Microprocessors and Digital Systems

Semester 2, 2022

Dr Mark Broadmeadow

Tarang Janawalkar

Contents

Part I
Microcontroller Fundamentals

1 Computer Systems Architecture

1.1 Introduction to Computers

Definition 1.1 (Computer)**.** A computer is a digital electronic machine that can be programmed to carry out sequences of arithmetic or logical operations (computations) automatically.

Definition 1.2 (Central Processing Unit)**.** The central processing unit (CPU) is a system of components that processes instructions, performs calculations, and manages the flow of data through a computer. It consists of several components such as the control unit, arithmetic logic unit, and processing core(s).

Definition 1.3 (Control unit)**.** The control unit is a component of the CPU responsible for managing the flow of instructions and data within the processor. It interprets program instructions, coordinates the activities of other CPU components, and ensures that operations are carried out in the correct sequence.

Definition 1.4 (Arithmetic logic unit)**.** The arithmetic logic unit (ALU) is a subsystem of the CPU that performs mathematical operations, such as addition and subtraction, as well as logical comparisons, such as equality or inequality checks. It is a critical component for executing calculations and decision-making tasks within a computer.

Definition 1.5 (Processing core)**.** A core is an independent processing unit within the CPU, capable of executing its own set of instructions. Modern CPUs may contain multiple cores, enabling parallel execution of tasks to improve efficiency and performance. Each core operates as a self-contained processor within the larger CPU system.

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 board used in this unit houses an AVR ATtiny1626 microcontroller. Some of the key features of this microcontroller are provided in the next section.

1.3 The AVR ATtiny1626 Microcontroller

The ATtiny1626 microcontroller has the following features:

- CPU: AVR Core (AVRxt variant)
- Memory:
  - Flash memory (16 K B) used to store program instructions in memory
  - SRAM (2 K B) used to store data in memory

  - EEPROM (256 B)
- 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 be erased in large chunks
- Typically used to store program data
- Generally read-only. Programmed via an external tool, which is loaded once and remains static during the lifetime of the program
- Writes are 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

1.3.4 The AVR Core

- 8-bit Reduced Instruction Set Computer (RISC)
- 32 working registers (R0 to R31)
- Program Counter (PC) — location in memory of the next instruction to execute
- Status Register (SREG) — stores key information from calculations performed by the ALU (i.e., whether a result is negative)
- Stack Pointer — location in memory of the top of the stack
- 8-bit core — all data, registers, and operations, operate within 8-bits

1.3.5 Status Register

The status register is an 8-bit register that stores the result of the last operation performed by the ALU. It has the following flags:

C Carry Flag

Z Zero Flag

N Negative Flag

V Two's Complement Overflow Flag

S Sign Flag

H Half Carry Flag

T Transfer Bit

I Global Interrupt Enable Bit

1.4 Computer Programming Basics

A computer program is a set of instructions written to perform a specific task or solve a problem. These instructions are processed by the CPU, which executes them step by step to produce the desired outcome. Programs are typically written in high-level programming languages, which are then translated into machine-readable formats for execution.

Definition 1.6 (Machine code)**.** Machine code is the lowest-level representation of a program, consisting of binary instructions (sequences of 0s and 1s) that the CPU can execute directly. Each instruction corresponds to a specific operation, such as arithmetic, memory access, or control flow, defined by the CPU's architecture.

Definition 1.7 (Assembly language)**.** Assembly language is a low-level programming language that provides a human-readable representation of machine code. It uses symbolic names (called operational codes) for operations and memory addresses, making it easier for programmers to write and understand code that is closely tied to the CPU's architecture. Assembly language must be translated into machine code by an assembler for execution.

1.5 Program Execution

At the time of reset, PC = 0 and the following steps are 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: increment once if the instruction is one word, otherwise increment twice. Control flow instructions may move the program to another location and, as a result, set the PC to a specific address.

This is illustrated in the following figure:

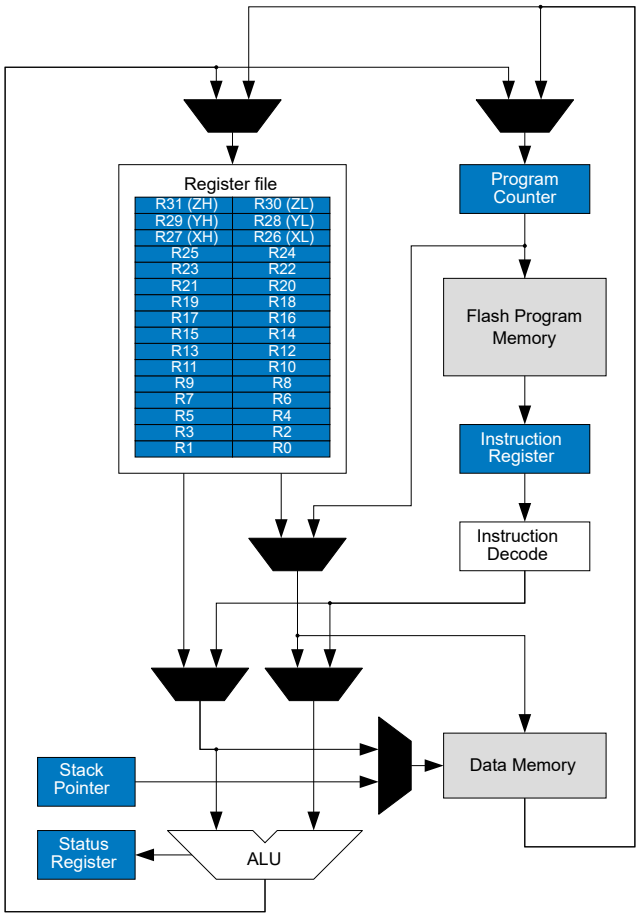


Figure 1: Program execution on the ATtiny1626.

1.5.1 Instructions

- The CPU understands and can execute a limited set of instructions — ~88 unique instructions for the ATtiny1626
- Instructions are encoded in program memory as opcodes. Most instructions are two bytes long, but some instructions are four bytes long

- The AVR Instruction Set Manual describes all the available instructions, and how they are translated into opcodes
- Instructions fall into five categories:
  - Arithmetic and logic — arithmetic and logical operations performed by the ALU
  - Change of flow — jumping to specific locations of the program unconditionally, or conditionally, by testing bits in the status register
  - Data transfer — moving data in/out of registers, into the data space, or into RAM
  - Bit and bit-test — inspecting data in registers (specifically for bit-level operations)
  - Control — Special microcontroller instructions

1.5.2 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 a large array of locations that can be read from, or written to, using an address. Peripherals can be controlled by reading from, and writing to, their registers which have a unique address in the data space. When peripherals are accessed in this manner we refer to them as being memory mapped. Different devices, peripherals, and memory can be included in a memory map (and sometimes a device can be accessed at multiple different addresses).

2 Digital Representations and Operations

2.1 Bits, Bytes, and Nibbles

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. These states can represent a variety of concepts:

- true, false (Boolean states)
- high, low (voltage states)
- on, off (switch states)
- set, reset (memory states)
- 1, 0 (binary states)

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.

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

2.2 Number Representations

2.2.1 Binary

The **binary system** is a base-2 system that uses a sequence of bits to represent a number. Bits are written right-to-left from **least significant** to **most significant** bit. The left-most 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.
- 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. As with the familiar decimal system, left-padded zeros do not change the value of a number, but are included here for formatting purposes.

2.2.2 Octal

The **octal system** is a base-8 system. It is most notably used as a shorthand for representing file permissions on UNIX systems, where three bits are used to represent read, write, execute permissions for the owner of a file, the groups the owner is part of, and other users.

$0_8 = 000_2$	$4_8 = 100_2$
$1_8 = 001_2$	$5_8 = 101_2$
$2_8 = 010_2$	$6_8 = 110_2$
$3_8 = 011_2$	$7_8 = 111_2$

As each octal digit maps to three bits, it is not very convenient for systems with byte-sized data. Despite this, it is still available in many programming languages for historical reasons.

2.2.3 Hexadecimal

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

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

2.2.4 Numeric Literals

When a fixed value is declared directly in a program, it is referred to as a **literal**. Here we must use prefixes to denote the base of the number:

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

For example, `0x80 = 0o200 = 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 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:

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

For 8-bit sequences, this range is: -127 to 127. However, this presents several issues:

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 $1 \to 0$). While the range of representable values are still the same:

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

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. It can be shown that the two's complement is calculated by adding 1 to the one's complement. The range of representable values is then:

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

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` a, $\sim a$, $\overline{a}$
- Conjunction: a `AND` b, a `&` b, $a \cdot b$, $a \wedge b$
- Disjunction: a `OR` b, a `|` b, $a + b$, $a \vee b$
- Exclusive disjunction: a `XOR` b, a `^` b, $a \oplus b$

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 used to **invert** a bit.

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**.

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**.

a	b	a `OR` b
0	0	0
0	1	1
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**.

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.

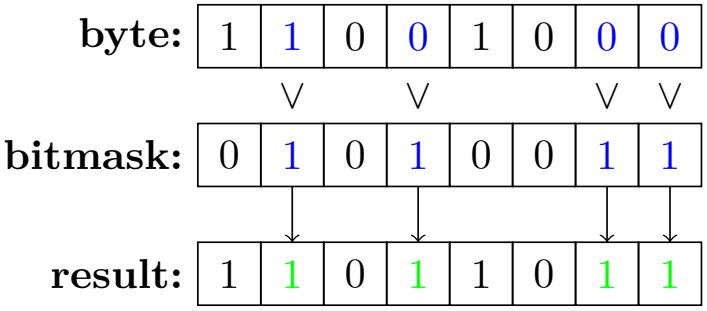


Figure 2: Setting bits using the logical or.

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.

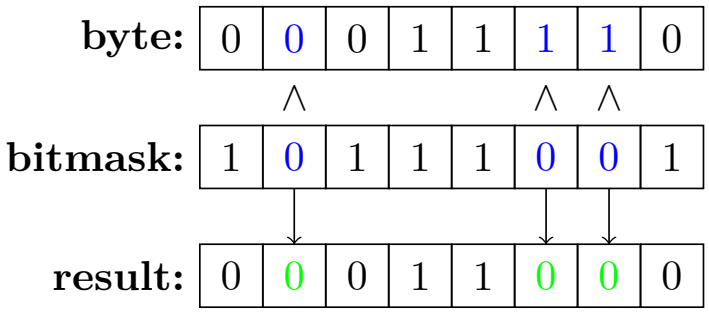


Figure 3: Clearing bits using the logical and.

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.

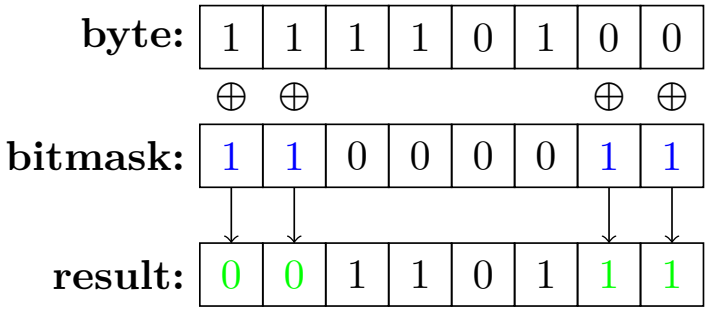


Figure 4: Toggling bits using the logical exclusive or.

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 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.

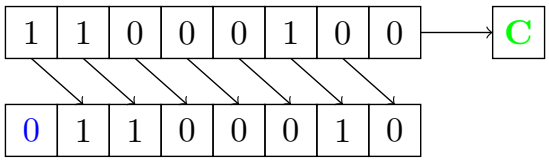


Figure 5: Right shift using `lsr` in AVR Assembly.

Similarly,

$$a \ll s$$

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

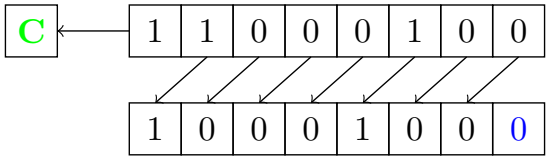


Figure 6: Left shift using `lsl` in AVR Assembly.

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

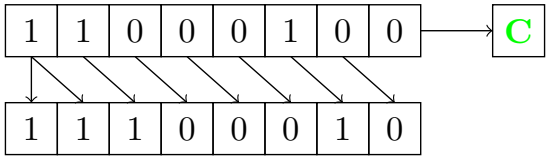


Figure 7: Arithmetic right shift using `asr` in AVR Assembly.

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

Rotations are used to shift bits with a carry from the previous instruction. To understand why, calculate the decimal value of the resulting byte after a shift.

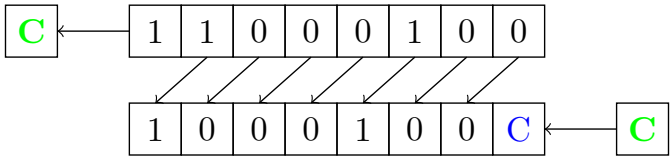


Figure 8: Rotate left using `rol` in AVR Assembly.

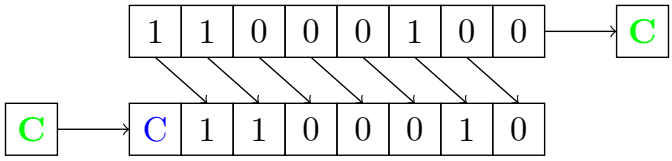


Figure 9: Rotate right using `ror` in AVR Assembly.

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 multibyte 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.

```
; Accumulator
ldi r16, 0
```

```
; First number
ldi r17, 29
add r16, r17 ; R16 <- R16 + R17 = 0 + 29 = 29

; Second number
ldi r17, 118
add r16, r17 ; R16 <- R16 + R17 = 29 + 118 = 147
```

Below is a graphical illustration of the above code.

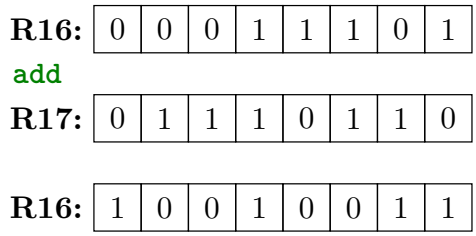


Figure 10: Overflow addition using `add` in AVR Assembly.

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. 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.

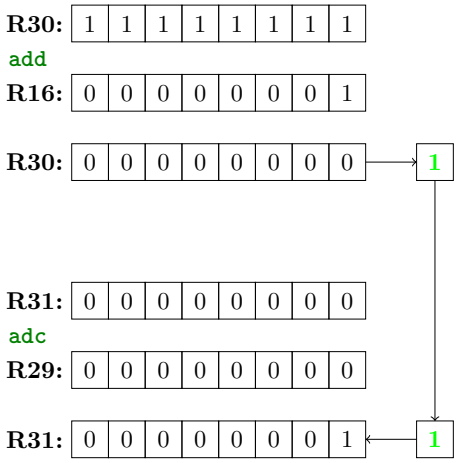


Figure 11: Overflow addition using `adc` in AVR Assembly.

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 multiplicand 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.

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

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 = 559
```

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.

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.

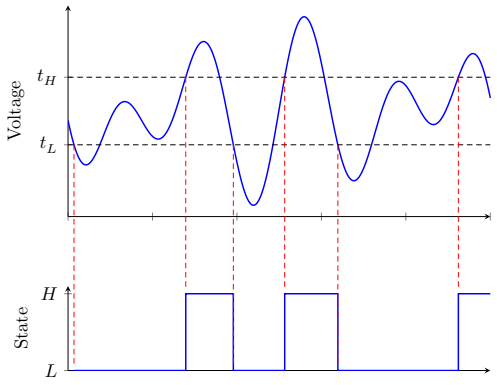


Figure 12: Example of hysteresis.

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 preferable to drive a digital input to an unambiguous voltage level.

3.2 Electrical Quantities

3.2.1 Voltage

Voltage v is 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, 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 is 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

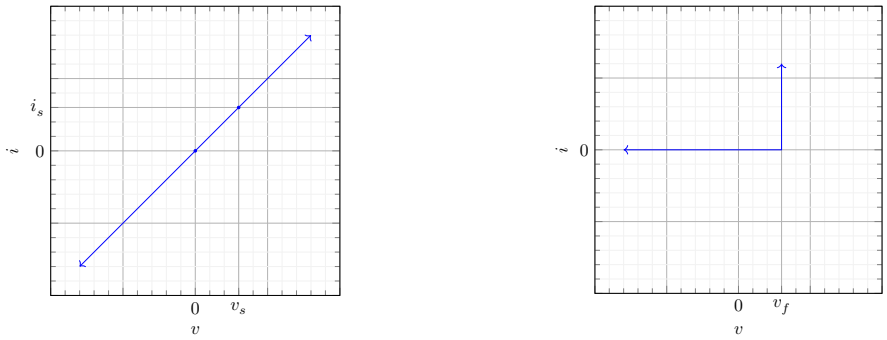
$$p = vi.$$

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), as 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 13: 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 Common Electrical Components

3.3.1 Resistors

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

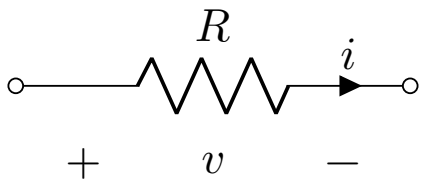


Figure 14: Resistor circuit symbol.

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**[2] current
- In the **closed** state, the switch **will conduct** current

Switches can take a variety of forms:

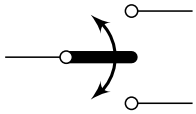
- **Poles** — the number of circuits the switch can control.
- **Throw** — the number of output connections each pole can connect its input to.

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

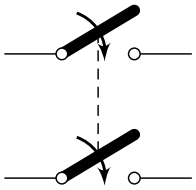
- Momentary or toggle action
- Different form factors, e.g., push button, slide, toggle, etc.

Switches are typically for user input.

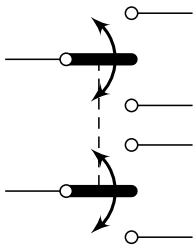
(a) Single pole single throw switch.



(b) Single pole double throw switch.



(c) Double pole single throw switch.



(d) Double pole double throw switch.

Figure 15: 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**.

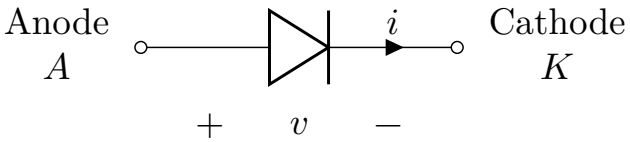


Figure 16: Diode symbol.

Diodes are a non-Ohmic device:

- 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 17: 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 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 a logical high, or logical 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).

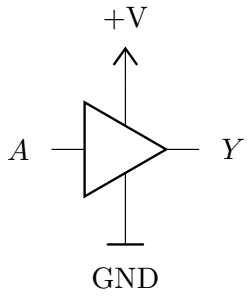


Figure 18: Push-pull output.

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 2: 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.

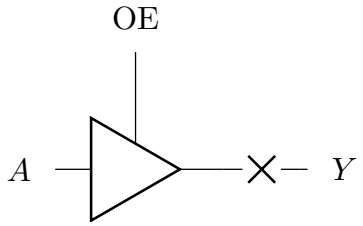


Figure 19: High-impedance output.

- 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: 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 devices share 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**.

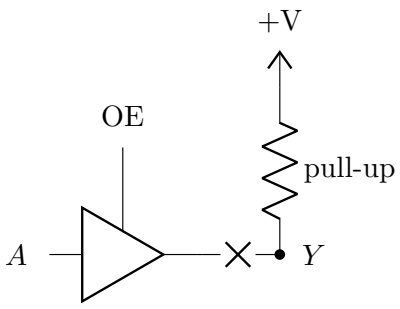


Figure 20: Pull-up resistor.

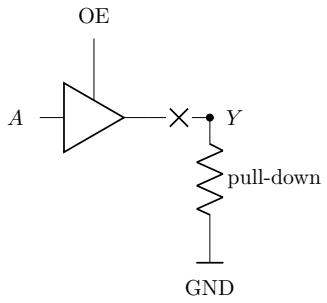


Figure 21: Pull-down resistor.

- 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.

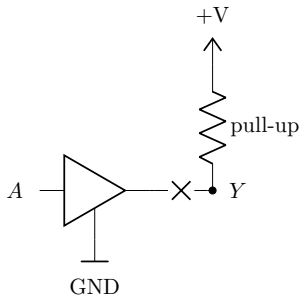


Figure 22: Open-drain output.

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 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).

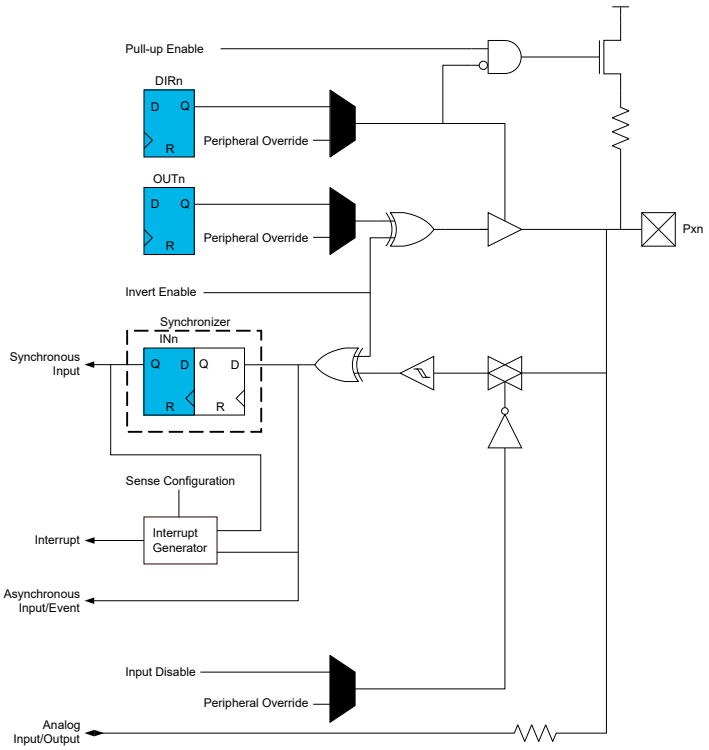


Figure 23: ATtiny1626 PORT block diagram.

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 resistor is enabled through software.
- The physical voltage on the pin can be routed to an analogue to digital converter (ADC) for measurement.
- Other peripheral functions can override port pin configurations and the output state.

3.5.1 Configuring an Output

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.

For example, assume an active HIGH device is connected to pin 5 on port B. To configure the pin as an output and set the output state to HIGH:

```
; 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

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.

For example, to read the state of pin 5 on port A:

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

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

; Isolate desired pin
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 3.1 (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 peripheral's configuration registers.

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

3.6 Interfacing to Simple I/O

3.6.1 Interfacing to 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 amount of current necessary for this is dependent on the LED and the capability of the driver device (the microcontroller). A typical indicator LED requires a current that ranges from 1 m A to 2 m A.
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.

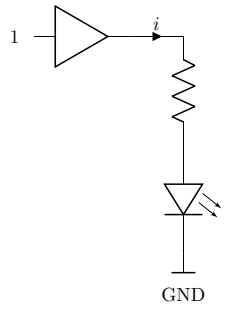


Figure 24: Active high configuration.

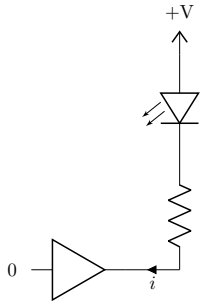


Figure 25: Active low configuration.

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.

3.6.2 Interfacing to Switches

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).

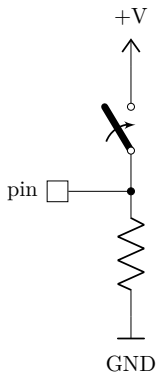


Figure 26: Active high configuration.

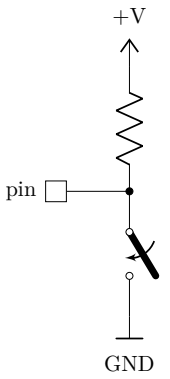


Figure 27: Active low configuration.

- 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 the switch.

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.

3.6.3 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.

3.7 Programming a Microcontroller

There are two main methods of programming a microcontroller. The first is to use Assembly language, which is the most direct way to interact with the instruction set of the microcontroller. The second is to use a higher-level language, such as C, which is more abstracted from the instruction set of the microcontroller. The choice of language depends on the complexity and performance requirements of the application.

Part II
AVR Assembly Programming

4 Introduction to AVR Assembly

Definition 4.1 (Word)**.** A word refers to a value that is two bytes in size (16-bit).

Definition 4.2 (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`.

```
ldi r16, 17 ; Load the value 17 into r16
```

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

Instructions in Assembly language are **mnemonics** that represent a specific operation that the microcontroller can perform. These instructions can take a number of **operands** that specify the **parameters** of the operation. The particular syntax for an instruction and its operands is dependent on the instruction, but generally, take the following form:

```
mnemonic [operand1, operand2]
```

For many instructions, the first operand often corresponds to the **destination** of an operation.

4.1 Unconditional Flow Control

Most instructions on the AVR Core increment the PC by 1 or 2 (depending on how many words the opcode occupies) 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 `jmp` (jump) instruction is used to simply jump to a different location in the program. By design, this instruction is capable of jumping to addresses up to 4 MB in program memory, although the ATtiny1626 does not have this much memory.

4.1.1 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).

```
jmp new_location ; Jump to the label new_location.
ldi r16, 1       ; This instruction is skipped

new_location: ; Label
    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. 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.

4.1.2 Absolute and Relative Addresses

`jmp` is a 32-bit instruction, that 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 a 16-bit PC can address on the ATtiny1626 (64 KB). As we will only ever need to jump within 64 KB of memory, it is inefficient to use the `jmp` instruction as it costs 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 is therefore more suitable as it only requires 2 CPU cycles.
Note the assembler throws an error if the specified address is not within the range of the PC.

4.2 Conditional Flow Control

Branching instructions jump to another location in memory based on a condition, i.e., user input, internal state, or other external factors. These instructions alter the PC differently based on the value of register(s) or flags. On the AVR there are two main categories of branching instructions:

- Branch instructions
- Skip instructions

4.2.1 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 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 address in these opcodes, branch instructions cannot jump as far as the `jmp` and `rjmp` instructions.

4.2.2 Compare Instructions

Both the `cp` and `cpi` instructions are used to compare the values in one or two registers. When used, the ALU performs a subtraction operation on the two operands and updates the SREG. Note that the result is not stored or used in any way as it is not relevant to the operation.

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

```
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.

4.2.3 Skip Instructions

The skip instructions are less flexible than branch instructions, but can sometimes be more suitable as they require less space and fewer cycles. Skip instructions skip the next instruction if a 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
```

Compare this with the branch instruction:

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

new_location: ; PC is now here
```

Note that the number of cycles for a skip instruction depends 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
```

Comparing this with the branch instruction:

```
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: ; 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
```

Comparing this with the branch instruction:

```
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:
```

4.3 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
```

5 Working with AVR Assembly

5.1 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.[3]
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)
- 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:

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

As the default prescaler is 6, the default effective clock speed is 3.333 MHz. The effective clock speed can therefore range between:

- Effective maximum clock frequency: 20 MHz (20 MHz clock & prescaler 1)[4]
- 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.

[3] 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.

[4] As the QUTy is supplied with 3.3 V, it is not safe to go above 10 MHz.

```
delay_1:
    ldi r16, x ; 1 CPU cycle
    ldi r17, 1 ; 1 CPU cycle ; Incrementor

    loop:
        add r16, r17 ; 1 CPU cycle
        brcc loop    ; 2 CPU cycles (1 CPU cycle when condition is false)
```

The register `r16` has the following relationship:

$$x = (2^8 - 1) - n \iff n = (2^8 - 1) - x$$

with

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

for a maximum delay of 230.7 µs $((3 \times (2^8 - 1) + 4)\, T)$[5]. To create larger delays, we can use multiple registers:

```
delay_2:
    ldi r24, x ; 1 CPU cycle
    ldi r25, y ; 1 CPU cycle

    loop:
        adiw r24, 1 ; 2 CPU cycles
        brcc loop   ; 2 CPU cycles (1 CPU cycle when condition is false)
```

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

$$(y : x) = (2^{16} - 1) - n \iff n = (2^{16} - 1) - (y : x)$$

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
```

[5] T is the period of one CPU cycle (using the default clock configuration): $T = \frac{1}{20\,\text{MHz}/6} = 300\,\text{ns}$.

```
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:

$$(z : y : x) = (2^{24} - 1) - n \iff n = (2^{24} - 1) - (z : y : x)$$

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 consumes 1 CPU cycle and does nothing. In addition to this, we can also utilise nested loops, although the timing is more complex to determine.

5.2 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 is used to simplify the design of the AVR Core and reduce chip complexity.
In the AVR architecture, programs are located in a separate address space, (although still accessible through the data space). This is in contrast to modern CPU architectures.

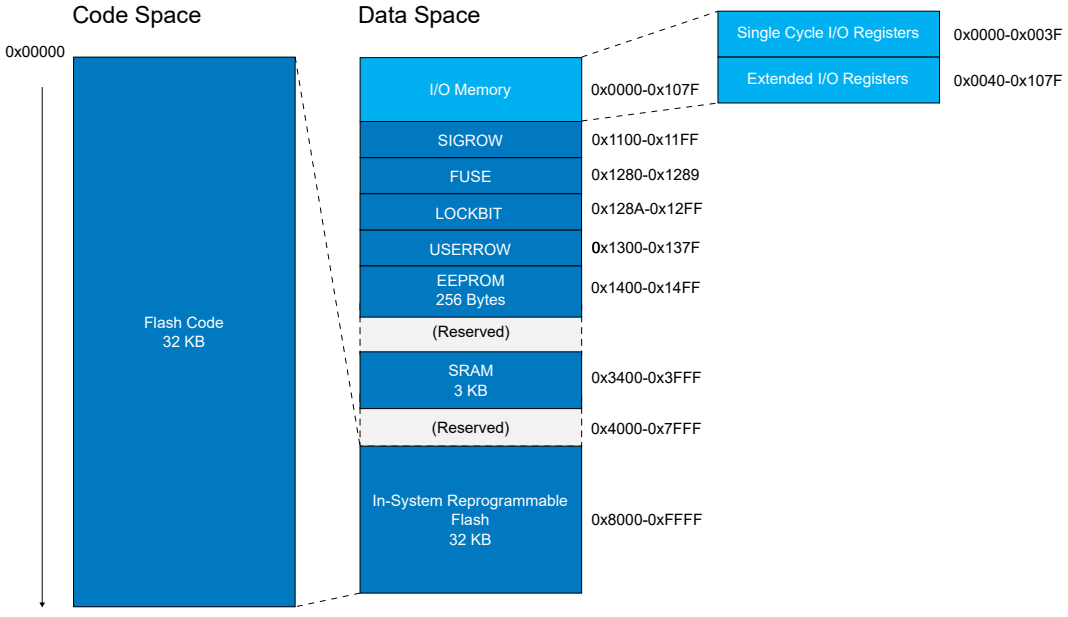


Figure 28: ATtiny1626 memory map.

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 SRAM — starting at `0x3800`)
- `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 their names suggest, these instructions only require a single CPU cycle and hence several addresses such as VPORT{A, B, C} (virtual ports) are mapped to this location, for fast access.

5.2.1 Load/Store Indirect

While the `lds`/`sts` instructions can be used to access addresses of bytes, they are generally not suitable for accessing data structures such as arrays. Instead, we can use the `ld`/`st` instructions to take advantage of their 16-bit pointer registers, which support some pointer arithmetic.

- `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:

```
ldi XL, lo8(RAMSTART) ; Store address of RAM in X
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 (and hence at RAMSTART) 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
```

5.2.2 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`.

5.3 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 (SP $\leftarrow$ SP $-$ 1)
- `pop` increments SP (SP $\leftarrow$ 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:

```
push ZL ; Temporarily store Z on the stack
push ZH
; Z may be used for another purpose
pop ZH ; Restore Z from the stack in reverse order
pop ZL
```

5.4 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
```

5.4.1 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
    ; Code that possibly modifies r16
    pop r16 ; Restore r16 from the stack
    ret

main_loop:
    ldi r16, 10
    rcall procedure ; Call procedure
    push r16 ; r16 should still be 10
```

5.4.2 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:
    push r0 ; Save r0
    in r0, CPU_SREG ; Save SREG
    push r0

    ; Calculate average
    add r16, r17
    ror r16

    pop r0 ; Restore SREG
    out CPU_SREG, r0
    pop r0 ; Restore r0
    ret

main_loop:
    ; Arguments
    ldi r16, 100
    ldi r17, 200
    rcall average
```

Using the stack:

```
rjmp main_loop

; Calculate the average of two numbers
; Inputs:
;     top two values on stack
; Outputs:
;     r16: average
average:
    push ZL ; Save Z
    push ZH
    in ZL, CPU_SREG ; Save SREG
    push ZL
    push r17 ; Save r17
```

```
    in ZL, CPU_SPL ; Get SP location
    in ZH, CPU_SPH

    ; Get numbers number
    ldd r16, Z+7
    ldd r17, Z+6

    ; Calculate average
    add r16, r17
    ror r16

    pop r17 ; Restore r17
    pop ZL ; Restore SREG
    out CPU_SREG, ZL
    pop ZH ; Restore Z
    pop ZL
    ret

main_loop:
    ; Arguments
    ldi r16, 100
    push r16
    ldi r16, 200
    push r16
    rcall average

    ; Remove arguments from the stack
    pop r0
    pop r0
```

Note that it is preferable to return values using registers.

Part III
C Programming

6 Introduction to 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 targeting 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.

6.1 Basic Structure

6.1.1 The 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.

6.1.2 Statements and Comments

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;
    }
    // x is now 7
    // y is no longer in scope
    return 0;
}
```

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.

```c
/*
    This is a multi-line comment.
    It can span multiple lines.
*/
```

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

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

All comments in C are ignored by the compiler.

6.2 Variables, Literals, and Types

6.2.1 Declaring and Initialising Variables

Variables are used to temporarily store values in memory. Variables have a **type** and a **name** and must be declared before use. To declare a variable in C, we must specify the type and name of that variable.

```c
int x;
```

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

```c
x = 4;
```

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.

```c
int x = 4;
```

Note that using **uninitialised variables** results in **unspecified behaviour** in C, meaning that the value of such variables is unpredictable. If we want to assign values to multiple variables of the same type, we can use the comma (,) operator.

```c
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.

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

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

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

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

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

6.2.2 Types

While AVR assembly uses 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`.

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. 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.*

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`
  - In a global context — indicates that the variable is only accessible within the file.
  - In a local context — indicates that the variable maintains its value between function invocations.

- `volatile` — indicates that the variable can be modified or accessed by other programs or hardware.

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.

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>

int8_t i8;
int16_t i16;
int32_t i32;
int64_t i64;

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

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 small values. These types are very useful on systems with a floating point unit (FPU) or equivalent. In most computer systems, floating point types are represented as 32-bit IEEE 754 single precision floating point numbers. The `float` type is a 32-bit floating point number and the `double` type is a 64-bit floating point number.
A single precision floating point number has a 1-bit sign, 8-bit exponent, and 23-bit mantissa. As such, the range of a single precision floating point number is $-2^{127} \dots 2^{127}$. The value of a floating point number f is defined as

$$f = (-1)^s \left(1 + 2^{-23} m\right) 2^{e-127}$$

where s is the sign bit, m is the mantissa, and e is the exponent. Note that values are not equally spaced and there are several special values that can be represented by floating point numbers.

- $e = 255 \implies 2^{128}$:
  - $m = 0$ (all 0s): `INFINITY` if $s = 0$, `-INFINITY` if $s = 1$

  - m is not all 0s: `NAN`
- $e = 0 \implies 2^{-126}$ (de-normalised):
  - $m = 0$ (all 0s): `0.0` if $s = 0$, `-0.0` if $s = 1$
  - m is not all 0s: Subnormal numbers

The flexibility of floating point numbers means that arithmetic operations are expensive if not performed on a Floating Point Unit (FPU). As the ATtiny1626 does not have an FPU, floating point operations must be handled using ALU instructions, which can be extremely slow, especially when compared to integer operations. In addition, floating point operations require the `avr-libc` floating point library to be linked, which increases the size of the program.
Fixed point mathematics is a technique for performing arithmetic operations on integers that are scaled by a power of two. This allows for integer arithmetic to be used instead of floating point arithmetic, which can be significantly faster at the cost of precision. For common operations such as sine and cosine, consider using lookup tables.

6.3 Literals

6.3.1 Integer Prefixes

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

- **Binary** (base 2) — `0b`
- **Octal** (base 8) — `0` (note that C does not use the `0o` prefix)
- **Decimal** (base 10) — no prefix
- **Hexadecimal** (base 16) — `0x`

6.3.2 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, or when an operation may lead to truncation due to an overflow.

```
#include <stdio.h>

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

6.3.3 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 Flow Control

6.4.1 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
}
```

6.4.2 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
}
```

A `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
}
```

6.4.3 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).

6.4.4 Break and Continue Statements

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

```
for (int i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break; // Terminate loop early
```

```
    }
    printf("%d\n", i);
}
```

```
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.

```
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);
    }
}
```

6.5 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.

6.5.1 Operation Precedence

Operation	Operator Symbol	Associativity
Postfix	`++`, `--`	
Function call	`()`	
Array sub-scripting	`[]`	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

6.5.2 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).

6.5.3 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, `? :`.

6.5.4 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.

6.5.5 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");
}
```

6.5.6 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");
}
```

6.5.7 Increment and Decrement Operators

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.

7 Compiling and Linking C Programs

7.1 Preprocessing

The preprocessor processes C source 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. The file generated by the preprocessor is called a *translation unit* (or a *compilation unit*). Two basic preprocessor directives are `#include` and `#define`.

7.1.1 Include Directives

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.

7.1.2 Define Directives

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, and thus must be defined carefully to avoid invalid code from being generated.

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

7.2 Compilation

After it has been preprocessed, a translation unit can be translated into machine code by a **compiler**, similar to how assembly code is translated into machine code by an **assembler**. Note that compilers may also emit assembly code as an intermediate step, which is then passed to an assembler to produce machine code.

7.2.1 Compilers

A compiler is a program that translates a translation unit written in a high-level language such as C. Compilers can be configured to produce executable code in various ways. Compiling without

optimisation will produce code that closely resembles source code, making the program easier to debug. This often also results in faster compilation times. When releasing a program, a compiler can also be configured to optimise code for minimum code size or maximum speed (or a combination of both). Compilers offer some advantages over assembly code:

- **Portability** — high-level languages are more portable than assembly code, as they are not tied to a specific instruction set architecture. This means that the same source code can be used on another platform, granted that a compiler for that platform is available.
- **Efficiency** — compilers allow us to design efficient program through the use of compiler optimisations, which can be difficult to achieve manually in assembly code.

7.2.2 Assemblers

Program code that is run directly on a CPU is not designed to be written by humans. The assembler prioritises performance and size efficiency, and accounts for simplified chip design. An assembler allows us to write programs in plain text without needing to memorise opcodes or manually keep track of memory locations. Assemblers prioritise performance and size efficiency, and account for simplified chip design in their operation. Modern assemblers also provide features such as macros that allow us to write reusable code. While the scope of an assembler is limited, the programmer can be confident that the code produced is efficient and optimal, as instructions are directly translated into machine code. Some advantages of assemblers are described below:

- **Efficiency** — assemblers allow for precise control over the hardware, which may not be available in a high-level language. In such cases, inline assembly can be used to write assembly code within a high-level language.
- **Precision** — precise timing of code is possible due to the ability to predict how the assembler generates code. Compilers may introduce additional instructions that can make it difficult to predict the exact timing of code.

7.3 Object Files

An object file is the output from the compilation or assembler phase. Object files mostly contain machine code and information about the symbols defined in the source code. Large modularised programs which split source code into multiple files can be compiled into object files with *unresolved* external references. These references are resolved during the linking stage to produce a single executable file.

7.4 Linking

During the linking stage, the **linker** combines multiple object files and links them together to produce a single executable file. The linker resolves all external references and updates addresses where required. This step is necessary when source code is split into multiple files, and is extremely fast as only addresses need to be updated. The linker can also perform some optimisations such as dead code elimination, which removes unused code. In assembly, the `.global` directive can be used to make labels available to the linker.

```
// function.S
.global function

function:
    ret

// main.S
rcall function
```

In this example, both files can be compiled into object files even though the `function` label is not defined in `main.S`. The linker will resolve the reference to `function` and produce a single executable file. In C, top-level symbols are public by default, but can be made private to the current translation unit by using the `static` keyword.

```
// main.c
static int a = 0;

// file1.c
a = 1; // Error: a is not visible
```

To make a symbol visible to other translation units, the `extern` keyword can be used.

```
// main.c
extern int a;
printf("%d\n", a); // Prints 5

// file1.c
int a = 5;
```

Any non-static symbols are implicitly global, and can be accessed from any translation unit.

7.5 Debugging

While most microcontrollers are equipped with debugging tools, we are often presented with no debugging tools at all. Therefore, it is important to develop strategies to be able to systematically debug embedded programs with access to basic I/O. Some simple methods include toggling pins on the microcontroller to indicate the state of the program and sending formatted strings through the serial port to a terminal. To route stdin and stdout to/from any serial communications interface that can read and write characters (e.g., UART, SPI, I^2C, etc.), we can use the `stdio.h` library:

1. Declare function prototypes for the read/write functions (function names are arbitrary):

   ```
   static int stdio_putchar(char c, FILE *stream);
   static int stdio_getchar(FILE *stream);
   ```

2. Declare a stream to be used for stdin/stdout, using the `FDEV_SETUP_STREAM` macro:

```
static FILE stdio = FDEV_SETUP_STREAM(stdio_putchar, stdio_getchar,
 ↪  _FDEV_SETUP_RW);
```

3. Implement the prototyped functions that read from the serial interface (i.e., via UART):

```
static int stdio_putchar(char c, FILE *stream)
{
    uart_putc(c);
    return c;
}

static int stdio_getchar(FILE *stream)
{
    return uart_getc();
}

void stdio_init(void)
{
    // Assumes serial interface is initialised elsewhere
    stdout = &stdio;
    stdin = &stdio;
}
```

Here we may use the following blocking functions to read and write characters to the UART interface:

```
uint8_t uart_getc(void) {
    while (!(USART0.STATUS & USART_RXCIF_bm)); // Wait for data

    return USART0.RXDATAL;
}

void uart_putc(uint8_t c) {
    while (!(USART0.STATUS & USART_DREIF_bm)); // Wait for TX.DATA empty

    USART0.TXDATAL = c;
}
```

Assembly listings are also useful for debugging in extreme cases, as they allow us to see exactly what instructions are being executed. This can be achieved via the `avr-objdump` tool.

8 Advanced C Programming

8.1 Pointers

When a variable is declared, the compiler automatically allocates a block of memory to store that variable. We can access this block of memory directly using the identifier for that variable, or indirectly, using a **pointer**. Pointers are variables that store the memory address of another variable and are declared using the following syntax:

```
uint8_t *ptr; // Uninitialised pointer to uint8_t data
```

This code declares a variable `ptr` that "points to" a `uint8_t`. Internally, as this pointer is simply an address, it only occupies the size of a **memory address**, which is 16 bits on the ATtiny1626.

8.1.1 Referencing

We can **reference** another variable using the following syntax:

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

The ampersand (`&`) operator is used here to return the **address of** the variable `x`. Notice that the type of the pointer `ptr` must match the type of `x` to ensure that the pointer is correctly dereferenced. If we know the address of a location in advance, we can also declare a pointer with a specific address, ensuring that we cast this address to a pointer type:

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

8.1.2 Dereferencing

Once we have defined a pointer, we can indirectly access the value it references using the **unary dereference** operator (`*`). This is also known as **indirection**.

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

// Indirectly read and write from x
uint8_t y = *ptr;  // y = 5  (read from x)
*ptr = 10;         // x = 10 (write to x)
```

8.1.3 Using Qualifiers

Various qualifiers can be used to modify the type of a pointer, and these qualifiers can apply to both the pointer and the variable it references. When reading a pointer declaration with qualifiers, we read from right to left.

- **Non-constant Data** — We can reference non-constant data without any qualifiers, as shown in the examples above.

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

uint8_t *ptr = &a; // pointer to a uint8_t

*ptr = 300; // Can indirectly modify referenced variable
ptr = &b;   // Can reassign pointer to another address
```

We can enforce constancy on the referenced variable by applying the `const` qualifier to the data type in the pointer's declaration, even if the data itself is not constant.

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

const uint8_t *ptr = &a; // pointer to a constant uint8_t

*ptr = 300; // Error: Cannot indirectly modify referenced variable
ptr = &b;   // Can reassign pointer to another address
```

- **Constant Data** — We can reference constant data by using the `const` qualifier on the referenced data type.

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

const uint8_t *ptr = &a; // pointer to a constant uint8_t

*ptr = 300; // Error: Cannot indirectly modify referenced variable
ptr = &b;   // Can reassign pointer to another address
```

If we wish to modify the referenced variable, we can reference the variable without the `const` qualifier:

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

uint8_t *ptr = &a; // pointer to a uint8_t

*ptr = 300; // Can indirectly modify referenced variable
ptr = &b;   // Can reassign pointer to another address
```

- **Constant Pointers** — We can enforce a pointer to be constant by applying the `const` qualifier to the pointer type. This disallows the pointer from being reassigned another address.

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

uint8_t *const ptr = &a; // constant pointer to a uint8_t

*ptr = 300; // Can indirectly modify referenced variable
ptr = &b;   // Error: Cannot reassign constant pointer
```

- **Constant Pointers to Constant Data** — We can enforce both the pointer and the referenced data to be constant by applying the `const` qualifier to both the pointer and the referenced data type.

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

const uint8_t *const ptr = &a; // constant pointer to a constant uint8_t

*ptr = 300; // Error: Cannot indirectly modify referenced variable
ptr = &b;   // Error: Cannot reassign constant pointer
```

8.1.4 Pointers to Pointers

Pointers can also reference other pointers.

```
uint8_t a = 100;

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

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

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

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

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

These examples demonstrate that pointers are simply variables that store memory addresses. We may increase the level of indirection by including more asterisks in a pointer declaration, but this

is generally not very common, nor is it particularly useful in most cases. In the following example, we apply qualifiers to pointers referencing other pointers:

```
uint8_t a = 100;
uint8_t *ptr = &a;             // Points to `a`
const uint8_t **ptr1 = &ptr;   // 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
```

8.1.5 Pointer Arithmetic

A common application of pointer types is for iterating through data structures such as arrays. Here we often want to move a pointer to the next or previous element in a sequence. This can be achieved through arithmetic operators such as `++` and `--`. Arithmetic on pointers affects the value of the pointer, so that the pointer references a different memory location. When performing arithmetic on pointers, the size of an increment is automatically determined by the type of the referenced variable.

```
uint8_t a = 100;
uint8_t *ptr = &a; // Points to `a`

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

8.1.6 Void Pointers

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

```
void *ptr;
```

Void pointers have no type, and as such, cannot be dereferenced without first being cast to a specific type. On the other hand, a void pointer can be assigned to any other pointer type without a cast.

```
uint8_t a = 100;
void *ptr = &a; // Void pointer

*ptr = 200;            // Error: Cannot dereference void pointer
*(uint8_t *)ptr = 200; // Cast void pointer to uint8_t pointer before
 ↪  dereferencing

uint8_t *ptr2 = ptr;   // Pointer value is copied to `ptr2`
```

8.1.7 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
```

8.2 Array Types

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 initialised to 0
uint8_t c[] = {1, 2, 3};   // Array of 3 uint8_t initialised to 1, 2, 3
uint8_t d[5] = {1, 2, 3}; // Array of 5 uint8_t initialised 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.

8.2.1 Indexing

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

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

```
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.

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

8.2.2 Array Decay

Expressions with array types can be converted to expressions with pointers of the same type, allowing us to use pointer arithmetic with arrays. However, it is important to note that this results in a loss of information about the size of the array.

```
uint8_t a[10];
sizeof(a); // Returns 10

uint8_t *ptr = a; // Pointer to first element of a
sizeof(ptr);      // Returns 2 (size of the pointer)
```

In the declaration of `ptr`, the array *decays* into a pointer. This property is especially useful when accessing arrays through function parameters, as arrays themselves cannot be passed by value. Instead, we can pass a reference to the array by taking advantage of array decay, ensuring that we also pass a second parameter for the size of the array, as this information is lost.

```
void set_first_to_100(uint8_t *arr) {
    arr[0] = 100;
}

uint8_t a[10] = {0};
set_first_to_100(a); // a = {100, 0, 0, 0, 0, 0, 0, 0, 0, 0}

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

uint8_t b[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
print_array(b, 10); // Prints 0 1 2 3 4 5 6 7 8 9
```

Note that the parameter syntax `uint8_t *arr` is equivalent to `uint8_t arr[]`, however the former is generally preferred. The syntax `arr[i]` is equivalent to `*(arr + i)`. 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
```

8.2.3 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.

8.2.4 Copying Arrays

Arrays can be copied in one of two ways. The first approach uses 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 approach uses the `memcpy` function from the `string.h` library, which performs the same operation.

```
uint8_t a[10];
uint8_t b[10];
memcpy(b, a, sizeof(a) / sizeof(a[0]));
```

8.2.5 Multidimensional Arrays

Multidimensional arrays (or multiple subscript arrays) are used to hold multidimensional data.

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

To declare a multidimensional 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.

```
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
    }
}
```

8.3 Functions

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

```c
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:

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

8.3.1 Parameters

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

```c
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`
}
```

When a function does not take any arguments, we can use the `void` keyword to prevent parameters from being passed in accidentally.

```c
void func(void) {
}
```

8.3.2 Return Values

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

8.3.3 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.

```c
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);
}

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.

8.3.4 Passing by Reference

As functions only return one value, we can use pointers to pass multiple values back to the caller. These output values can be passed as additional parameters.

```
void swap(uint8_t *a, uint8_t *b) {
    uint8_t temp = *a;
    *a = *b;
    *b = temp;
}

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

    swap(&a, &b);
}
```

8.3.5 Call 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 reverted 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.

8.4 Scope

All variables and other identifiers in C are scoped. 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.

8.4.1 Global Scope

Variables declared outside 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.

8.4.2 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.

8.4.3 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.

8.4.4 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.

8.5 Advanced Type Techniques

8.5.1 Volatile Qualifiers

As seen in the previous section, we can use the `volatile` keyword to directly reference memory locations by their address. This is useful for accessing memory mapped IO.

```
volatile uint8_t *portb_outclr = 0x0426;
*portb_outclr = 0b00100000;
```

The `volatile` keyword is important here because this data is outside the control of the program. The compiler will therefore not optimise accesses to this variable. The `avr/io.h` header file includes macros and type definitions for accessing various registers on the AVR microcontroller with similar declarations.

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

8.5.2 Type Casting

While C is a statically typed language, it is possible to convert between types. Some type conversions are performed implicitly, such as when converting a `uint8_t` to a `uint16_t`. However, some implicit type conversions generate warnings usually because they result in a loss of information, or because

the conversion is not portable across platforms. Therefore, to explicitly convert a variable to a different type, we can use the unary type casting operator.

```
volatile uint8_t *portb_outclr = (volatile uint8_t *)0x0426;
```

Applying this type cast does not make this code more portable, but rather it tells the compiler that the programmer is aware of the conversion being made, and that it is intentional. Some common type casts are performed between:

- **Integer types**: Conversions between integer types (signed or unsigned) will expand or narrow the type, resulting in a truncation or zero extension.

  ```
  (uint8_t)-3 // 253
  ```

- **Floating point types**: Conversions from floating point types to integer types results in truncation of the fractional part.

  ```
  (int16_t)-3.45 // -3
  ```

- **Pointer types**: Conversions on pointer types change the pointer's type but do not affect the referenced data. As platforms may store data differently, these conversions are not portable.

  ```
  uint16_t a = 12345;
  uint8_t *b = (uint8_t *)&a; // likely 57, but may vary on different platforms
  ```

  Casting an integer to a pointer will cause the resulting value to be treated as an address.

  ```
  uint8_t *ptr = (uint8_t *)0x1234;
  ```

  Likewise, casting a pointer to an integer will cause the pointer to be treated as an integer, effectively giving us the value of the pointer.

  ```
  uint8_t *ptr = (uint8_t *)0x1234;
  uint16_t addr = (uint16_t)ptr; // addr = 0x1234
  ```

  These conversions are not portable, but are often necessary when accessing memory mapped IO.

Type casting can also be used to add/remove qualifiers such as `const` and `volatile`, although this can lead to undefined behaviour if not used correctly. For example, some platforms store `const` variables in read-only memory, and attempting to modify these variables is undefined behaviour.

```
const uint8_t a = 10;
const uint8_t *b = &a;

*((uint8_t *)b) = 20; // May lead to undefined behaviour
```

Casting can also be used to avoid truncation errors when performing arithmetic.

```
uint16_t a = 25000;
uint16_t b = 10000;

uint32_t c = a * b;           // c = 45696 (incorrect)
uint32_t d = (uint32_t)a * b; // d = 250000000
```

9 Objects

9.1 Structures

Structures are used to group related data together.

```
struct Point {
    uint8_t x;
    uint8_t y;
};

struct Point p;
```

The members of a structure can be accessed using the dot operator.

```
p.x = 30;
p.y = 40;
```

Struct members can also be initialised using braces as with arrays.

```
struct Point p = { 30, 40 };
```

Unlike arrays, structures need not be accessed via pointers and can be passed between functions, and copied normally.

```
void func(struct Point p) {
    p.x = 50;
}
```

```
struct Point p = { 30, 40 };
func(p);

struct Point q = p; // q.x = 50, q.y = 40
```

Due to this, structs can contain arrays which can be passed and copied by placing them in structs. Along with this, functions can also return structs.

```
struct Point func() {
    struct Point p = { 30, 40 };
    return p;
}

struct Point p = func(); // p.x = 30, p.y = 40
```

9.1.1 Memory Layout

Struct members are stored in memory in the order they are declared. If the platform has alignment requirements, the compiler will insert padding to ensure that the next member is aligned correctly. This is done to ensure that the compiler can access the members of the struct efficiently.

9.1.2 Anonymous Structures

Structures can be declared without a name if they are only used once.

```
struct {
    uint8_t x;
    uint8_t y;
} p;
```

The type of this variable is `unnamed`.

9.1.3 Structures Inside Structures

Structures can contain other structures.

```
struct Point {
    uint8_t x;
    uint8_t y;
};

struct Rectangle {
    struct Point p1;
    struct Point p2;
};
```

```
struct Rectangle r = { { 10, 20 }, { 30, 40 } };
```

9.1.4 Structures and Pointers

Structures and members of structs can be addressed normally with the address-of operator.

```
struct Point p = { 30, 40 };
struct Point *ptr = &p;

ptr->x = 50; // Equivalent to (*ptr).x = 50
```

When accessing members of structs through pointers, the arrow operator (`->`) can be used. Structures can also contain pointers.

9.1.5 Typedef

Typedefs can be used to give a type an alias so that the variables type is determined by the typedef instead of the actual type. If we want to use a structure multiple times, we can use a typedef to give it a (new) name.

```
typedef struct PointStruct {
    uint8_t x;
    uint8_t y;
} Point;

Point p = { 30, 40 }; // Point is an alias to struct PointStruct
```

The type of this variable is `Point`. The struct also need not be defined inside of the typedef.

```
struct PointStruct {
    uint8_t x;
    uint8_t y;
};

typedef struct PointStruct Point;

Point p = { 30, 40 };
```

This can be useful when the struct is defined in a header file and the typedef is defined in a source file. In both cases, it is still possible to use the struct name to declare variables.

```
struct PointStruct p;
```

If the struct name is omitted, the type of the struct is `unnamed`.

```c
typedef struct {
    uint8_t x;
    uint8_t y;
} Point;

Point p = { 30, 40 }; // Point is an alias to an unnamed struct
```

In this case, the struct name cannot be used to declare variables as it is anonymous. Typedefs can also be used with qualifiers to reduce unnecessary code.

9.2 Unions

Unions have similar syntax to structures, but all the members of a union share the same overlapping memory. While structs have capacity to store multiple members, unions only have the capacity to store their largest member.

```c
union Character {
    char character;
    uint8_t integer;
};

union Character c = { 'A' };

printf("%c\n", c.character); // Prints 'A'
printf("%u\n", c.integer); // Prints 65
```

This allows us to interpret the same memory location as multiple types without needing to perform a cast. When used with structs (or other aggregates), the order of members in those structs is also maintained.

```c
struct a {
    uint8_t i;
    float f;
};

struct b {
    uint8_t i;
    char c[4];
};

union u {
    struct a a;
    struct b b;
};
```

```
union u u;

u.a.i = 10;
u.a.f = 3.14;

// u.a.i = 10; u.a.f = 3.14

u.b.c[0] = 'A';
u.b.c[1] = 'B';
u.b.c[2] = 'C';
u.b.c[3] = 'D';

// u.b.i = 10; u.b.c = { 'A', 'B', 'C', 'D' }
```

9.3 Bitfields

Bitfields can be used within structures or unions to specify types of specific **bit** sizes.

```
struct {
    uint8_t x : 4;
    uint8_t y : 4;
} bits;

bits.x = 13;
bits.y = 7;

printf("%u\n", bits.x); // Prints 13
printf("%u\n", bits.y); // Prints 7
printf("%lu\n", sizeof(bits)); // Prints 1 (8 bits)
```

In this example, the members `x` and `y` each occupy 4 bits. Note that the base type of each member must be able to store the specified number of bits.

9.3.1 Properties of Bitfields

The address of a bitfield cannot be taken.

```
struct {
    uint8_t x : 4;
    uint8_t y : 4;
} bits;

uint8_t *ptr = &bits.x; // Error
```

A bitfield cannot be an array.

```
struct {
    uint8_t x : 4;
    uint8_t y[4] : 4;
} bits; // Error
```

The name of a bitfield can be omitted to introduce padding.

```
struct {
    uint8_t x : 4;
    uint8_t : 4;
} bits;
```

A zero-width bitfield can be used to align the next member to the next word boundary.

```
struct {
    uint8_t x : 4;
    uint8_t : 0;
    uint8_t y;
} bits;

printf("%lu\n", sizeof(bits)); // Prints 2
```

9.4 Strings

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

```
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.

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

When using the `printf` function, the null terminator is required to indicate the end of the string. Strings can therefore be indexed and passed to functions like arrays.

10 Interrupts

An interrupt is a signal sent to the processor to indicate that it should *interrupt* the current code that is being executed to execute a function called an **interrupt service routine** (or *interrupt handler*). Rather than polling for individual events (such as button presses), interrupts allow the processor to be notified when an event occurs.

10.1 Interrupts and the AVR

On the ATtiny1626, interrupts work as follows:

1. An interrupt-worthy event occurs.
2. The appropriate interrupt flag (`INTFLAGS`) in the peripheral is set.
3. If the corresponding interrupt is enabled (`INTCTRL` field of the peripheral), the interrupt is triggered, and we proceed to the following step.
4. If the global interrupt flag (`SREG.I`) is set, the interrupt can be executed, and we proceed to the following step.
5. The PC is pushed onto the stack and jumps to the interrupt vector (the address of the interrupt handler). See page 63–64 on the datasheet.

10.1.1 Interrupt Vectors

The **interrupt vector** is a table of addresses that the processor jumps to when an interrupt is triggered. These addresses are usually stored at the beginning of program memory.

10.1.2 Interrupt Service Routine

The code that handles an interrupt is called an **interrupt service routine** (ISR) (or *interrupt handler*). An ISR is simply a function that is executed as a result of an interrupt. When configuring an interrupt, we must temporarily disable interrupts globally to prevent the interrupt from being triggered while we are configuring it.

```
cli(); // Disable interrupts globally
// Configure interrupts
sei(); // Enable interrupts globally
```

It is also important to restore the state of the CPU or registers before the ISR returns. This is because another interrupt can be triggered while the ISR is executing. To tackle this, we can use the `ISR` macro from the `avr/interrupt.h` header file which will automatically save and restore the state of the CPU and registers.

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

ISR(TCB0_INT_vect) {
```

```
    // Interrupt service routine for TCB0
}
```

This header file also sets aside program memory for the interrupt vector table.

10.1.3 Interrupt Flags

Each peripheral has an interrupt flag field (`INTFLAGS`) that is set when the conditions for that interrupt occur (even if interrupts are disabled). The exact format of this field depends on the type of interrupt, but in general a bit is set for the type of interrupt. See the datasheet for more information. As some peripherals have 1 interrupt vector with multiple interrupt sources, the interrupt flag fields can be used to determine the exact source of the interrupt. The interrupt flag field is *usually* cleared by writing a 1 to the corresponding bit.

10.1.4 Peripheral Interrupts

Peripherals differ in what causes interrupts to be raised and many have multiple interrupt sources.

10.1.5 Port Interrupts

To configure `BUTTON0` as an interrupt source, we must enable the interrupt in the `PORTA.PIN4CTRL` peripheral.

```
// Interrupt service routine for PORTA
ISR(PORTA_PORT_vect) {
    // Check if the interrupt was caused by PORTA.PIN4
    if (PORTA.INTFLAGS & PIN4_bm) {
        // Interrupt service routine for PORTA.PIN4

        // Clear interrupt flag
        VPORTA.INTFLAGS = PIN4_bm;
    }
}

cli();
// Enable pull-up resistor and interrupt on falling edge
PORTA.PIN4CTRL = PORT_PULLUPEN_bm | PORT_ISC_FALLING_gc;
sei();
```

10.1.6 Interrupts and Synchronisation

ISR's may interact with state used by other code running at the same time, which can cause problems with synchronisation, similar to those faced in multithreaded programs. To avoid this, we should make use of the `volatile` keyword so that the compiler does not make assumptions about variable states. The `cli` and `sei` functions can also be used to disable interrupts and create a memory barrier which prevents instructions from being reordered by the compiler.

11 Hardware Peripherals

Microcontrollers typically include a variety of hardware peripherals that remove the burden of having to write software for common functionality such as timers, serial communication, and analogue to digital conversion. They can provide very precise timing and very fast (nanosecond) response times. Hardware peripherals can run independent of the CPU (in parallel) so that:

- peripherals can perform tasks without software intervention
- peripherals are not subject to timing constraints (execution time of instructions, CPU clock speed)
- the CPU can be used to perform other computations while the peripheral is busy

All control of, and communication with peripherals is done through **peripheral registers** which the CPU can access via the memory map. Peripherals also typically have direct access to hardware resources such as pins.

11.1 Configuring Hardware Peripherals

Upon reset, most hardware peripherals are **disabled by default** and must be **configured and enabled** by writing to the appropriate peripheral registers. This is often done once at the start of the program, but can also be reconfigured dynamically if required, depending on the application. Information about peripheral registers is found in the datasheet and often recommended steps are also provided. In general:

1. Set bits on peripheral registers to configure the peripheral in the correct mode
2. Enable peripheral interrupts and define an associated ISR, if required
3. Enable the peripheral

It is best practice to globally disable interrupts when configuring peripherals.

11.2 Timers

Timers provide precise measurements of **elapsed clock cycles** in hardware, independent of software and the CPU. Timers are used to generate periodic events (via an interrupt), measure time between two events, or generate periodic signals on a pin.

11.2.1 Timer Implementations

Most timer implementations use the same basic structure, a **counter** which is incremented or decremented by a clock/event/etc. By comparing the value of this counter, the timer can perform more complex behaviours such as generating an interrupt or changing pin state.

11.2.2 Timer Counters

The ATtiny1626 has two 16-bit counters, Timer Counter A and B (TCA/TCB). As both timers are highly configurable, the datasheet should be consulted for more information. The counter, accessible via the `CNT` register, increments by 1 each clock cycle. The clock cycle can be reconfigured with a prescaler to increase the duration of each clock cycle.

11.2.3 Timer Periods

To generate an event that occurs every T seconds, we must configure a timer with a period of T seconds. This can be done using the period register, `PER`, which is used to set the maximum value of the counter, so that when the counter reaches this value, it overflows back to 0. As the counter increments by 1 each clock cycle, the duration of a single clock cycle T_{clk} is given by:

$$T_{\text{clk}} = \frac{1}{f_{\text{main}}/\text{prescaler}},$$

where f_{main} is the frequency of the main clock and prescaler is the prescaler applied to the timer peripheral. Here we assume the main clock frequency to be 20 MHz/6, where the main clock has its own default prescaler of 6. If we want to generate an event every T seconds, we need to set the period register to be the number of counts of the timer clock required to reach the period T:

$$n = \frac{T}{T_{\text{clk}}}$$

Here, n is the value that should be written to the period register, `PER`. The value of the prescaler influences the timer period and timer resolution, as an increase in the prescaler leads to both an increase in the period duration and a decrease in the timer resolution (time elapsed between each count). Therefore, the smallest prescaler that allows the desired period should be chosen.

11.2.4 Timer Counter B Example Configuration

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

void tcb_init()
{
    TCB0.CTRLB = TCB_CNTMODE_INT_gc; // Configure TCB0 in periodic interrupt mode
    TCB0.CCMP = 3333;                // Set interval for 1 ms (3333 clocks @
     ↪  3.333 MHz)
    TCB0.INTCTRL = TCB_CAPT_bm;      // Invoke the CAPT ISR when the counter
     ↪  reaches CCMP
    TCB0.CTRLA = TCB_ENABLE_bm;      // Enable TCB0
}

int main(void)
{
    cli();
    tcb_init();
    sei();

    while (1);
}
```

11.3 Pulse Width Modulation

Pulse width modulation (PWM) is a technique used to generate a **periodic signal** with a variable duty cycle. The **duty cycle** D of a signal is a measure of the ratio of the HIGH time of the signal compared to the total PWM period.

$$D = \frac{T_{\text{HIGH}}}{T_{\text{HIGH}} + T_{\text{LOW}}} = \frac{T_{\text{HIGH}}}{T}$$

- A duty cycle of 0% corresponds to a signal that is always LOW
- A duty cycle of 100% corresponds to a signal that is always HIGH

PWM can be used as a form of digital to analogue conversion, where a **modulating signal** is used to set the duty cycle of a PWM output. In analogue, a triangular waveform known as the **carrier** is compared with this modulating signal so that the PWM output is HIGH when the modulating signal is greater than the carrier.

11.3.1 PWM Implementation

On the ATtiny1626, the carrier is generated by a timer counter (i.e., `TCA0.CNT`) and the modulating signal is the compare value `TCA0.CCMP`. By setting the compare value to a value less than the counter value, the PWM output's duty cycle can be controlled, via the following equation:

$$D_{\text{PWM}} = \frac{\texttt{TCA0.CMPn}}{\texttt{TCA0.PER} + 1}$$

11.3.2 PWM Brightness Control Example

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

void tca_init()
{
    // DISP EN
    PORTB.DIRSET = PIN1_bm;

    // Set waveform generation mode to single slope
    // Waveform output controls PA1 PWM (display brightness)
    TCA0.SINGLE.CTRLB = TCA_SINGLE_WGMODE_SINGLESLOPE_gc | TCA_SINGLE_CMP1EN_bm;
    TCA0.SINGLE.PER = 0xFF;                    // Set period to some value
    TCA0.SINGLE.CMP1 = 0xFF;                   // 100% duty cycle
    TCA0.SINGLE.CTRLA = TCA_SINGLE_ENABLE_bm; // Enable TCA0
}

int main(void)
{
    cli();
    tca_init();
```

```
    sei();

    while (1);
}
```

As the duty cycle of this PWM signal is set to 100%, the display will be at full brightness. To dynamically change the brightness of the display, the compare value can be changed using the buffered register `TCA0.CMP1BUF`. The same approach should be used to update the period register. This is done to ensure that a new value is updated only when the counter is at the bottom of the waveform.

11.4 Analog to Digital Conversion

Analogue to digital conversion (ADC) is a technique used to convert an analogue signal to a digital signal. The analogue signal is sampled at a regular interval and the sampled value is converted to a digital value. Digital quantities are both discrete in amplitude and time.

11.4.1 Quantisation

Discretisation in amplitude is referred to as **quantisation**. Each amplitude is assigned a digital **code**. The **code width** determines the **amplitude resolution** and introduces **quantisation error**.

11.4.2 Sampling

Discretisation in time is referred to as **sampling**. The **sampling rate** determines the **time resolution** and introduces **aliasing error**. This rate is typically the period of the CPU clock.

11.4.3 ADC Implementation

Analogue to digital conversion is the process of discretising a continuous signal (typically a voltage) into a digital code. Specialised hardware called an **analogue to digital converter** (ADC) performs this function. The ADC samples the analogue signal at a regular interval at an instant in time, and converts the sampled value to a digital value.

11.4.4 ADC Potentiometer Example

```
#include <stdio.h>

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

void adc_init()
{
    // Select AIN2 (potentiometer R1)
    ADC0.MUXPOS = ADC_MUXPOS_AIN2_gc;
```

```
    // Need 4 CLK_PER cycles @ 3.333 MHz for 1us, select VDD as ref
    ADC0.CTRLC = (4 << ADC_TIMEBASE_gp) | ADC_REFSEL_VDD_gc;
    // Sample duration of 64
    ADC0.CTRLE = 64;
    // Free running
    ADC0.CTRLF = ADC_FREERUN_bm;
    // Select 8-bit resolution, single-ended
    ADC0.COMMAND = ADC_MODE_SINGLE_8BIT_gc | ADC_START_IMMEDIATE_gc;

    // Enable ADC
    ADC0.CTRLA = ADC_ENABLE_bm;
}

int main(void)
{
    cli();
    adc_init();
    sei();

    while (1)
    {
        printf("%u\n", ADC0.RESULT0);
    }
}
```

11.5 Serial Communication

Serial communication is the process of transmitting data **one bit at a time**. On a microcontroller, this is typically done via a digital I/O pin. The form of serial communication is determined by the **protocol** and **physical interface** used. The protocol specifies how the data is arranged, and the timing of bit transfers. The physical interface specifies the electrical characteristics of the communication medium (i.e., voltage level used to represent binary values). For two devices to communicate, they must both use the same protocol and physical interface.

11.5.1 Serial Communication Terminology

- **Transmit**: to send data, often abbreviated to **Tx**.
- **Receive**: to receive data, often abbreviated to **Rx**.
- **Simplex**: unidirectional communication. Requires one wire as data only flows in one direction.
- **Half-duplex**: bidirectional communication, occurring in one direction at a time. Requires one wire as data flows in one direction at a time.
- **Full-duplex**: bidirectional communication, occurring simultaneously. Requires two wires as data flows in both directions simultaneously.

- **Synchronous**: communication relying on a shared clock. Requires one wire for the clock signal.
- **Asynchronous**: communication that does not rely on a shared clock.

There are many serial interfaces that can be used in embedded systems for communication. Some common interfaces include:

- **UART**: Universal asynchronous receiver/transmitter.
- **SPI**: Serial peripheral interface.
- **I^2C**: Inter-integrated circuit.
- **CAN**: Controller area network.
- **I^2S**: Inter-IC sound.

11.5.2 UART

UART is a simple and cost effective serial communication protocol. As it is asynchronous, its clock is not shared between the two communicating devices. Instead, the sender and receiver must agree on a **baud rate** (the number of bits transmitted per second). This is typically in the range of 9600 baud to 115 200 baud (with a 2 M baud maximum). UART is a frame based protocol, where each frame is signalled by a start bit (always LOW), and is fixed in length and format. UART can be used in both full-duplex or half-duplex, depending on the hardware implementation, where the transmitter and receiver are fully independent. This means either a 1- or 2-wire mode is possible (plus 1 for GND).

11.5.3 UART Frame Format

The UART frame format is as follows:

- **Start bit**: always LOW.
- **Data bits**: 5 to 9 bits of data.
- **Parity bit**: optional bit used to detect errors.
- **Stop bit**: always HIGH.
- **Idle**: in the idle state, the line is HIGH.

The parity bit is used to detect errors in the data bits. It allows the receiver to detect a single-bit error in the frame. The parity bit can be configured to either be odd or even parity.

- For **even** parity, the total number of 1s in the data and parity bits must be even.
- For **odd** parity, the total number of 1s in the data and parity bits must be odd.

If a parity error is detected in a received frame, the receiver may choose to reject the frame.

11.5.4 USART0 on the ATtiny1626

The USART (Universal Synchronous/Asynchronous Receiver/Transmitter) peripheral is used to configure UART communication on the ATtiny1626. The transmission operation is as follows:

1. The user loads the data for transmission into the `TXDATA` register.
2. When the TX shift register is empty, the data will immediately be copied into the shift register.
3. Data is shifted out from the TX shift register, one bit at a time, according to the baud rate.
4. The transmitter is double-buffered so that:
    - If a second byte is loaded into the `TXDATA` register before the first byte has finished transmitting, this byte will be transferred into the TX buffer and transmitted after the first byte.
    - Additionally, writing a third byte will cause it to remain in the `TXDATA` register, until the previous two bytes have been transmitted.

The reception operation is as follows:

1. The start of an incoming frame is detected based on a falling edge on the RX line.
2. Data is shifted into the RX shift register, one bit at a time, according to the baud rate.
3. Once the correct number of data bits have been shifted into the shift register, the data will be copied into the `RXDATA` register.
4. The receiver is double-buffered so that:
    - If a second byte is shifted out of the shift register before the first byte is read, it will be stored in the RX buffer.
    - Additionally, a third byte will remain in the RX shift register until the RX buffer is empty.

For more information about the USART0 peripheral, see the datasheet.

11.5.5 USART0 Example Configuration

```
void uart_init(void)
{
    PORTB.DIRSET = PIN2_bm;                         // Output enable TX pin
    USART0.BAUD = 1389;                             // 9600 baud
    USART0.CTRLA = USART_RXCIE_bm;                  // Enable RX interrupt
    USART0.CTRLB = USART_RXEN_bm | USART_TXEN_bm; // Enable RX and TX
}
```

11.5.6 Serial Peripheral Interface

SPI is a synchronous serial communication protocol where a clock is transmitted to allow for higher bit rates in the range 10 MHz to 20 MHz. SPI is typically used for high-speed, inter-IC communications (communication with other peripherals) over short distances. SPI is also full-duplex, and can be configured to use a 2-, 3-, 4-wire modes (plus 1 for GND). It can be used to communicate with multiple devices simultaneously, using **chip select** (CS) (or slave select) lines to select the device to communicate with. Typically, in a master-slave model, the master device controls the clock.
The SPI peripheral can be used to interface to devices that produce or consume a serial bit stream, clocked or otherwise. The clock phase and polarity can also be modified to suit the device being interfaced to. It is also possible to have multiple masters on the same bus, but this requires a careful mechanism to arbitrate access to the bus.

11.5.7 SPI0 Example Configuration

```
void spi_init(void)
{
    PORTC.DIRSET = PIN0_bm | PIN2_bm; // Output enable SPI CLK and SPI MOSI

    PORTMUX.SPIROUTEA = PORTMUX_SPI0_ALT1_gc;

    SPI0.CTRLB = SPI_SSD_bm;  // Disable client select line
    SPI0.INTCTRL = SPI_IE_bm; // Enable SPI interrupts (to latch SPI DATA)
    SPI0.CTRLA = SPI_MASTER_bm | SPI_ENABLE_bm; // Enable SPI as master
}
```

11.5.8 Other Serial Protocols

- **I^{2}C**: Inter-integrated circuit.
  - Very common on microcontrollers, suitable for short distances only (typically < 300 m m). Widely used for external peripherals.
  - Typically, up to 400 k baud.
  - Half-duplex, synchronous, 2-wire bus, with bidirectional signalling, where devices use open-drain outputs.
- **CAN**: Controller area network.
  - Very prevalent automotive and industrial standard, suitable for medium distances (40 m to 500 m)
  - Robust and reliable (safety critical systems)
  - Built-in message priority and arbitration
  - Typically up to 1 M baud
  - Half-duplex, asynchronous, 2-wire bus (one differential pair)

- Very precise timing requirements compared with UART
- Complex protocol and controller

11.5.9 Polled vs Interrupt Driven

In a polled model for serial communication, the CPU is continuously checking the status of the peripheral to see if it is ready to transmit or receive data. This is referred to as a **blocking** read/write, as the program does not proceed until the read/write completes. This delay may be significant for a slow serial interface as the CPU cannot do anything else while waiting for the peripheral to complete the operation.
Alternatively, we can use an interrupt-driven model, where we can use interrupts to signal when the peripheral is ready for new data to be read/written. In this model, no delays are incurred by the CPU, and we are guaranteed that data is already ready to be read/written. The read/write operations are also completed in a deterministic amount of time, and the CPU can do other tasks while waiting for the peripheral to complete the operation.

11.6 Serial Communications on the QUTy

11.6.1 Virtual COM Port via USB-UART Bridge

The CP2102N is a USB-UART bridge that allows the QUTy to communicate with a PC via a USB cable. On the USB side (host/computer), it presents itself as a virtual COM port (VCP), and on the microcontroller side, it presents itself as a UART interface. The bytes written via UART are received by the VCP, and vice versa. Note that the TX pin (output) of the microcontroller is connected to the RX pin (input) of the VCP, and vice versa.
The UPDI pin is used to program the flash memory of the QUTy with a program. It shares the USB-UART bridge with the UART interface, which necessitates a switch to toggle between the two.

11.6.2 Controlling the 7-Segment Display

The 7-segment display is interfaced to the microcontroller by a 74HC595 **shift register**. A shift register is a device which translates **serial** input/output into **parallel** input/output. On the QUTy, it takes a serial, 1-bit output from the microcontroller and uses this to control, in parallel, the 7-segment display, plus a digit select signal. The 74HC595 takes a **clocked serial data stream** as an input, which makes interfacing via the SPI peripheral very simple.
Q0-Q6 on the shift register control the 7 segments of the display, and Q7 controls the digit select. The first bit clocked out of the microcontroller will set the state of Q7, and the last bit clocked out will set the state of Q0. To latch the data shifted into the shift register and consequently update the state of Q0-Q7, requires a **rising edge** on the DISP LATCH net.

11.6.3 Time Multiplexing

As only one side of the 7-segment display can be illuminated at a time, we can use a **time multiplexing** scheme, where we periodically illuminate each digit for a short duration. When a sufficiently large refresh rate is used, the human eye perceives both digits as being illuminated at the same time. If we wish to display a number, we can simply store the state of each digit and

switch between them using an interrupt-driven timer. For example, to configure a 100 Hz display, we must configure a total period of 10 ms (across both digits). This corresponds to a switch period of 5 ms, so that both digits are illuminated for an equal amount of time, leading to a uniform brightness across both digits.

```
// Bytes latched onto 7-segment display
volatile uint8_t left_byte = DISP_0 | DISP_LHS;
volatile uint8_t right_byte = DISP_0;

// 5ms interrupt
ISR(TCB0_INT_vect)
{
    static uint8_t current_display_side = 0;

    if (current_display_side)
    {
        SPI0.DATA = left_byte;
    }
    else
    {
        SPI0.DATA = right_byte;
    }

    // Toggle side
    current_display_side ^= 1;

    // Clear interrupt flag
    TCB0.INTFLAGS = TCB_CAPT_bm;
}

ISR(SPI0_INT_vect)
{
    // Latch the byte
    PORTA.OUTCLR = PIN1_bm; // Prepare for rising edge
    PORTA.OUTSET = PIN1_bm; // Generate rising edge

    // Clear interrupt flag
    SPI0.INTFLAGS = SPI_IF_bm;
}
```

The display is updated by writing to the variables `left_byte` and `right_byte`.

11.7 Pushbutton Handling

Pushbuttons have two possible states, **pressed** and **released**. Given an active-low pushbutton,

- when a pushbutton is pressed, the corresponding net is pulled LOW, and the pins value is `0`.

- when the pushbutton is released, the corresponding net is pulled HIGH, and the pins value `1`.

The state of individual pushbuttons can be read using bitwise AND operations with the `PORTA.IN` register.

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

PORTA.PIN4CTRL = PORT_PULLUPEN_bm;

while (1)
{
    if (PORTA.IN & PIN4_bm)
    {
        // Pushbutton is released
    }
    else
    {
        // Pushbutton is pressed
    }
}
```

In this loop structure, the state of the pushbutton will be in the pressed state until the pushbutton is released. This may not be desirable if we want to perform a single action when the pushbutton is pressed. To solve this, we must respond to a *change in state*.

- A **falling edge** is created when the pushbutton is pressed (transition from `1` to `0`).
- A **rising edge** is created when the pushbutton is released (transition from `0` to `1`).

This is known as **edge detection**. To implement this in C, we can use the XOR operator (`^`) to detect a change in the signal. To identify the direction of the edge, we can additionally condition the result with one of the states.

```
uint8_t pb_previous_state = 0xFF;
uint8_t pb_current_state = 0xFF;

while (1)
{
    pb_previous_state = pb_current_state; // save previous state
    pb_current_state = PORTA.IN;          // update with latest measurement

    uint8_t pb_edge = pb_previous_state ^ pb_current_state;
    uint8_t pb_falling_edge = pb_edge & pb_previous_state;
    uint8_t pb_rising_edge = pb_edge & pb_current_state;

    // alternatively if we do not need the edge
    uint8_t pb_falling_edge = pb_previous_state & ~pb_current_state;
    uint8_t pb_rising_edge = ~pb_previous_state & pb_current_state;
}
```

11.7.1 Pushbutton Sampling

The actuation of mechanical pushbuttons is slow and therefore it is important to sample pushbutton states fast enough to detect changes in state. **Latency** refers to the delay between user input and the reaction of a system. For user input, latency should be acceptably small[6]. Latency as low as 2 ms is perceptible in particular user input applications, however latency between 20 ms to 60 ms is acceptable for most applications.

11.7.2 Switch Bounce

Switch bounce is an artefact of electromechanical switches where switch contacts bounce back and forth after a switch is actuated. This results in a signal that is not stable and can cause false positives when detecting a change in state. As a digital system can sample a voltage much faster than a mechanical switch can change state, the system may detect multiple transitions of the switch state.
To prevent this, we **debounce** pushbuttons either by using a debouncing circuit, or through software. To implement this in software, we can take multiple samples of a pin and only accept a change in state if these samples are consistent over multiple samples. This can be implemented using an ISR to capture the state of the pushbutton at regular intervals.

```
// this variable is used instead of PORTA.IN when detecting edges
volatile uint8_t pb_debounced_state = PIN4_bm;

// Periodic 5ms interrupt
ISR(TCB1_INT_vect)
{
    static uint8_t counter = 3;

    // Capture the state of the pushbutton from the port pin
    uint8_t pb_sample = PORTA.IN & PIN4_bm;
    // Detect a change in state
    uint8_t pb_edge = pb_sample ^ pb_debounced_state;

    if (pb_edge)
    {
        if (counter-- == 0)
        {
            // Update debounced state
            pb_debounced_state = pb_sample;

            // Reset counter
            counter = 3;
        }
    }
    else
    {
```

[6] What is acceptably small depends on what magnitude of latency is perceptible and is specific to the application.

```
        // Reset counter
        counter = 3;
    }

    // Clear interrupt flag
    TCB1.INTFLAGS = TCB_CAPT_bm;
}
```

The above implementation can only debounce a single pushbutton, as the counter corresponds to a single pushbutton, S1. To debounce multiple pushbuttons, we can declare counters for each pushbutton, however this can lead to a large amount of duplicate code. Instead, we can utilise vertical counters.

11.7.3 Vertical Counters

Instead of using a counter variable for each pushbutton, we can use the bits of a single variable to represent the counters for each pushbutton. Doing so reduces the maximum value of the counter, but this is not a concern as we only require 3 samples. The following code implements a vertical counter for 4 pushbuttons.

```
// We can also use PIN4_bm | PIN5_bm | PIN6_bm | PIN7_bm, as we do not care about
 ↪  the other bits
volatile uint8_t pb_debounced_state = 0xFF;

// Periodic 5ms interrupt
ISR(TCB1_INT_vect)

    // Two vertical counters for a total of 4 counter states
    static uint8_t counter0 = 0;
    static uint8_t counter1 = 0;

    // Capture the state of the pushbuttons from the port pin
    uint8_t pb_sample = PORTA.IN;
    // Detect a change in state
    uint8_t pb_edge = pb_sample ^ pb_debounced_state;

    // Update counters
    // If the state of the pushbutton has changed, increment the counter
    counter1 = (counter1 ^ counter0) & pb_edge;
    counter0 = ~counter0 & pb_edge;

    // Update debounced state if counter reaches 3, or immediately on falling
     ↪  edge
    pb_debounced_state ^= (counter1 & counter0) | (pb_edge & pb_previous_state);

    // Clear interrupt flag
    TCB1.INTFLAGS = TCB_CAPT_bm;
```

```
}
```

This code allows us to debounce up to 8 pushbuttons using a single 8-bit variable. The debounced state of the pushbuttons is stored in `pb_debounced_state` and can be used to perform edge detection similar to the code in the previous section. This variable is updated if the state of the pushbutton is consistent over 3 samples, or if a falling edge is detected. Note that if both falling and rising edges need to be detected, it is not recommended to immediately update the debounced state on a falling edge, as this leads to inconsistent latency between rising and falling edges.

12 State Machines

A state machine or finite state machine (FSM) is a mathematical model of computation in which a machine can only exist in one of a finite number of states. The machine transitions between states in response to inputs, and performs actions during transitions. A state machine is fully defined by its list of states, initial state, and the conditions for transitioning between states.

12.1 State Machine Implementation

To translate a state machine into a C program, we can make use of an enumerated type. Enumerated types are a special type of data that allow us to define a set of named constants. Enumerated types can be used to implement a state machine as follows:

- Each enumerator can be used to represent a state.
- A `switch` statement can be used to implement the behaviour in each state.
- An `if` statement can be used to implement the conditions for transitioning between states.

```
typedef enum
{
    START,
    STATE1,
    STATE2
} state_t;

// Initial state
state_t state = START;

// Configure outputs for START state

while (1)
{
    // State machine
    switch (state)
    {
        case START:
```

```
            if (condition1)
            {
                // Configure outputs for STATE1 state

                // Transition if condition is met
                state = STATE1;
            }
            break;
        case STATE1:
            if (condition2)
            {
                // Configure outputs for STATE2 state

                // Transition if condition is met
                state = STATE2;
            }
            break;
        case STATE2:
            if (condition3)
            {
                // Configure outputs for START state

                // Go back to start if condition is met
                state = START;
            }
            break;
        default: // Invalid state (program should never reach this state)
            // Configure outputs for START state

            // Go back to start
            state = START;
            break;
    }
}
```

12.2 Enumerated Types

Enumerated types are defined similarly to structures, via the **enum** keyword, and can be anonymous, or named. The values of an enumerated type are constants, called enumerators, that are assigned an integer value starting from 0.

```
typedef enum
{
    FALSE,
    TRUE
} boolean_t;
```

```
boolean_t b = TRUE; // b is assigned the value 1

b == TRUE; // TRUE is assigned the value 1, so this is true
b == 0; // FALSE is assigned the value 0, so this is false
```

While they can be compared to integers, it is recommended to use the enumerators in comparisons. Enumerated types can also be defined with explicit values, and can be used to represent bitmasks.

```
typedef enum
{
    MONDAY = 0b00000001,
    TUESDAY = 0b00000010,
    WEDNESDAY = 0b00000100,
    THURSDAY = 0b00001000,
    FRIDAY = 0b00010000,
    SATURDAY = 0b00100000,
    SUNDAY = 0b01000000,
    WEEKEND = SATURDAY | SUNDAY,
    WEEKDAY = MONDAY | TUESDAY | WEDNESDAY | THURSDAY | FRIDAY
} DAYS;

enum DAYS d = SATURDAY; // d is assigned the value 0b00100000

if (d & WEEKEND) // Check if d is a weekend day
{
    // Do something
}
```

12.3 Switch Statements

A `switch` statement is a control structure that allows us to select a block of code to execute based on the value of an expression. When a case is matched, a `break` statement may be used to prevent the program from falling through the case, however this may be omitted if two states perform the same tasks. The `default` case is executed if no case is matched.

13 Serial Protocols

A serial protocol is an agreed-upon standard by which two devices can communicate with each other, enabling them to exchange data. UART and SPI are standards for transmitting data, but do not ascribe any meaning to the data.

13.1 Serial Protocol Design

13.1.1 Requirements for a Serial Protocol

A serial protocol must:

- able to receive data during a transmission
- able to recover from errors
- engage in flow control
- be simple to implement/understand for both the transmitter and receiver

13.1.2 Symbols

A symbol is the fundamental data type used in serial communication protocols, which can be comprised of several bits. The number of bits is usually set by the underlying medium and depends on the baud rate. Smaller symbols are more flexible and allow for more symbols to be transmitted, whereas larger symbols are more efficient and allow for more data to be transmitted.

13.1.3 Messages

If the information to be exchanged can be entirely encoded within a single symbol, there is no need for a message structure. However, more complex protocols require a message structure for large quantities of data or information of variable length. This is done by dividing the communication into discrete messages.

13.1.4 Encoding

The choice of encoding may also be of concern, depending on the communication medium, symbol length, and other factors such as human readability. For example using the entire ASCII character set may not be desirable as it is not human-readable. Human-readable encoding schemes usually limit the number of symbols to a small subset of the ASCII character set:

- ASCII `32-126` (`0x20-0x7E`) which uses 8-bit symbols
- Base64 (`0-9`, `A-Z`, `a-z`, `+`, `/`) which encodes 6 bits into an 8-bit symbol
- Hexadecimal (`0-9`, `A-F`) which encodes 4-bits per symbol

13.1.5 Message Structure

Messages typically contain the following information:

1. A **start sequence** to indicate the beginning of a message
2. An **identifier** to indicate what type of message is being sent (if the protocol requires multiple messages)
3. A **payload** containing the data specific to the message

4. A provision for **escape sequences** to allow for special characters (e.g., arbitrary data is not confused with start sequences)
5. A **checksum** (or message digest), to ensure the integrity of the message
6. A **stop sequence** to indicate the end of a message

13.1.6 Start Sequences

As a serial communication transmits a sequence of symbols with no structure, there is no guarantee that the entire message is received. To address this, a start sequence is used to indicate the beginning of a message. The start sequence is usually a fixed number of unique symbols that do not appear in the payload, providing a synchronisation point. If payloads need to contain arbitrary sequences of symbols, escape sequences may be used.

13.1.7 Multi-Symbol Start Sequences

While a single symbol start sequence is simple, a multi-symbol start sequence has potential benefits:

- Reduces need for escape sequences
- Reduced likelihood of misinterpreting corrupted data as a start sequence

13.1.8 Sub-Symbol Start Sequences

If messages are encoded with fewer than 8 bits per symbol, the remaining bits can be used to encode a start sequence. For example, in UTF-8 encoding, the high bit is always cleared in the first byte of a sequence.

13.1.9 Message Identifiers

Serial protocols often have multiple categories of messages that may be transmitted. Commonly a fixed-length identifier is transmitted so that the receiver can respond with the appropriate action, or know when to expect a payload.

13.1.10 Payloads

A payload is used when a message identifier alone is insufficient to convey the information required. Payloads should be as small as possible to reduce the overhead of the protocol, as longer payloads increase the risk of transmission errors, so it may be preferable to split a large payload into multiple messages.

13.1.11 Payload Length

Payloads may be of both fixed and variable length, depending on the protocol.

- For fixed length payloads, the message type itself may define the payload length and hence will know when to expect the end of the payload.
- For variable length payloads, the payload length is encoded in the message itself, by either specifying the length within the payload, or by using a delimiter to indicate the end of the payload.

13.1.12 Variable Length Payloads

When variable length payloads are expected, two strategies are commonly used:

- Encode the payload length at the start of the payload, either with one or two symbols for (1–256) or (1–65536) bytes respectively.
- Use a **sentinel** to indicate the end of the payload. This is a special symbol that is not used in the payload, such as a null character.

Note that the second strategy requires the receiver to be able to buffer the entire payload before processing it. If the payload is too large, this may not be possible.

13.1.13 Escape Sequences

When there is potential ambiguity for whether a given symbol or sequence of symbols is part of a sequence, a payload, or a sentinel for a payload, escape sequences may be necessary to handle certain characters. In C, a backslash (\) is used to escape characters like the double quote ("), to tell the compiler that the character is not the end of the string.
This may not be feasible in a serial protocol as the backslash may be missed during transmission, and the next character may be treated as a start sequence. To address this, the escape sequence should not contain the symbol it is escaping. Instead, an alternate sequence should be used to represent symbols when they are part of a payload or other contexts. Note that the escape sequence itself may be part of the payload, so it is important to account for this also.

13.1.14 Handshakes

A protocol where the sender purely transmits data does not know whether the same information has been received and handled by the receiver. As such, it is common for protocols where only side is sending information and the other is receiving, to have the receiving side acknowledge what it has received (if the serial communications medium is half-duplex or full-duplex). The most common form of handshakes are **ACK** and **NACK** messages (for acknowledged and not acknowledged).

- ACK indicates that the message was received, and that its contents were understood.
- NACK indicates that the message was received, but that there was an error in the message. For example if the message was malformed, failed its checksum, or was unable to be processed.

  In these situations, the sender may retransmit the message, or send a different message.

13.1.15 Message Verification

As serial communications are prone to transmission errors, it is important to verify that the message was received correctly. This is done by including a checksum in which the transmitter computes a value based on the contents of the message, such as the sum of the bytes, and transmits it with the message. The receiver then computes a similar checksum based on what it receives and verifies that it matches the transmitted checksum. While this simple checksum detects many transmission errors, it is not guaranteed to handle symbols sent out of order, or symbols that were corrupted without changing the checksum.

13.1.16 Flow Control

When the receiver operates on little power or low storage, it may not be able to process data when it is transmitted at its usual rate. Hence, the sender may respond with a flow control message, such as WAIT, to indicate that it is currently busy processing the previous message, and RESUME when it is ready to receive more data.

13.2 Serial Protocol Parsing

Many serial protocols are designed to be simple to parse, due to the limitations of hardware used in serial communication. However, it is important to ensure that concerns around timing, state, and buffers are addressed to ensure that the parser is robust and reliable. As other actions may be performed during the parsing of a message, it may not be possible to use blocking functions such as `scanf()`. Similarly, a periodic interrupt may not be feasible if symbols are not sent frequently. As such, a better choice may be to use the `UART_RX` interrupt to handle a single character at a time, using a state machine to handle the parsing of the message.
This state machine can be placed within the interrupt handler, or in a separate function that is called in the main loop when new characters are received[7]. The state machine should be designed to handle the following states:

1. **Idle**: The parser is waiting for a start sequence (and ignores all other symbols)
2. **Start Sequence**: The parser is receiving the start sequence
3. **Message Identifier**: The parser is receiving the message identifier
4. **Payload**: The parser is receiving the payload (may be separate states for various identifiers)
5. **Checksum**: The parser is receiving the checksum

[7] This may be more suitable if the state machine consumes many CPU cycles.