTB_OUTCLR, r16 ; LOW if active HIGH
sts PORTB_OUTSET, r16 ; HIGH if active LOW

; Enable output
sts PORTB_DIRSET, r16 ; Enable output on PB5

; Set output state to desired value
sts PORTB_OUTSET, r16 ; Set state of PB5 to HIGH
```

3.5.2 Configuring an Input in Assembly

1. If required, enable the internal pull-up resistor by **setting** the PULLUPEN bit in the corresponding PINnCTRL register.
2. Read the IN register to get the current state of the pin.
3. Isolate the relevant pin using the AND operator.

```
; Load macros for easy access to port data space addresses.
#include <avr/io.h>

; Bitmask for pin 5
ldi r16, PIN5_bm

; Enable internal pull-up resistor if required
sts PORTB_PIN5CTRL, r16

; Read output state from data space
lds r17, PORTA_IN
; Read output state using virtual PORT
in r17, VPORTA_IN
```

```
14
15 ; Isolate desired pin
16 andi r17, r16
```

3.5.3 Peripheral Multiplexing

Pins can be used to connect internal peripheral functions to external devices. As microcontrollers have more peripheral functions than available pins, peripheral functions are typically multiplexed onto pins.

Definition 3.5.1 (Multiplexing)**.** Multiplexing is a method by which **multiple peripheral functions** are mapped to the **same pin**. In this scenario, only one function can be enabled at a time, and the pin cannot be used for GPIO.

- Peripheral functions can be mapped to different **sets of pins** to provide flexibility and to avoid clashes when multiple peripherals are used in an application.
- When enabled, peripheral functions **override** standard port functions.
- The **Port Multiplexer** (PORTMUX) is used to select which **pin set** should be used by a peripheral.
- Certain peripherals can have their inputs/outputs mapped to different **sets of pins** through the PORTMUX.

Note that we cannot re-map a single peripheral function to another pin, but must consider the entire set.

3.6 Interfacing to Simple IO

3.6.1 Driving LEDs

The **brightness** of an LED is proportional to the **current** passing through it. As LEDs are non-Ohmic, we cannot drive them directly with a voltage as this would result in an uncontrolled flow of current that may damage the LED or driver.
Instead, LEDs are paired with a **series resistor** to limit the flow of current. The appropriate current is dependent on the specific LED that is used and the capability of the driver device (microcontroller). A typical indictor LED requires a current of 1 m A to 2 m A.

3.6.2 Interfacing to LEDs

An LED can be driven in two different configurations from a microcontroller pin:

- **active high**; in which case the LED is **lit** when the pin is **HIGH**.
- **active low**; in which case the LED is **lit** when the pin is **LOW**.

Both of these configurations have their benefits, and the best configuration depends entirely on the context.

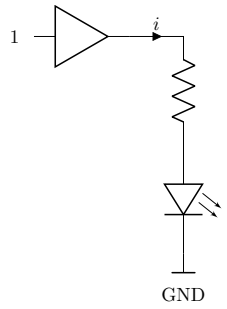


Figure 3.13: LED in an active high configuration.

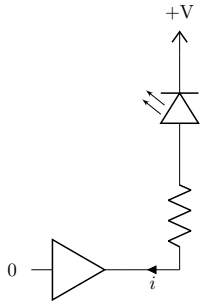


Figure 3.14: LED in an active low configuration.

On the QUTy, the LED display is driven in the **active low** configuration. This has a number of advantages:

- If the internal pull-up resistors are mistakenly enabled, no current will flow into the LEDs.
- The microcontroller pins can sink higher currents than they can source, allowing us to drive the display to a higher brightness.
- The display used on the QUTy has a common anode configuration, hence we must use an active low configuration to drive the display segments independently.

An LED is an example of a simple **digital output**, as we can map **logical states** to **LED states** (lit or unlit) for a digital output.

3.6.3 Switches as Digital Inputs

The state of a switch can be used to **set** the state of a pin. As the switch has two states (open or closed), these can be mapped directly to **logical states**.
This can be done by connecting the switch between the pin and voltage source representing one of the logic levels (ground or a positive supply).

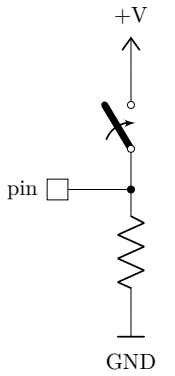


Figure 3.15: Switch in an active high configuration.

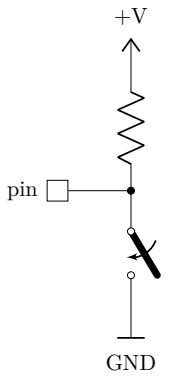


Figure 3.16: Switch in an active low configuration.

- When the switch is **open**, the pull-up/pull-down resistor is used to define the state of the switch.
- When the switch is **closed**, the state of the pin is defined by the voltage connected to via the switch.

3.6.4 Interfacing to Switches

As with LEDs, we can interface switches to microcontroller pins in two different configurations:

- **active high**; in which case the pin is **HIGH** when the switch is **closed**.
- **active low**; in which case the pin is **LOW** when the switch is **closed**.

An **active low** configuration is usually preferred as:

- it allows for the utilisation of an **internal pull-up resistor** that is commonly implemented in microcontrollers.
- It eliminates the risk of unsafe voltages being applied to the pin from the power supply in an active high configuration.
- It is easier to access a ground reference on a circuit board.

3.6.5 Interfacing to Integrated Circuits

For digital ICs,

- **Inputs** are typically **high impedance**
- **Outputs** are typically **push-pull**

This generally means that we can interface an IC by connecting its pins directly to the pins of a microcontroller.

- For **IC inputs**, the microcontroller pin is configured as an **output**, and the **microcontroller sets** the logic level of the net.
- For **IC outputs**, the microcontroller pin is configured as an **input**, and the **IC sets** the logic level of the net.

As microcontroller pins are typically configured as **inputs on reset**, a pull-up/pull-down resistor may be required if it is important for an IC input to have a **known state** prior to the configuration of the relevant microcontroller pins as outputs.

Part II

Assembly Programming

Definition 3.6.1 (Word)**.** A word refers to a value that is two bytes in size (16-bit).

3.7 Registers

A register refers to a memory location that is 1 byte in size (8-bit). The ATtiny1626 has 32 registers of which `r16` to `r31` can be loaded with an immediate value (0 to 255) using `ldi`.

```
1 ldi r16, 17
2 ldi r19, 0b10101010
3 ldi r31, 0xFF
```

Values are commonly loaded into registers as many other operations can be performed on them.

3.8 Flow Control

Instructions on the AVR Core increment the PC by 1 or 2 (depending on whether the OPCODE is 1 or 2 words) when they are executed so that any successive instructions are executed after the first. To divert execution to a different location, we can utilise **change of flow** instructions. The AVR Core has many change of flow instructions, that each have a different effect on the execution of the program.
The `jmp` (jump) instruction is used the simplest to jump to a different location in the program. This instruction is capable of jumping to an address withtin the entire 4M (words) program memory, however, this is highly excessive for the ATtiny1626.

3.9 Labels

Most change of flow instructions take an **address** in program memory as a parameter. Hence to make this process easier, we can use labels to refer to locations in program memory (and also RAM).

```
1 entry:
2     ldi r16, 0
3     jmp new_location ; Jump to the label new_location.
4     ldi r16, 1 ; This instruction is skipped
5
6 ; Label
7 new_location:
8     push r16
```

When a label appears in source code, the assembler replaces references to it with the address of the directive/instruction immediately following that label. Labels work for both **absolute** and **relative** addresses and the assembler will automatically adjust the address to the correct type.
Additionally, labels can also be used as parameters to other immediate instructions if we store the high and low bytes in registers and wish to reference the location in an indirect jumping instruction.

3.10 Absolute and Relative Addresses

`jmp` is a 32-bit instruction, which uses 22 bits to specify an address between `0x000000` and `0x3FFFFF`, or $2^{23} - 1$ bits of memory (8 MB). As mentioned earlier, this is much larger than what the 16-bit PC can address on the ATtiny1626 (64 KB).
As we will only need to jump within 64 KB of memory, it is inefficient to use the `jmp` instruction as it requires 3 CPU cycles to execute. Therefore, many AVR change of flow instructions take a value that is **added** onto the current PC to calculate the destination address, allowing them to fit within 16 bits.
The `rjmp` (relative jump) instruction works in the same manner as the absolute jump instruction. It only requires 2 CPU cycles and because the assembler calculates the relative address locations, we do not need to determine the relative distances.
Note the assembler throw an error if the address is not within the range of the PC.

3.11 Branching

A branching instruction jumps to a different location in response to something that occurs, i.e., user input, internal state, or other external factors. Many change of flow instructions are conditional, and will alter the PC differently based on register value(s) or flags. In AVR there are two main categories of branching instructions:

- Branch instructions
- Skip instructions

3.11.1 Branch Instructions

Branch instructions use the following logic:

1. Check if the specified flag in SREG is cleared/set
2. If true, jump to the specified address ($\text{PC} \leftarrow \text{PC} + k + 1$)
3. Otherwise, proceed to the next instruction as normal ($\text{PC} \leftarrow \text{PC} + 1$)

Although there are are 20 branch instructions listed in the instruction set summary, the following two form the basis of all branching instructions:

- `brbc` (branch if bit in SREG is cleared)
- `brbs` (branch if bit in SREG is set)

All other branching instructions are specific cases of the above instructions, that are provided to make programming in Assembly easier. As these instructions check the bits in the SREG, they are usually preceded by an ALU operation such as `cp` or `cpi` to trigger the required flags.
As only 7 bits are allocated to the destination in the OPCODE, branch instructions jump shorter distances than relative jumps.

3.11.2 Compare Instructions

Both the `cp` and `cpi` instructions are used to compare the values in one or two registers. The ALU performs a subtraction operation whose result is used to update the SREG. Note that the result is not stored or used in any way.

- `cp Rd, Rr` performs `Rd − Rr`
- `cpi Rd, K` performs `Rd − K`

Note `cp` compares the values in two registers, whereas `cpi` compares the value in a register with an immediate value. The result of this operation is used to set the appropriate flags in the SREG.

```
entry:
    ldi r16, 0
    ldi r19, 10
    cp r16, r19 ; Compare values in registers r16 and r19
    brge new_location ; Branch if r16 greater than or equal to r19

new_location:
```

Note that many instructions are able to set the Z flag, which is used to indicate if the result of the operation is zero. In these cases, the compare instruction may be redundant.

3.11.3 Skip Instructions

The skip instructions are less flexible then branch instructions, but can sometimes require less space or fewer cycles. Skip instructions skip the next instruction if the condition is true.
In this example we will skip the line which increments register 16.

```
cpse r16, r17 ; Skips next instruction if r16 == r17
inc r16 ; This is skipped

push r16 ; PC is now here
```

Same example which uses a branch instruction:

```
entry:
    cp r16, r17
    breq new_location ; Skips to new_location if r16 == r17
    inc r16 ; This is skipped

new_location:
    push r16 ; PC is now here
```

Note that the number of cycles for a skip instruction depend on the size of the instruction being skipped.
The **sbrc** and **sbrs** instructions are used to skip the next instruction if the specified bit a register is cleared/set.

```
ldi r16, 0b00101110

sbrc r16, 0 ; Skips next instruction if bit 0 of r16 is cleared
inc r16 ; This is skipped

push r16 ; PC is now here
```

Comparing with branch instructions

```
entry:
    ldi r16, 0b00101110

    andi r16, 0b00000001 ; Isolate bit 0

    breq new_location ; Skips next instruction if r16 == 0
    inc r16 ; This is skipped

new_location:
    push r16 ; PC is now here
```

The **sbis** and **sbic** instructions are used to skip the next instruction if the specified bit an I/O register is set/cleared. For example, if we wish to toggle the decimal point LED (DISP DP) on the QUTy (PORT B pin 5) when the first button (BUTTON0) was pressed (PORT A pin 4),

```
ldi r16, PIN5_bm ; Bitmask of pin 5
sbis VPORTA_IN, 0b00010000 ; Skip next instruction if pin 4 of PORT A is set
sts PORTB_OUTTGL, r16 ; Toggle the output driver of pin 5 on PORT B
```

Using branch instructions:

```
entry:
    in r17, VPORTA_IN ; Read the input register of PORT A
    andi r17, 0b00010000 ; Isolate pin 4

    brne new_location ; Skip instructions if r17 != 0

    ldi r16, PIN5_bm ; Bitmask of pin 5
    sts PORTB_OUTTGL, r16 ; Toggle the output driver of pin 5 on PORT B

new_location:
```

3.12 Loops

By jumping to an earlier address, we can loop over a block of instructions.

```
infinite_loop:
    ; Code to repeat

    rjmp infinite_loop
```

Loops can also be finite, in which case the loop will terminate when a counter reaches zero.

```
ldi r16, 10 ; Set counter to 10

loop:
    dec r16 ; Decrement counter
    brne loop ; Branch if counter != 0
```

Loops can also be used to repeat until some external event occurs.

```
main_loop:
    in r17, VPORTA_IN ; Read the input register of PORT A
    andi r17, 0b00010000 ; Isolate pin 4

    brne main_loop ; Branch if counter != 0

    rjmp button_pressed

button_pressed:
    ; Execute instructions

    rjmp main_loop ; Return to main loop
```

3.13 Delays

Loops can be utilised to delay the execution of instructions. These instructions do not execute any useful code. This is useful for when we wish to wait for an external event to occur.[2]
To create a precisely timed delay, we must take the following values into account.

- The clock speed — frequency of the clocks oscillations (default: 20 MHz — configurable in CLKCTLR_MCLKCTRLA)

[2]Note that this type of loop is not recommended for time-sharing systems, such as a personal computer, as the lost CPU cycles cannot be used by other programs. In these cases, clock interrupts are preferred. However, on a device such as the ATtiny1626, delay loops can be utilised to precisely insert delays in a program.

- The prescaler — reduces the frequency of the CPU clock through division by a specific amount; 12 different settings from 1x to 64x (default: 6 — configurable in CLKCTLR_MCLKCTRLA)

The clock oscillates at its effective clock speed:

$$\text{effective clock speed} = \text{clock speed} \times \frac{1}{\text{prescaler}}$$

The default prescaler is 6, so the effective clock speed is 3.33 MHz by default. Note that the effective clock speed can therefore range between:

- Effective maximum clock frequency: 20 MHz (20 MHz clock & prescaler 1)[3]
- Effective minimum clock frequency: 512 Hz (32.768 kHz clock & prescaler 64)

Therefore to create a delay, we must first determine the required number of CPU cycles in the body of the loop and iterate until the number of CPU cycles reaches the required amount.
The following examples utilise counters of various sizes to create delays. Note that n represents the number of iterations.

```
1  delay_1:
2      ldi r16, 255 - n ; 1 CPU cycle
3
4      ; Incrementor
5      ldi r17, 1 ; 1 CPU cycle
6
7      loop:
8          add r16, r17 ; 1 CPU cycle
9          brcc loop ; 2 CPU cycles (1 CPU cycle when condition is false)
```

The total number of CPU cycles is:

$$\begin{aligned}\text{total cycles} &= 1 + 1 + (n+1) + 2n + 1 \\ &= 3n + 4\end{aligned}$$

for a maximum delay of 230.7 µs ($(3 \times (2^8 - 1) + 4)\,T$)[4]. To create larger delays, we can use multiple registers:

```
1  delay_2:
2      ldi r24, x ; 1 CPU cycle
3      ldi r25, y ; 1 CPU cycle
4
5      loop:
6          adiw r24, 1 ; 2 CPU cycles
7          brcc loop ; 2 CPU cycles (1 CPU cycle when condition is false)
```

[3] As the QUTy is supplied with 3.3 V, it is not safe to go above 10 MHz.
[4] T is the period of one CPU cycle (using the default clock configuration): $T = \frac{1}{20\,\text{MHz}/6} = 300\,\text{ns}$.

The register pair $(y : x)$ has the following relationship:

$$(y : x) = (2^{17} - 1) - n \iff n = (2^{17} - 1) - (y : x)$$

with

$$\begin{aligned} \text{total cycles} &= 1 + 1 + 2\,(n + 1) + 2n + 1 \\ &= 4n + 5 \end{aligned}$$

for a maximum delay of 78.644 ms ($(4 \times (2^{16} - 1) + 5)\,T$). With three registers,

```
delay_3:
    ldi r24, x ; 1 CPU cycle
    ldi r25, y ; 1 CPU cycle
    ldi r26, z ; 1 CPU cycle

    loop:
        adiw r24, 1 ; 2 CPU cycles
        adc r26, r0 ; 1 CPU cycle (r0 represents a register with value 0)
        brcc loop ; 2 CPU cycles (1 CPU cycle when condition is false)
```

The register triplet $(z : y : x)$ is determined through:

$$(z : y : x) = (2^{25} - 1) - n \iff n = (2^{25} - 1) - (z : y : x)$$

with

$$\begin{aligned} \text{total cycles} &= 1 + 1 + 1 + 2\,(n + 1) + (n + 1) + 2n + 1 \\ &= 5n + 7 \end{aligned}$$

for a maximum delay of 25.166 s ($(5 \times (2^{24} - 1) + 7)\,T$). This approach can be extended to create delays of any length.
If needed, we can also include the `nop` (no operation) instruction which requires 1 CPU cycle and does nothing. In addition to this, we can also utilise nested loops, however the timing is more complex to determine.

3.14 Memory and IO

On the AVR Core, as both I/O and SRAM are accessed through the data space, they can be directly accessed using instructions that read/write to memory. This approach is known as memory-mapped I/O (MMIO) and it significantly reduces chip complexity.
In contrast to modern CPU architectures, such as x86, in the AVR architecture, programs are located in a separate address space (although the memory is still accessible through the data space).

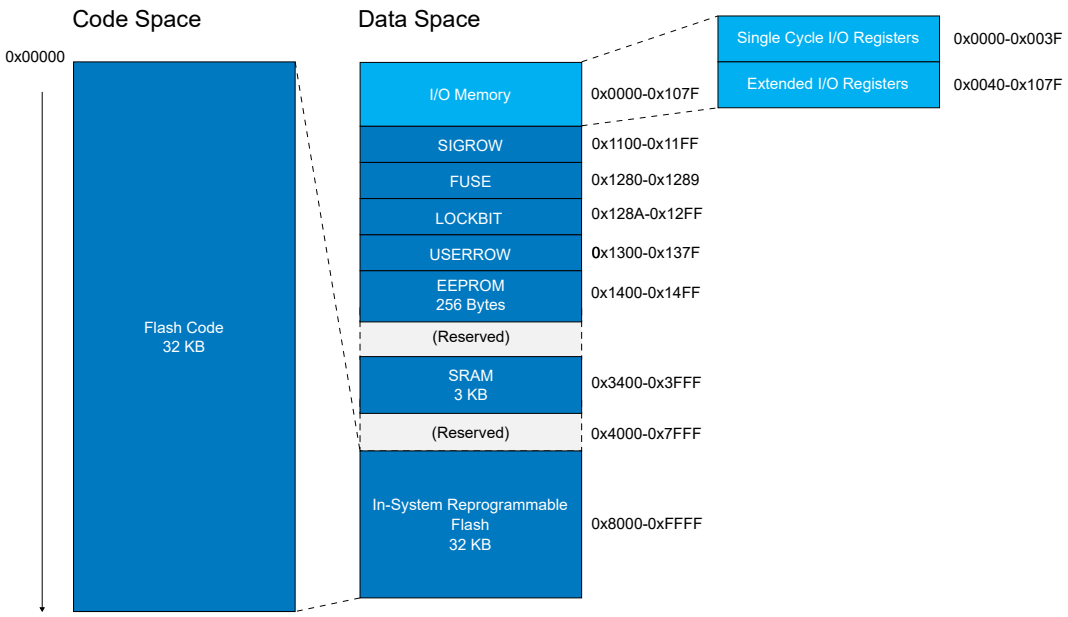


Figure 3.17: ATtiny1626 memory map.

The following instructions may be used to access memory from the data space:

- `lds` (load direct from data space to register)
- `sts` (store direct from register to data space)
- `ld` (load indirect from data space to register)
- `st` (store indirect from register to data space)
- `push`/`pop` (stack operations)
- `in`/`out` (single cycle I/O register operations)
- `sbi`/`cbi` (set/clear bit in I/O register)

Note that the `in`/`out` instructions can only access the low 64 bytes of the I/O register space and the `sbi`/`cbi` instructions can only access the low 32 bytes of the I/O register space.
As the name suggests, these instructions only require a single CPU cycle and hence several VPORT{A, B, C} (virtual ports) addresses are mapped to this location, so that they can be accessed through these single cycle instructions.
VPORT addresses simply hold a copy of their corresponding PORTs.

3.14.1 Load/Store Indirect

Although the `lds`/`sts` instructions can be used to access exact addresses of bytes, they are generally not suitable for accessing data structures such as arrays.

Instead, the `ld`/`st` instructions allow transfers between registers and the data space, where addresses are stored in a pointer register. These pointer registers are available for use only by `ld`/`st` through `X`, `Y`, and `Z`.
These 16-bit pointer registers occupy the same space as registers `r26` to `r31`.

- `r26` → `XL` (`X`-register low byte)
- `r27` → `XH` (`X`-register high byte)
- `r28` → `YL` (`Y`-register low byte)
- `r29` → `YH` (`Y`-register high byte)
- `r30` → `ZL` (`Z`-register low byte)
- `r31` → `ZH` (`Z`-register high byte)

For example, if we wanted to access a byte in RAM, we can do the following:

```
; Store address of RAM in X
ldi XL, lo8(RAMSTART)
ldi XH, hi8(RAMSTART)

ld r16, X ; Load byte from X to r16
; The byte in X is now in r16

ldi r17, 24
st X, r17 ; Store byte from r16 to X
; The byte in X is now 24
```

These pointer registers also support post-increment and pre-decrement operations:

- `X+` (post-increment pointer address)
- `-X` (pre-decrement pointer address)

```
ld r16, X+ ; Load byte from X to r16, then X <- X + 1
st X+, r16 ; Store byte from r16 to X, then X <- X + 1

ld r16, -X ; X <- X - 1, then load byte from X to r16
st -X, r16 ; X <- X - 1, then store byte from r16 to X
```

This operation can be used to copy bytes from one location to another:

```
; Copy 10 bytes from RAM to RAM+10
ldi XL, lo8(RAMSTART)
ldi XH, hi8(RAMSTART)

```

```
ldi YL, lo8(RAMSTART+10)
ldi YH, hi8(RAMSTART+10)

ldi r16, 10 ; Loop 10 times
loop:
    ld r0, X+ ; Load byte from X to r0, then X <- X + 1
    st Y+, r0 ; Store byte from r0 to Y, then Y <- Y + 1
    dec r16
    brne loop
```

3.14.2 Load/Store Indirect with Displacement

In addition to the `ld`/`st` instructions, the `ldd`/`std` instructions are a special form that allow us to load/store from/to the address of the pointer register **plus** $q = \{0 \text{ to } 63\}$.

```
ldi YL, lo8(RAMSTART)
ldi YH, hi8(RAMSTART)

ldd r0, Y+20 ; Load byte from Y+20 to r0
std Y+21, r0 ; Store byte from r1 to Y+21
; Note Y still points to RAMSTART
```

Note this form is only available for `Y`, and `Z`.

3.15 Stack

The stack is a last-in first-out (LIFO) data structure in SRAM. It is accessed through a register called the stack pointer (SP), which is not part of the register file like SREG.
Upon reset, SP is set to the last available address in SRAM (`0x3FFF`), and can be modified through `push`/`pop` and other methods that are generally not recommended.

- `push` stores a register to SP then decrements SP ($\text{SP} \leftarrow \text{SP} - 1$)
- `pop` increments SP ($\text{SP} \leftarrow \text{SP} + 1$) then loads to a register from SP

If a particular register is required without modifying other code, we can temporarily store the value of that register on the stack, and pop it back when we are done:

```
; Temporarily store Z on the stack
push ZL
push ZH

; Z may be used for another purpose

; Restore Z from the stack
```

```
pop ZH
pop ZL
```

Notice that the values are popped in reverse order.

3.16 Procedures

Procedures allow us to write modular, reusable code which makes them powerful when working on complex projects. Although they are usually associated with high level languages as methods, or functions, they are also available in assembly.
Procedures begin with a label, and end with the `ret` keyword. They must be **called** using the `call`/`rcall` instructions.

```
procedure:
    ; Procedure body
    ret ; Return to caller
```

3.16.1 Saving Context

To ensure that procedures are maximally flexible and place no constraints on the caller, we must always restore any modified registers before returning to the caller. The same is true for the SREG.

```
rjmp main_loop

procedure:
    push r16 ; Save r16 on the stack

    ; Procedure body
    ; Code that modifies r16
    ldi r16, 0x12
    loop:
        dec r16
        brne loop

    pop r16 ; Restore r16 from the stack
    ret

main_loop:
    ldi r16, 10
    rcall procedure ; Call procedure

    push r16 ; r16 should still be 10
```

3.16.2 Parameters and Return Values

Parameters can be passed using registers or the stack depending on the size of the inputs.

```
rjmp main_loop

; Calculate the average of two numbers
; Inputs:
;     r16: first number
;     r17: second number
; Outputs:
;     r16: average
average:
    ; Save r0
    push r0

    ; Save SREG
    in r0, CPU_SREG
    push r0

    ; Calculate average
    add r16, r17
    ror r16

    ; Restore SREG
    pop r0
    out CPU_SREG, r0

    ; Restore r0
    pop r0

    ; Return
    ret

main_loop:
    ; Arguments
    ldi r16, 100
    ldi r17, 200

    ; Call procedure
    rcall average
```

Using the stack:

```
rjmp main_loop

```

```
3  ; Calculate the average of two numbers
4  ; Inputs:
5  ;     top two values on stack
6  ; Outputs:
7  ;     r16: average
8  average:
9      ; Save Z
10     push ZL
11     push ZH
12
13     ; Save SREG
14     in ZL, CPU_SREG
15     push ZL
16
17     ; Get SP location
18     in ZL, CPU_SPL
19     in ZH, CPU_SPH
20
21     ; Save r17
22     push r17
23
24     ; Get first number
25     ldd r16, Z+7
26     ; Get second number
27     ldd r17, Z+6
28
29     ; Calculate average
30     add r16, r17
31     ror r16
32
33     ; Restore r17
34     pop r17
35
36     ; Restore SREG
37     pop ZL
38     out CPU_SREG, ZL
39
40     ; Restore Z
41     pop ZH
42     pop ZL
43
44     ; Return
45     ret
46
47 main_loop:
48     ; Arguments
```

```
49      ldi r16, 100
50      push r16
51      ldi r16, 200
52      push r16
53
54      ; Call procedure
55      rcall average
56
57      ; Remove arguments from the stack
58      pop r0
59      pop r0
```

Note that it is preferrable to return values using registers.

Part III

C Programming

C is a programming language developed in the early 1970s by Dennis Richie. C is a compiled language, meaning that a separate program is used to efficiently translate the source code into assembly. Its compilers are capable of targetting a wide variety of microprocessor architectures and hence it is used to implement all major operating system kernels. Compared to many other languages, C is a very efficient programming language as its constructs map directly onto machine instructions.

Chapter 4

Introduction

4.1 Main Function

C is a procedural language and hence all code subsides in a procedure (known as a **function**). In C, the `main` function is the **entry point** to the program. Program execution will generally begin in this function, where we can make calls to other functions.

```c
int main()
{
    // Function body
    return 0;
}
```

The purpose of returning a zero at the end of the `main` function is to signify the **exit status code** of the process. An exit status of `0` is traditionally used to indicate success, while all non-zero values indicate failure.

4.2 Statements

C programs are made up of statements. Statements are placed within scopes (indicated by braces (`{}`)) and are executed in the order they are placed. All statements in C must terminate with a semicolon (`;`). Although assembly instructions translate to a single OPCODE, a single C statement can translate to multiple OPCODEs.

```c
int main()
{
    int x = 3;
    {
        int y = 4;
        x = x + y;
    }
```

```
8       // x is now 7
9       // y is no longer in scope
10      return 0;
11  }
```

4.3 Comments

C supports two styles of comments. The first of these are known as "C-style comments", which allow multi-line/block comments. Multi-line comments use the `/* */` syntax.

```
1   /*
2       This is a multi-line comment.
3       It can span multiple lines.
4   */
```

The second style is known as "C++-style comments", which allow single-line comments. These comments are denoted by the `//` syntax.

```
1   // This is a single-line comment.
2   int x = 3; // It can be placed after a statement.
```

All comments in C are ignored by the compiler.

Chapter 5

Variables

Variables are used to temporarily store values in memory. Variables have a **type** and a **name** and must be declared before use.

5.1 Declaration

To declare a variable in C, we must specify the type and name of that variable.

```
1 int x;
```

This variable can then be **assigned to** using the = operator.

```
1 x = 4;
```

5.2 Initialisation

To optionally assign a value during declaration, we can apply the assignment operator after the declaration. This is known as a variable **initialisation**, as we are assigning an initial value to the variable.

```
1 int x = 4;
```

Note that using **uninitialised variables** results in **unspecified behaviour** in C, meaning that the value of such variables is unpredictable.

5.3 Types

While AVR assembly supports 8-bit registers, C supports larger data types by treating them as a sequence of bytes. We can also create compound data types with `struct` and `union`.

5.3.1 Type Specifiers

Type specifiers in declarations define the type of the variable. The `signed char`, `signed int`, and `signed short int`, `signed long int` types, together with their `unsigned` variants and `enum`, are all known as **integral** types. `float`, `double`, and `long double` are known as **floating** or **floating-point** types. Arithmetic between
The following table summarises various numeric types in C:

Description	Size	Equivalent Definitions
Character data	1 B	`signed char c; char c;`
Signed short	2 B	`signed short int s; signed short s; short s;`
Unsigned short	2 B	`unsigned short int us; unsigned short us;`
Signed integer	4 B	`signed int i; signed i; int i;`
Unsigned integer	4 B	`unsigned int ui; unsigned ui;`
Signed long	8 B	`signed long int l; signed long l; long l;`
Unsigned long	8 B	`unsigned long int ul; unsigned long ul;`
Single precision floating	4 B	`float f;`
Double precision floating	8 B	`double d;`
Long double precision floating	16 B	`long double ld;`

Note that the size of these types is not necessarily the same across platforms, hence it is discouraged to use these keywords for platform specific tasks. *See the section on Exact Width Types for more information.*

5.3.2 Type Qualifiers

Types can be qualified with additional keywords to modify the properties of the identifier. Three common qualifiers are **const**, **static**, and **volatile**.

- `const` — indicates that the variable is **constant** and cannot be modified.
- `static` — indicates that the variable has a global lifetime (maintains value between function invocations).
- `volatile` — indicates that the variable can be modified or accessed by other programs or hardware.

5.3.3 Portable Types

C has a set of standard types that are defined in the language specification, however the type specifiers shown above may have different storage sizes depending on the platform. Although this may be insignificant for most platforms, microcontrollers use specific sizes for registers, meaning it is important to refer to the correct type specifiers when declaring a variable.

5.3.4 Exact Width Types

The standard integer (`stdint.h`) library provides **exact-width** type definitions that are specific to the development platform. This ensures that variables can be initialised with the correct size on any platform.

```
#include <stdint.h>

int main()
{
    int8_t i8;
    int16_t i16;
    int32_t i32;
    int64_t i64;

    uint8_t ui8;
    uint16_t ui16;
    uint32_t ui32;
    uint64_t ui64;

    return 0;
}
```

5.3.5 Floating-Point Types

The `float` and `double` types can store **floating-point** value types in C. Their implementation allows for variable levels of precision, i.e., extremely large and extremely small values. These types are very useful on systems with a floating point unit (FPU) or equivalent.
As the ATtiny1626 does not have an FPU, arithmetic involving floating point values is highly inefficient. Therefore, integer arithmetic should be utilised when possible. Note that a single floating point number or operation causes the entire floating point library to be included which can require a large amount of memory.

Chapter 6

Literals

6.1 Integer Prefixes

Integer literals are assumed to be base 10 unless a prefix is specified. C supports all of the following prefixes:

- **Binary** (base 2) — `0b`
- **Octal** (base 8) — `0`
- **Decimal** (base 10) — no prefix
- **Hexadecimal** (base 16) — `0x`

6.2 Integer Suffixes

Integer literals can be suffixed to specify the size/type of the value:

- **Unsigned** — `U`
- **Long** — `L`
- **Long Long** — `LL`

Suffixes are generally only required when clarifying ambiguity of values where the user wishes to use a different type than the default type.

```c
#include <stdio.h>

int main()
{
    printf("%d\n", 2147483648); // Treated as signed integer and throws warning
    printf("%d\n", 2147483648U); // Treated as unsigned integer

    return 0;
}
```

6.3 Floating Point Suffixes

As with integer types, floating point values can also be suffixed to specify which type to use.

- **Float** — `f`
- **Double** — `d`

6.4 Character and String Literals

- **Character** — surrounded by single quotes `'A'`
- **String** — surrounded by double quotes `"Hello World"`

Chapter 7

Flow Control

7.1 If Statements

C provides a standard branching control structure known as an `if` statement. This structure tests a condition and executes a block of code if that condition is **true**.

```
if (condition)
{
    // Code to execute if condition is true
}
```

This structure can be nested and also supports `else` and `else if` statements.

```
if (x > 1)
{
    // Code to execute if x is greater than 1
    if (x < 10)
    {
        // Code to execute if x is greater than 1 and less than 10
    }
} else if (x < -1)
{
    // Code to execute if x is less than -1
    if (x > -10)
    {
        // Code to execute if x is less than -1 and greater than -10
    }
} else
{
    // Code to execute if x is not greater than 1 and not less than -1
}
```

7.2 While Loops

The simplest loop structure in C is achieved by using a `while` loop. This loop executes a block of code while the condition is **true**.

```
while (condition)
{
    // Code to execute while condition is true
}
```

An `do` while loop is similar to a `while` loop, but the loop will execute at least once.

```
do
{
    // Code to execute at least once
} while (condition);
```

This loop structure is typically accompanied by a looping variable known as an iterator:

```
int i = 0; // Iterator

// Execute code 10 times
while (i < 10)
{
    // Code to execute while i is less than 10

    i++; // Increment i by 1
}
```

7.3 For Loops

`for` loops are similar to `while` loops, but they usually result in more understandable code.

```
for (initialisation; condition; increment)
{
    // Code to execute while condition is true
}
```

Note the initialisation and increment statements are optional, and while the condition statement is also optional, we must ensure that the loop can terminate from within the structure (see next section).

7.4 Break and Continue Statements

`break` and `continue` statements are used to terminate a loop early.

```c
for (int i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break; // Terminate loop early
    }
    printf("%d\n", i);
}
```

```c
for (int i = 0; i < 10; i++)
{
    if (i == 5)
    {
        continue; // Skip current iteration and continue with next iteration
    }
    printf("%d\n", i);
}
```

If the loop is nested within another loop, the `break` and `continue` statements will only terminate the innermost loop.

```c
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        if (j == 5)
        {
            break; // Terminate inner loop early
        }
        printf("%d\n", j);
    }
}
```

Chapter 8

Expressions

C provides a number of operators which can be used to perform arithmetic/logical operations on values. C follows the same precedence rules as mathematics, however caution should be used when comparing precedence of certain logical and bitwise operations.

8.1 Operation Precedence

Operation	Operator Symbol	Associativity
Postfix	`++`, `--`	
Function call	`()`	
Array subscripting	`[]`	Left to right
Member access	`.`	
Member access through pointer	`->`	
Prefix	`++`, `--`	
Unary	`+`, `-`	
Logical NOT and bitwise NOT	`!`, `~`	
Type cast	`(type)`	Right to left
Dereference	`*`	
Address-of	`&`	
Size-of	`sizeof`	
Multiplicative	`*`, `/`, `%`	Left to right
Additive	`+`, `-`	Left to right
Bitwise shift	`<<`, `>>`	Left to right
Relational	`<`, `>`, `<=`, `>=`	Left to right
Equality	`==`, `!=`	Left to right
Bitwise AND	`&`	Left to right
Bitwise XOR	`^`	Left to right
Bitwise OR	`|`	Left to right
Logical AND	`&&`	Left to right
Logical OR	`||`	Left to right
Conditional	`? :`	Right to left
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`	Right to left
Sequential evaluation	`,`	Left to right

8.2 Arithmetic Operations

All arithmetic operations work as expected, noting that integer division is truncated.
If an arithmetic operation causes a type overflow, the result will depend on the type. For signed integers, the result of an overflow is **undefined** in C. For unsigned integers, the result is truncated to the type size (or the value modulo the type size).

8.3 Operator Types

- **Unary** operators — have a single operand. For example, `++` and `--`, or `+` and `-`.
- **Binary** operators — have two operands. For example, `+`, `-`, `*`, and `/`.
- **Ternary** operators — have three operands. For example, `? :`.

8.4 Assignment

To assign a value to a variable, use the assignment (=) operator.

```
int x = 5;
```

8.5 Multiple Assignment

If we want to assign values to multiple variables of the same type, we can use the comma (,) operator.

```
int x = 1, y = 2, z = 3;
```

We can also use the assignment (=) operator to assign the same value to multiple variables of the same type.

```
int x, y, z;
x = y = z = 5;
```

8.6 Compound Assignment

Compound assignment operators perform the operation specified by the additional operator, then assign the result to the left operand.

```
char x = 0b11001010;
x |= 0b00000001; // x = 0b11001010 | 0b00000001 = 0b11001011

int y = 25;
y += 5; // y = 25 + 5 = 30

char z = 0b10000010;
z <<= 1; // z = 0b10000010 << 1 = 0b00000100
```

8.7 Bitwise Operations

Binary operators behave as expected in C.

```
char x = 0b11001010;
unsigned char y = 0b01100001;

char a = ~x;     // a = ~0b11001010 = 0b00110101
```

```
char b = x & y;  // b = 0b11001010 & 0b01100001 = 0b01000000
char c = x | y;  // c = 0b11001010 | 0b01100001 = 0b11101011
char d = x ^ y;  // d = 0b11001010 ^ 0b01100001 = 0b10101011

char e = x << 1; // e = 0b11001010 << 1 = 0b10010100
char f = x >> 1; // f = 0b11001010 >> 1 = 0b11100101
char g = y >> 1; // g = 0b01100001 >> 1 = 0b00110000
```

Note that right shifts are automatically sign-extended in C.

8.8 Relational Operations

Relational operators can be used to compare two values.

```
int x = 5;
int y = 10;
int z = 15;

if (x < y) {
    printf("x is less than y\n");
}

if (x != 15)
{
    printf("x is not equal to 15\n");
}
```

8.9 Logical Operations

Logical operators can be used to combine two boolean expressions.

```
int x = 5;
int y = 10;
int z = 15;

if (x < y && x != 15) {
    printf("x is less than y and x is not equal to 15\n");
}
```

8.10 Increment and Decrement

Increment and decrement operators are unary operators that can be used to increment or decrement a variable by 1.

```
int x = 5;
x++; // x = 6
```

The increment and decrement operators can be used as either prefix or postfix operators.

```
int x = 5;
int y = x++; // y = 5, x = 6
int z = ++x; // z = 7, x = 7
```

Prefix operators are evaluated before the statement is executed, while postfix operators are evaluated after the statement is executed.

Chapter 9

Preprocessor

The preprocessor processes C code before it is passed onto the compiler. The preprocessor strips out comments, handles **preprocessor directives**, and replaces macros. Preprocessors begin with the `#` character and no non-whitespace characters can appear on the line before the preprocessor directive.

9.1 Includes

The `#include` directive is used to include the contents of another file into the current file. This directive has two forms.

- `#include <filename>` — include header files for the C standard library and other header files associated with the target platform.
- `#include "filename"` — include programmer-defined header files that are typically in the same directory as the file containing the directive.

When this directive is used, it is equivalent to copying the contents of the file into the current file, at the location of the directive. The included file is also preprocessed and may contain other include directives.

9.2 Header Files

Object files containing **compiled code** can be linked into a program to allow programmers to call existing functions. For C to have knowledge of the functions in this object file, the authors of those functions should store the function prototypes in a **header file**.
Header files end in the `.h` extension. They can be included into the source file using the `#include` directive and can significantly reduce compile times by reducing the amount of code that needs to be compiled.
In the following example, we will define an `add` function and include it into another C program.

```c
// add.c
int add(int x, int y)
```

```
{
    return x + y;
}
```

This file is compiled to `add.o`. To allow the `add` function to be called from the main program, we need to create a header file containing the function prototype of `add`.

```
// add.h
int add(int x, int y);
```

We can then include this header file into the main program.

```
// main.c
#include <stdio.h> // Include printf definition
#include "add.h" // Include add function definition

int main()
{
    int x = 5;
    int y = 10;

    int z = add(x, y);
    printf("%d\n", z);

    return 0;
}
```

9.3 Definitions

The *#define* directive is used to define **preprocessor macros**. Whenever these macros appear in the source file, they are replaced with the value specified by the macro. Macros are a simple text replacement mechanism, an thus must be defined carefully to avoid invalid code from being generated.

```
#include <stdio.h>
#define PI 3.14159265358979

int main()
{
    printf("%f\n", 2 * PI);
    return 0;
}
```

Aside from constant values, macros can also be used to create small compile-time “functions”, that expand to code:

```c
#include <stdio.h>
#define MAX(x, y) ((x) > (y) ? (x) : (y))

int main()
{
    int x = 5;
    int y = 10;

    int z = MAX(x, y);
    printf("%d\n", z);

    return 0;
}
```

Note that the semi-colon is omitted at the end of the macro definition, as it would also be substituted into the program.
Only a single preprocessor directives can appear on a line, and the directives must occupy a single line (note that a backslash (\) can be used to break long lines).

Chapter 10

Pointers

When a variable is declared, the compiler automatically allocates a block of memory to store that variable. If we want to access this block of memory indirectly, we must use a **pointer**. In C, pointers are declared as "pointing to" an object of another type.

```
uint8_t *ptr; // Pointer to a uint8_t variable
```

This code declares a variable `ptr` that points to a `uint8_t`. Note this is not the same as declaring a `uint8_t` variable.
Internally, a pointer contains a **memory address**, which on the ATtiny1626 is 16-bit wide.

10.1 Addressing

When the location we want to access is known in advance, pointers can be declared with a specific address:

```
volatile uint8_t *ptr = (volatile uint8_t *)0x0421; // The address of PORTB DIRSET
```

Note the type-cast will be discussed in the next sections.
A more common usage of pointers is to *reference* **other variables**.

```
uint8_t x = 5;
uint8_t *ptr = &x; // Address of x
```

The amperstand (`&`) operator is used to return the **address of** the variable `x`. Here the pointer type of `ptr` must match the type of `x`.

10.2 Dereferencing

Once we have a pointer, we can access the value at the address it points to using the **unary dereference** operator (`*`).

```c
uint8_t x = 5;
uint8_t *ptr = &x; // Address of x

// Read the value at the address pointed to by ptr
uint8_t y = *ptr; // y = Value at ptr = 5

// Write the value 10 to the address pointed to by ptr
*ptr = 10; // Value at ptr := 10
// c = 10 but y = 5
```

This is also known as **indirection**, as we are *indirectly accessing* a value through a pointer.

10.3 Strings

In C, strings are represented as arrays of characters, terminated by a character with the value 0. Strings are declared using double quotes (`""`) and are automatically terminated by a null character.

```c
char *str = "Hello World";

printf("%s\n", str); // Prints "Hello World\n"

// Because str is a pointer, it can be printed directly.
printf(str); // Prints "Hello World"
printf("\n"); // Prints "\n"
```

In the example above, the compiler automatically allocates a block of memory to store the string, which in this case is 12 bytes long (11 characters + null terminator).
The pointer `str` points to the first character in the string.

```c
char *str = "Hello World";
*str == 'H'; // True
```

When using the `printf` function, the null terminator is required to indicate the end of the string. We will see how to index into strings in the section on arrays.

10.4 Qualifiers

Various **qualifiers** can be used to modify the type of a pointer. Typically these qualifiers apply to the memory pointed to by the pointer.
If the variable which the pointer points to is **constant**, the dereference operator cannot be used to reassign the value of the variable.

```c
const uint8_t a = 100; // Constant
uint8_t *ptr = &a; // Points to the constant `a`
```

```
3
4 *ptr = 200; // Error: Cannot modify `a` because `a` is constant
```

If the pointer is declared as **constant**, the pointer imposes a **read-only** restriction on the memory it points to.

```
1 uint8_t a = 100; // Variable
2 const uint8_t *ptr = &a; // Points to `a` but treats it as constant
3
4 *ptr = 200; // Error: Cannot modify `a` because `ptr` is constant
```

Note this does not mean that the pointer itself is constant, only that the memory it points to is constant. The following is valid:

```
1 uint8_t a = 100; // Variable
2 uint8_t b = 200; // Variable
3
4 const uint8_t *ptr = &a; // Points to `a` but treats it as constant
5 ptr = &b; // Valid: `ptr` is not constant
```

If the qualifier is placed after the asterisk, the pointer itself is constant, meaning that it cannot be reassigned to another address.

```
1 uint8_t a = 100; // Variable
2 uint8_t b = 200; // Variable
3
4 uint8_t *const ptr = &a; // Points to `a` but cannot be reassigned
5 *ptr = 200; // Valid: `ptr` points to `a` which is not constant
6 ptr = &b; // Error: Cannot reassign `ptr`
```

If we wish, we can apply the qualifiers to both the pointer and the variable which that pointer points to.

```
1 uint8_t a = 100; // Variable
2 uint8_t b = 200; // Variable
3
4 const uint8_t *const ptr = &a; // Points to `a` but cannot be reassigned nor modified
5 ptr = &b; // Error: Cannot reassign `ptr`
6 *ptr = 200; // Error: Cannot modify `a` because `ptr` is constant
```

All of the above also applies to the `volatile` qualifier.

10.4.1 Pointers to Pointers

Pointers can also point to other pointers.

```c
uint8_t a = 100; // Variable

uint8_t *ptr = &a; // Points to `a`
uint8_t **ptr2 = &ptr; // Points to `ptr`
```

This can be used to modify the **address** of a pointer indirectly.

```c
uint8_t a = 100; // Variable
uint8_t b = 200; // Variable

uint8_t *ptr = &a; // Points to `a`
uint8_t **ptr2 = &ptr; // Points to `ptr`

*ptr2 = &b; // `ptr` now points to `b`
```

For high levels of indirection, we can use more asterisks, although this is uncommon.
Qualifiers can also be applied to pointers to pointers.

```c
uint8_t a = 100; // Variable

uint8_t *ptr = &a; // Points to `a`

const uint8_t **ptr1 = &ptr; // Pointer to pointer to constant uint8_t

uint8_t * const *ptr2 = &ptr; // Pointer to constant pointer to uint8_t

uint8_t ** const ptr3 = &ptr; // Constant pointer to pointer to uint8_t
```

10.4.2 Pointer Arithmetic

Pointers can be changed with arithmetic operators such as + and −. Arithmetic on pointers affects the address of the pointer, so that the pointer points to another location.
When performing arithmetic on pointers, the size of an increment is determined by the type of the variable that the pointer is pointing to.

```c
uint8_t a = 100; // Variable

uint8_t *ptr = &a; // Points to `a`

ptr++; // Increment by 1 byte (size of uint8_t)
// ptr now points to the next byte after `a`
```

10.4.3 Void Pointers

When a pointer needs to point to a memory address of an unknown type, it can be declared with the `void` keyword.

```
void *ptr;
```

Void pointers have no type, so they cannot be dereferenced. Pointers of other types can be assigned to void pointers, but not vice versa.

```
uint8_t a = 100;

void *ptr = &a; // Pointer to uint8_t
uint8_t *ptr2 = ptr; // Error: Cannot assign void pointer to uint8_t pointer
```

10.4.4 Size-of

The `sizeof` function can be used to determine the size of a variable in bytes.

```
uint8_t a = 100;
uint16_t b = 200;

sizeof(a); // Returns 1
sizeof(b); // Returns 2
```

10.5 Arrays

Array types are used to hold multiple values of the same type in a contiguous block of memory. Arrays can be declared in the following ways:

```
uint8_t a[10]; // Array of 10 uint8_t
uint8_t b[10] = {0}; // Array of 10 uint8_t initialized to 0
uint8_t c[] = {1, 2, 3}; // Array of 3 uint8_t initialized to 1, 2, 3
uint8_t d[5] = {1, 2, 3}; // Array of 5 uint8_t initialized to 1, 2, 3, 0, 0
```

The brace (`{ }`) syntax can only be used to initialise an array and if the length of the array which is being assigned is less than the length of the array being assigned to, the remaining values will be set to 0.

10.5.1 Character Arrays

A character array is a special type of array which is used to store strings. Character arrays can be declared using the `char` keyword.

```c
char a[] = "Hello World";

// Equivalent to:
char b[12] = {'H', 'e', 'l', 'l', 'o', ' ',
              'W', 'o', 'r', 'l', 'd', '\0'};
```

This method allocates 12 bytes of SRAM and initialises those bytes with the string `"Hello World"`. This means that the string can be modified later in the program. If we use the `const` keyword, the string will be stored in flash memory and cannot be modified.

```c
const char a[] = "Hello World";
```

10.5.2 Indexing

Array elements can be accessed with the array index operator (`[ ]`). In C, array indices start at 0.

```c
uint8_t a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

a[0]; // Returns 0
a[1] = 10; // a is now {0, 10, 2, 3, 4, 5, 6, 7, 8, 9}
```

It is undefined behaviour to access an array element which is out of bounds. However it is possible to have a pointer to an element one past the end of an array as long as the pointer is not dereferenced.

```c
uint8_t a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

uint8_t *ptr = &a[10];
```

To loop through an array, we can use a `for` loop.

```c
uint8_t a[10];

for (uint8_t i = 0; i < 10; i++) {
    a[i] = i;
}
```

10.5.3 Pointers and Arrays

Arrays are implicitly converted to pointers to the first element of the array.

```c
uint8_t a[10];

uint8_t *ptr = a; // Equivalent to `uint8_t *ptr = &a[0]`
*ptr = 100; // `a` is now {100, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

This is especially useful when passing arrays to functions, as arrays cannot be passed to functions by value, but rather the pointer to that array can. This lets us index into an array in a function and the changes will be reflected in the original array.

```
void func(uint8_t *arr) {
    arr[0] = 100;
}

uint8_t a[10];

func(a); // `a` is now {100, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

The syntax `arr[i]` is equivalent to `*(arr + i)`. Note that this is possible because arrays are stored contiguously in memory.
Note that it is not possible to change an array's address.

```
uint8_t a[10];

a++; // Error: Cannot change the address of an array
```

10.5.4 Array Length

The length of an array can be determined with the `sizeof` function.

```
uint8_t a[10];
uint16_t b[5];

sizeof(a) / sizeof(a[0]); // Returns 10
sizeof(b) / sizeof(b[0]); // Returns 5
```

We divide by the size of the first element of the array because the type of the array may be larger than 1 byte.

10.5.5 Copying Arrays

Arrays can be copied in two ways. The first way is to use a `for` loop.

```
uint8_t a[10];
uint8_t b[10];

for (uint8_t i = 0; i < sizeof(a) / sizeof(a[0]); i++) {
    b[i] = a[i];
}
```

The second way is to use the `memcpy` function from the `string.h` library.

```c
uint8_t a[10];
uint8_t b[10];

memcpy(b, a, sizeof(a));
```

10.5.6 Multidimensional Arrays

Multidimensional arrays (also known as multiple subscript arrays) are used to hold multi-dimensional data.

```c
uint8_t a[][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

To declare a multi-dimensional array, all dimensions but the first need to be specified. The rows of the array must be specified within additional braces (`{ }`).
Elements can be accessed by specifying the index of each dimension.

```c
a[0][0]; // Returns 1
a[1][2]; // Returns 6
```

These arrays are also stored contiguously in memory, in **row-major** order, and hence pointer arithmetic is performed differently.

```c
uint8_t a[][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

uint8_t rows = 3;
uint8_t cols = 3;

for (uint8_t i = 0; i < rows; i++) {
    for (uint8_t j = 0; j < cols; j++) {
        // Double indexing
        printf("%d ", a[i][j]);

        // Single indexing
        printf("%d ", a[i * cols + j]);

        // Pointer arithmetic
```

```
        printf("%d ", *(*(a + i) + j));
        // Equivalent to: printf("%d ", *(a[i] + j));
        // Each row is a pointer to the first element of that row
    }
}
```

10.6 Functions

Procedures are called functions in C. Functions can return values and take arguments. The main function is the entry point of a program.

```
int main(void) {
    return 0;
}
```

Functions in C must be declared in the top-level of a C program, and thus cannot be declared inside other functions. Functions are declared with the following syntax:

```
return_type function_name(param_type param_name, ...) {
    // Function body
}
```

10.6.1 Parameters

The parameters of a function are local variables scoped to that function.

```
uint8_t add(uint8_t a, uint8_t b) { // `a` and `b` are parameters of `add`
    return a + b;
}

int main(void) {
    uint8_t a = 10;
    uint8_t b = 20;

    uint8_t c = add(a, b); // `a` and `b` are arguments to `add`
    return 0;
}
```

To pass an array to a function, we can pass a pointer to that array. To do so, we must specify the length of the array as well.

```
void print_array(uint8_t *arr, uint8_t len) {
    for (uint8_t i = 0; i < len; i++) {
        printf("%d ", arr[i]);
```

```
    }
}

int main(void) {
    uint8_t a[10];

    for (uint8_t i = 0; i < sizeof(a) / sizeof(a[0]); i++) {
        a[i] = i;
    }

    print_array(a, sizeof(a) / sizeof(a[0]));
    return 0;
}
```

When a function does not take any arguments, we can specify `void` as the parameter list.

```
void func(void) {
    // Function body
}
```

10.6.2 Return Values

To return a value from a function, we use the `return` keyword. When a function does not return a value, we can specify `void` as the return type. Note that a void function does not need to use the `return` keyword.

10.6.3 Function Prototypes

C uses **single-pass** compilation, meaning that functions need to be declared before they can be called. Function prototypes are used to declare a function without having to specify the entire body of the function.

```
uint8_t add(uint8_t a, uint8_t b); // Function prototype

int main(void) {
    uint8_t a = 10;
    uint8_t b = 20;

    uint8_t c = add(a, b);
    return 0;
}

uint8_t add(uint8_t a, uint8_t b) {
    return a + b;
}
```

The compiler uses the function prototype to generate the code required to call the function without having to know the entire body of the function. The linker will then resolve all function calls to the appropriate function definitions. Note that parameter names are not required in function prototypes.

10.6.4 Passing by Reference

As seen previously, we can pass variables by value and arrays by reference through pointers.
As functions only return one value, we can use pointers to pass multiple values back to the caller. These output values are also passed to the functions parameter list.

```
1  void swap(uint8_t *a, uint8_t *b) {
2      uint8_t temp = *a;
3      *a = *b;
4      *b = temp;
5  }
6
7  int main(void) {
8      uint8_t a = 10;
9      uint8_t b = 20;
10
11     swap(&a, &b);
12     return 0;
13 }
```

10.6.5 Stack

As functions can call other functions, or even call themselves, local variables inside functions are stored on the **stack**. The return address of where a function is called from is also stored on the stack so that the program counter can be set to that address when the function returns.
Local variables inside functions do not increase the explicit SRAM usage reported by the compiler. Rather, this memory will be allocated on the stack when the function is called. Therefore it is important to ensure that the stack does not overflow, through recursive functions or large local variables.

10.7 Scope

Variables and other identifiers in C have scope. Scope affects the **visibility** and **lifecycle** of variables. Scope is **hierarchical**, meaning that variables declared in a parent scope are visible to all child scopes.
Variables declared in a child scope can also hide variables declared in a parent scope declared with the same name.

10.7.1 Global Scope

Variables declared outside of any function are declared in the global scope. Global variables are visible to all functions in a program.

```
uint8_t a = 10; // Global variable

int main(void) {
    uint8_t b = 20; // Local variable

    a++; // `a` is visible to `main`
    return 0;
}
```

Global variables are allocated a fixed location in SRAM and do not exist on the stack.

10.7.2 Local Scope

Variables declared inside a function are declared in the local scope. Their lifetime is limited to the function in which they are declared. By default, local variables go on the stack.

10.7.3 Block Scope

The block scope is a subset of the local scope. Variables declared inside blocks such as `if` statements have their own scope. These variables are only visible inside the block. We can create a new scope by using curly braces.

10.7.4 Static Variables

When applied to a **local variable**, the `static` keyword changes the lifetime of a variable to the lifetime of the program. This means that the variable will not be destroyed when the function returns, and will retain its value between function calls. Static variables are allocated in SRAM and not on the stack.
When applied to a **global variable**, the `static` keyword changes the visibility of the variable to the file in which it is declared.