Microprocessors and Digital Systems

Semester 2, 2022

Dr Mark Broadmeadow

Tarang Janawalkar

Contents

Chapter 1

Microcontrollers

1.1 Architecture of a Computer

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

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

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

1.2 Microprocessors & Microcontrollers

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

1.3 ATtiny1626 Microcontroller

The ATtiny1626 microcontroller has the following features:

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

1.3.1 Flash Memory

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

1.3.2 SRAM

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

1.3.3 EEPROM

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

1.4 AVR Core

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

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

- 8-bit Reduced Instruction Set Computer (RISC)
- 32 working registers (R0 to R31)
- Program Counter (PC) — location in memory where the program is stored
- Status Register (SREG) — stores key information from calculations performed by the ALU (i.e., whether a result is negative)
- Stack Pointer — temporary data that doesn't fit into the registers
- 8-bit core — all data, registers, and operations, operate within 8-bits

1.5 Status Register

The status register is a 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

1.6 Program Execution

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

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

This is illustrated in the following figure:

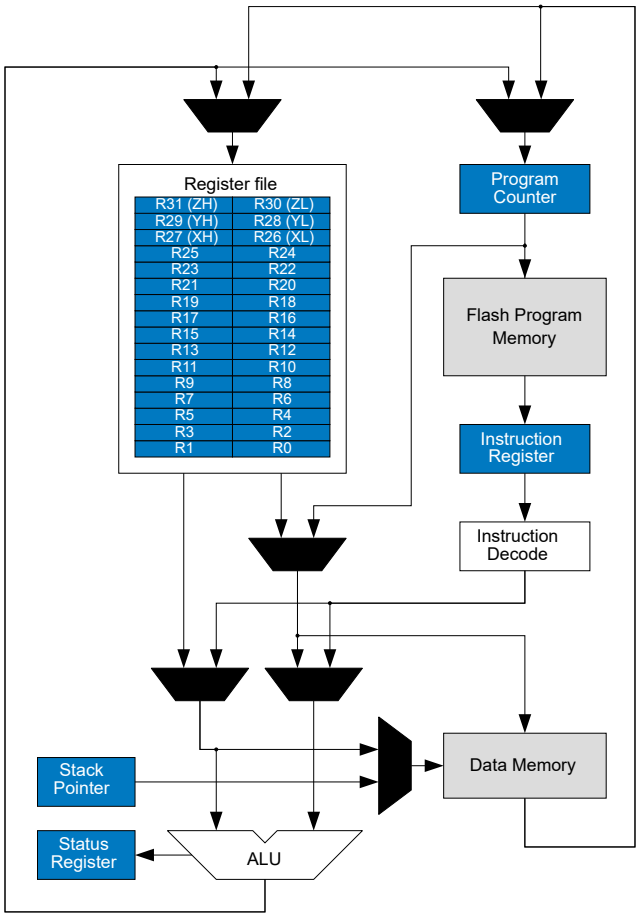


Figure 1.1: Program execution on the ATtiny1626.

1.7 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 of the available instructions, and how they are translated into opcodes
- Instructions fall loosely into five categories:
  - Arithmetic and logic — ALU
  - Change of flow — jumping to different sections of the code or making decisions
  - Data transfer — moving data in/out of registers, into the data space, or into RAM
  - Bit and bit-test — looking at data in registers (which bits are set or not set)
  - Control — changing what the CPU is doing

1.8 Interacting with memory and peripherals

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

1.9 Assembly code

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

Chapter 2

Digital Representations and Operations

2.1 Representation

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

2.1.1 Binary Representation

Bits are written right-to-left from **least significant** to **most significant** bit.

- 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.

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

2.1.2 Hexadecimal Representation

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

2.1.3 Numeric Literals

When a fixed value is declared directly in a program, it is referred to as a **literal**. Prefixes denote various bases:

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

2.2 Unsigned Integers

The **unsigned integers** represent the set of counting (natural) numbers, starting at 0. In the **binary system** (base-2) the unsigned integers are encoded using a sequence of binary digits (0–1). For example,

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

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

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

Table 2.1: Range of available values in binary representations.

2.3 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)

2.3.1 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. 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)

2.3.2 One's Complement

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

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

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

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

2.3.3 Two's Complement

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

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

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

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

2.4 Logical Operators

2.4.1 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`.

2.4.2 Negation

A unary operator used to **invert** a bit.

a	`NOT` a
0	1
1	0

2.4.3 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

2.4.4 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

2.4.5 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

2.4.6 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.

2.5 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 $0 \to 1$, or $1 \to 0$)

2.5.1 Setting Bits

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

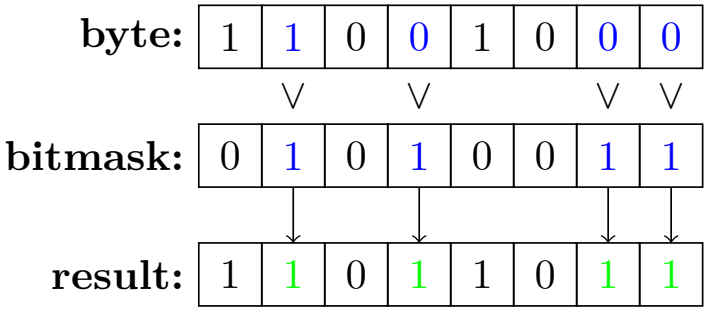


Figure 2.1: Setting bits using the logical or.

2.5.2 Clearing Bits

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

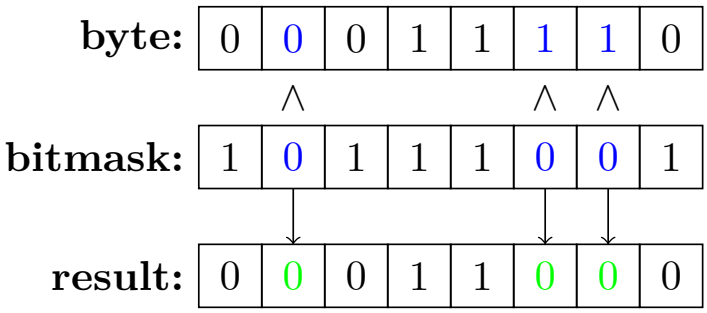


Figure 2.2: Clearing bits using the logical and.

2.5.3 Toggling Bits

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

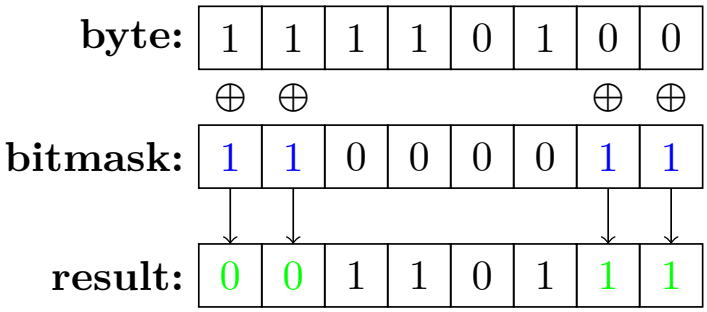


Figure 2.3: Toggling bits using the logical exclusive or.

Other bitwise operations act on the entire byte.

- One's complement (bitwise `NOT`)
- Two's complement (bitwise `NOT` + 1)

- Shifts
  - Logical
  - Arithmetic (for signed integers)
- Rotations

2.5.4 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.

2.5.5 Two's Complement

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

2.5.6 Shifts

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

$$a \gg s$$

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

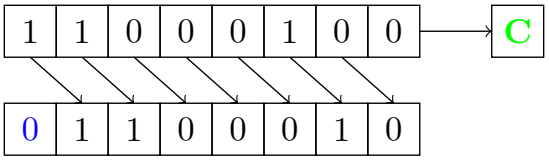


Figure 2.4: Right shift using `lsr` in AVR Assembly.

Similarly

$$a \ll s$$

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

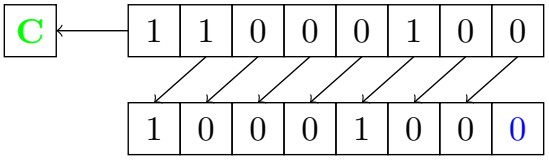


Figure 2.5: Left shift using `lsl` in AVR Assembly.

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

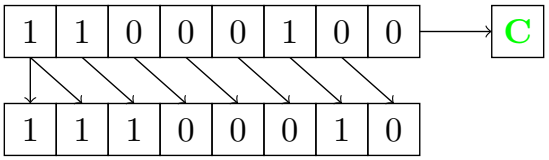


Figure 2.6: Arithmetic right shift using `asr` in AVR Assembly.

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

2.5.7 Rotations

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

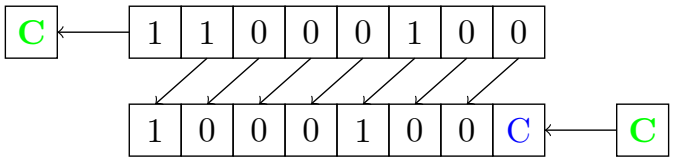


Figure 2.7: Rotate left using `rol` in AVR Assembly.

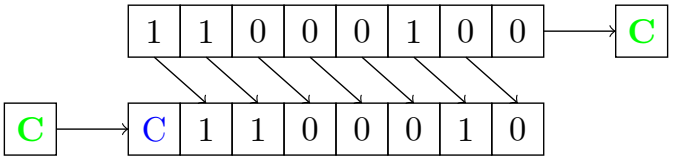


Figure 2.8: Rotate right using `ror` in AVR Assembly.

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

2.6 Arithmetic Operations

2.6.1 Addition

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

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

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

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

Below is a graphical illustration of the above code.

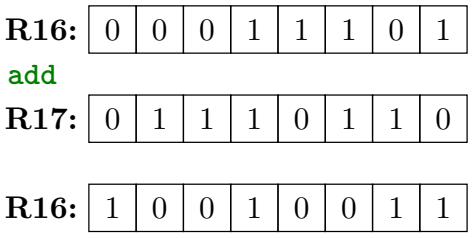


Figure 2.9: Overflow addition using `add` in AVR Assembly.

2.6.2 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.

```
1  ; Low byte
2  ldi r30, 0
3  ; High byte
4  ldi r31, 0
5
6  ; Empty byte for adding carry bit
7  ldi r29, 0
8
9  ; First number
```

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

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

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

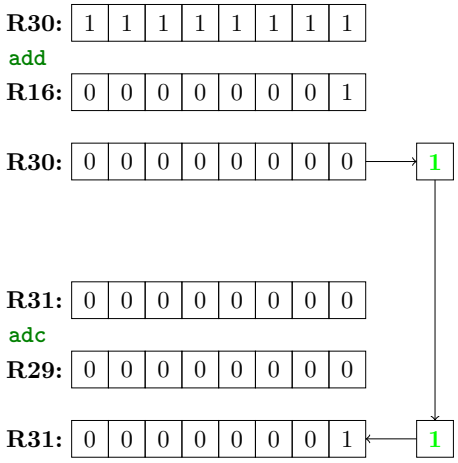


Figure 2.10: Overflow addition using `adc` in AVR Assembly.

2.6.3 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.

2.6.4 Multiplication

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

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

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

```
; First number
ldi r16, 13
; Second number
ldi r17, 43

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

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

2.6.5 Division

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

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

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

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

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

Chapter 3

Microcontroller Interfacing

3.1 Logic Levels

3.1.1 Discretisation

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

3.1.2 Logic Levels

For digital input/output (IO), conventionally:

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

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

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

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

3.1.3 Hysteresis

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

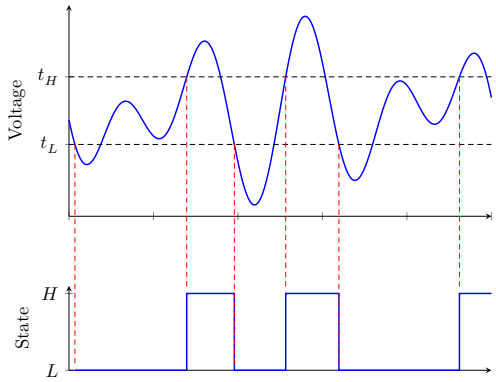


Figure 3.1: Example of hysteresis.

Given a transition:

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

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

3.2 Electrical Quantities

3.2.1 Voltage

Voltage v 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

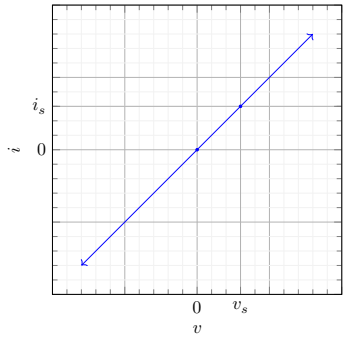
$$p = vi.$$

3.2.4 Resistance

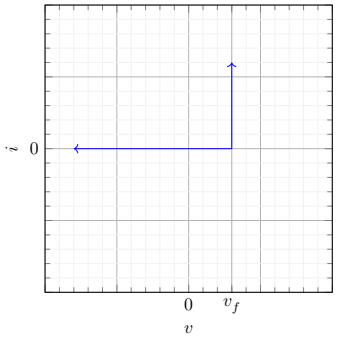
Resistance R is a property of a material to *resist the flow of current*, measured in **Ohms** (Ω). Ohm's law states that the voltage across a component is proportional to the current that flows through it:

$$v = iR.$$

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



(a) VI curve for Ohmic components.



(b) VI curve for diodes.

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

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

3.3 Electrical Components

3.3.1 Resistors

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

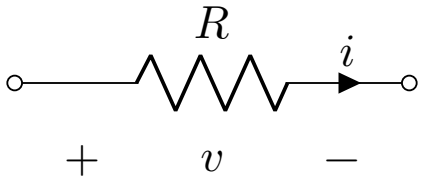


Figure 3.3: Resistor circuit symbol.

3.3.2 Switches

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

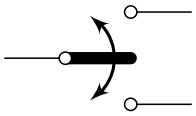
- In the **open** state, the switch **will not conduct**[1] current
- In the **closed** state, the switch **will conduct** current

Switches can take a variety of forms:

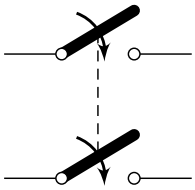
- **Poles** — the number of circuits the switch can control.
- **Throw** — the number of output connections each pole can connect its input to.
- Momentary or toggle action
- Different form factors, e.g., push button, slide, toggle, etc.

Switches are typically for user input.

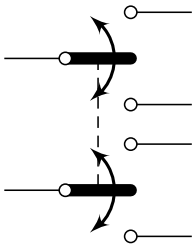
(a) Single pole single throw switch.



(b) Single pole double throw switch.



(c) Double pole single throw switch.



(d) Double pole double throw switch.

Figure 3.4: Various types of switches.

3.3.3 Diodes

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

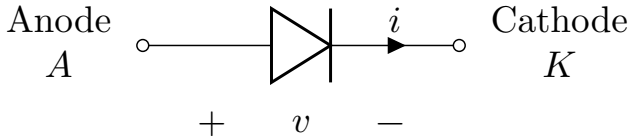
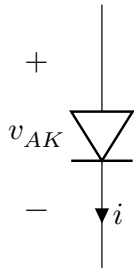


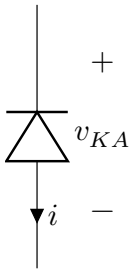
Figure 3.5: Diode symbol.

Diodes are a non-Ohmic device:

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



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



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

Figure 3.6: Diodes in forward and reverse bias.

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

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

3.3.4 Integrated Circuit

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

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

3.4 Digital Outputs

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

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

3.4.1 Push-Pull Outputs

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

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

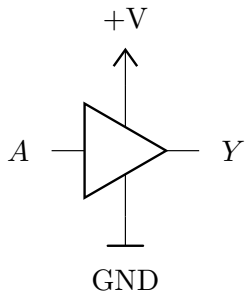


Figure 3.7: Push-pull output.

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

$$Y = A.$$

A	Y
LOW	LOW
HIGH	HIGH

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

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

3.4.2 High-Impedance Outputs

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

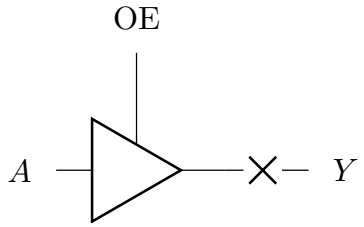


Figure 3.8: High-impedance output.

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

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

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

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

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

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

3.4.3 Pull-up and Pull-down Resistors

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

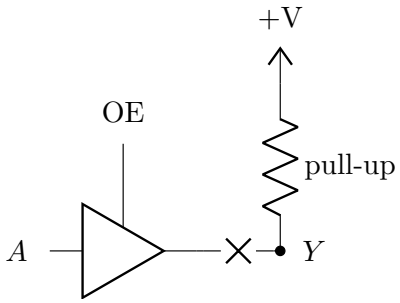


Figure 3.9: Pull-up resistor.

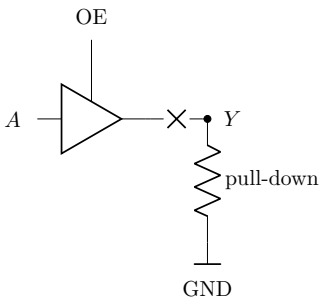


Figure 3.10: Pull-down resistor.

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

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

3.4.4 Open-Drain Outputs

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

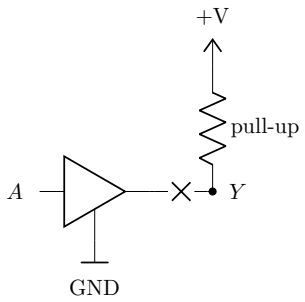


Figure 3.11: Open-drain output.

An open-drain output is either:

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

3.5 Microcontroller Pins

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

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

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

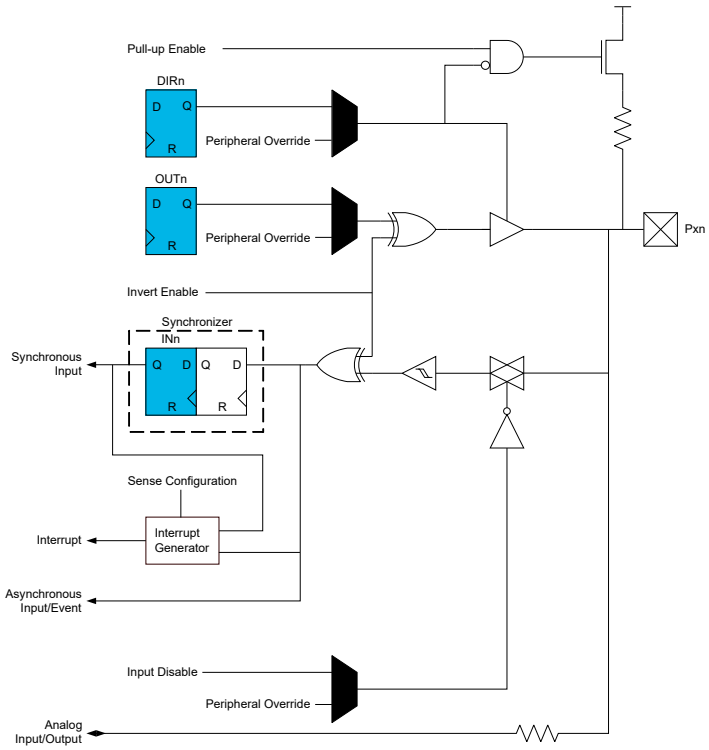


Figure 3.12: ATtiny1626 PORT block diagram.

To summarise this diagram:

- The data direction register (DIR) controls the push-pull output enable.
- The output driver register (OUT) drives the output state.
- The input register (IN) reads the output state.

- The internal pull-up register enabled through software.
- The physical voltage on the pin can be routed to an analogue to digital converter (ADC)
- Other peripheral functions can override port pin configurations and the output state.

3.5.1 Configuring an Output in Assembly

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

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

; Bitmask for pin 5
ldi r16, PIN5_bm

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

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

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

3.5.2 Configuring an Input in Assembly

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

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

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

- Certain peripherals can have their inputs/outputs mapped to different **sets of pins** through the PORTMUX.

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

3.6 Interfacing to Simple IO

3.6.1 Driving LEDs

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

3.6.2 Interfacing to LEDs

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

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

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

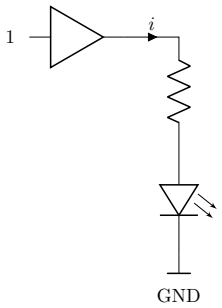


Figure 3.13: Active high configuration.

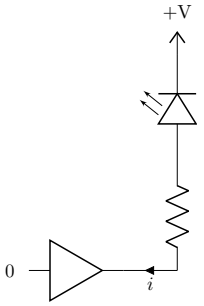


Figure 3.14: Active low configuration.

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

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

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

3.6.3 Switches as Digital Inputs

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

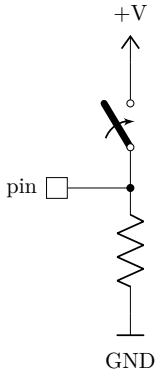


Figure 3.15: Active high configuration.

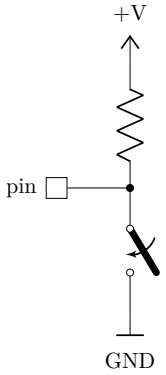


Figure 3.16: Active low configuration.

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

3.6.4 Interfacing to Switches

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

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

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

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

3.6.5 Interfacing to Integrated Circuits

For digital ICs,

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

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

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

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

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

3.7 Registers

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

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

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

3.8 Flow Control

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

3.9 Labels

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

```
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.
Additionally, labels can also be used as parameters to other immediate instructions if we store the high and low bytes in registers and wish to reference the location in an indirect jumping instruction.

3.10 Absolute and Relative Addresses

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

3.11 Branching

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

- Branch instructions
- Skip instructions

3.11.1 Branch Instructions

Branch instructions use the following logic:

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

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

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

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

3.11.2 Compare Instructions

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

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

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

new_location:
```

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

3.11.3 Skip Instructions

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

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

Same example which uses a 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 with branch instructions

```
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
```

Using branch instructions:

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

brne new_location ; Skip instructions if r17 != 0

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

new_location:
```

3.12 Loops

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

```
infinite_loop:
    ; Code to repeat
    rjmp infinite_loop
```

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

```
ldi r16, 10 ; Set counter to 10
loop:
    dec r16 ; Decrement counter
    brne loop ; Branch if counter != 0
```

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

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

    brne main_loop ; Branch if counter != 0
    rjmp button_pressed

button_pressed:
    ; Execute instructions
    rjmp main_loop ; Return to main loop
```

3.13 Delays

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

- The clock speed — frequency of the clocks oscillations (default: 20 MHz — configurable in CLKCTLR_MCLKCTRLA)
- The prescaler — reduces the frequency of the CPU clock through division by a specific amount; 12 different settings from 1x to 64x (default: 6 — configurable in CLKCTLR_MCLKCTRLA)

The clock oscillates at its effective clock speed:

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

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

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

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

```
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)
```

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

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

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 $((3 \times (2^8 - 1) + 4)\, T)$[4]. 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

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

3.14 Memory and IO

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

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

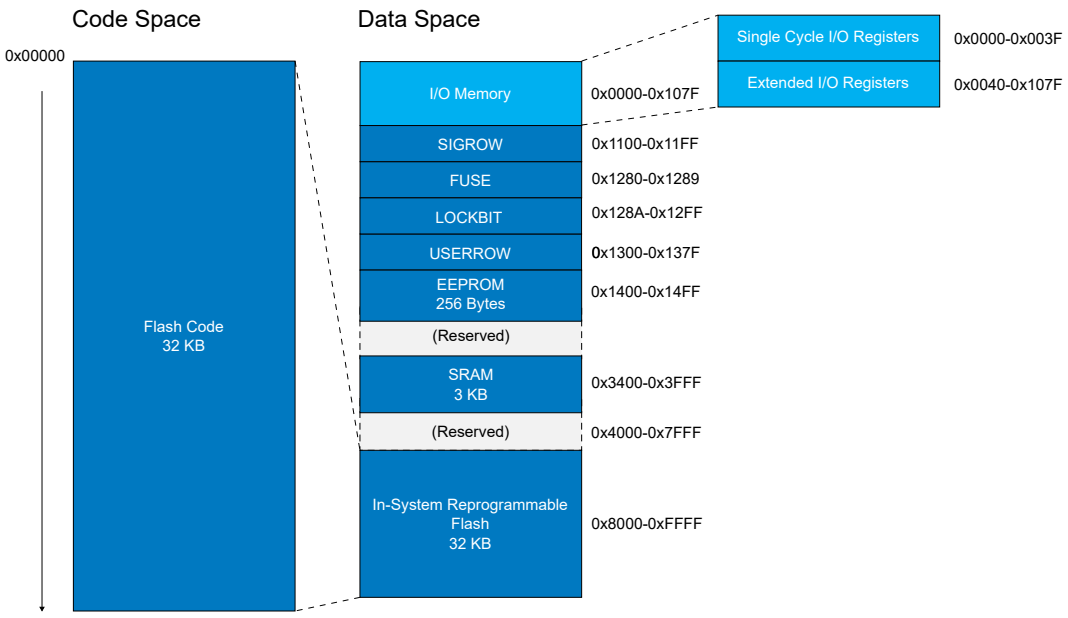


Figure 3.17: ATtiny1626 memory map.

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

- `lds` (load direct from data space to register)
- `sts` (store direct from register to data space)
- `ld` (load indirect from data space to register)
- `st` (store indirect from register to data space)
- `push`/`pop` (stack operations in 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 the name suggests, 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.

3.14.1 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
```

3.14.2 Load/Store Indirect with Displacement

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

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

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

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

3.15 Stack

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

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

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

```
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
```

3.16 Procedures

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

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

3.16.1 Saving Context

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

```
rjmp main_loop

procedure:
    push r16 ; Save r16 on the stack
    ; 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
```

3.16.2 Parameters and Return Values

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

```
rjmp main_loop

; Calculate the average of two numbers
; Inputs:
;     r16: first number
;     r17: second number
; Outputs:
;     r16: average
average:
    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 preferrable to return values using registers.

Chapter 4

Variables

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

4.1 Declaration

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

```
1 int x;
```

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

```
1 x = 4;
```

4.2 Initialisation

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

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

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

4.3 Types

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

4.3.1 Type Specifiers

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

4.3.2 Type Qualifiers

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

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

4.3.3 Portable Types

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

4.3.4 Exact Width Types

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

4.3.5 Floating-Point Types

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

Chapter 5

Literals

5.1 Integer Prefixes

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

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

5.2 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.

```
#include <stdio.h>

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

5.3 Floating Point Suffixes

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

- **Float** — `f`
- **Double** — `d`

5.4 Character and String Literals

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

5.4.1 Character Literals

The encoding of a character is platform-dependent, but typically characters 0–127 are defined through the American Standard Code for Information Interchange (ASCII), with slight variations depending on locale. In this character set, characters 0–31 and 127 are control characters, and characters 32–126 are printable characters. Characters 128–255 are typically defined by the local character set, and are not portable between platforms.
A character literal is a single character enclosed in single quotes, e.g., `'a'`, and escape sequences can be used to represent control characters, e.g., `'\n'`. Characters are stored in memory as 8-bit values, and are typically represented as unsigned integers in C.

5.4.2 String Literals

A string is a sequence of characters, terminated by a null character (`'\0'`). Strings can be defined using double quotes, e.g., `"Hello, world!"` or as an array of characters. When defined using double quotes, a null character is implicitly added to the end of the string, however this is not the case when defined as an array of characters.
If the size of the character array is specified for a character array, then

- If the size is greater than the length of the string, then the remaining elements of the array are initialised to 0.
- If the size is less than the length of the string, then the string is truncated to fit the array, and no null character is added.

When accessing strings to perform operations on them, we can utilise the null character to determine the length of the string. To modify a string, we can utilise pointer arithmetic to modify each character in-place. The `<string.h>` header file contains a number of functions for manipulating strings, including

- `strlen()` — Get length of string
- `strcpy()`, `strncpy()` — Copy string
- `memcpy()` — Copy data
- `strcmp()` — Compare strings
- `strchr()`, `strrchr()` — Find character in string
- `strstr()` — Find string in string
- `strcat()` — Concatenate two strings

5.4.3 Standard Input/Output

The C standard I/O header file `<stdio.h>` library provides a number of functions that read input and write output. Many of these functions work with the standard input and output devices, `stdin` and `stdout`. On a PC, these will normally read from or write to, a terminal by default. On the QUTy, these devices are non-functional by default, and must be configured to read from/write to USART0.
The `putchar()` and `getchar()` functions are used to write a single character to the standard output device and read a single character from the standard input device, respectively.

5.4.4 Formatted Output

The `printf()` function (print formatted) is used to write formatted output to the standard output device. The format string is a string that contains text and format specifiers, which are used to specify how the arguments are to be formatted. The format specifiers are prefixed by a percent sign (`%`).

```c
#include <stdio.h>

printf("Hello, world!\n"); // Print a string
printf("%d\n", 100); // Print a signed integer (or %i)
printf("%d%%\n", 100); // Escape a percent sign
printf("%u\n", 100); // Print an unsigned integer
printf("%o\n", 100); // Print an unsigned octal
printf("%x\n", 100); // Print an unsigned hexadecimal (%X uppercase)
printf("%p\n", &var); // Print a pointer
```

Preceding the format specifier with an optional **integer length modifier** will change the size of the argument that is formatted.

```c
#include <stdio.h>

printf("%d\n", 100); // Print a signed integer
printf("%hd\n", 100); // Print a signed short integer
printf("%ld\n", 100); // Print a signed long integer
```

Additionally, a number before the specifier can be used to specify the minimum width of the output, and a number after the period can be used to specify the number of decimal places.

```c
#include <stdio.h>

printf("%d\n", 100); // Print a signed integer
printf("%10d\n", 100); // Print a signed integer with a minimum width of 10
printf("%-10d\n", 100); // Print a signed integer with a minimum width of 10 and
                        // justify left
```

For floating point numbers, the `%f` specifier can be used to print a fixed-point number, and the `%e` specifier can be used to print a floating-point number in scientific notation. `%g` prints either depending on the size. A decimal point can be used to specify the number of decimal places to print.

```c
#include <stdio.h>

printf("%f\n", 100.0); // Print a float
printf("%.2f\n", 100.0); // Print a float with 2 d.p.
printf("%e\n", 100.0); // Print a float in scientific notation
```

For performance reasons, `printf()` is buffered, and will store characters in a buffer in SRAM. By default `stdout` is line-buffered, meaning that the buffer is flushed when a newline character is written to the buffer. `fflush(stdout)` can be used to flush the buffer.

5.4.5 Formatted Input

The `scanf()` function (scan formatted) is used to read formatted input from the standard input device. The format string is similar to that of `printf()`, except that the format specifiers are used to specify how the input is to be read.

```c
#include <stdio.h>

int i;
scanf("%d", &i); // Read a signed integer
// A signed integer input will be stored in i
// Otherwise, the function returns 0

char str[10];
scanf("abc%d", str); // Read a string with a prefix
// If the string is "abc123", then str will be "123"
// If the string is "5", then "5" will stay in the buffer and the function returns 0
// If the string is "abcd", then scanf will stop reading and return 0
// Otherwise, the function returns 0
```

If two or more format specifiers are used, then the function will read input until it reaches a whitespace character, and then stop.

```c
#include <stdio.h>

int i, j;
scanf("%d %d", &i, &j); // Read two signed integers
// If the input is "123 456", then i will be 123 and j will be 456
// If the input is "123", then i will be 123 and the function returns 0
// Otherwise, the function returns 0
```

Whitespace characters are ignored when reading input,

```c
#include <stdio.h>

char c;
scanf("%c", &c); // Read a character
// If the input is "a", then c will be 'a'
// If the input is " a", then c will be 'a'
// If the input is "  a", then c will be 'a'
```

A width specifier tells the function how many characters to read, this is useful to prevent buffer overflows.

```c
#include <stdio.h>

char str[10];
scanf("%9s", str); // Read a string with a maximum length of 9
// If the input is "123456789", then str will be "123456789"
// If the input is "1234567890", then str will be "123456789"
```

The asterisk (*) can be used to ignore input.

```c
#include <stdio.h>

int i;
scanf("%*d %d", &i); // Read a signed integer
// If the input is "123 456", then i will be 456
// If the input is "123", then the function will return 0
// Otherwise, the function returns 0
```

A scanset can be used to specify a set of characters that are allowed to be read.

```c
#include <stdio.h>

char c;
scanf("%[abc]", &c); // Read a character
// If the input is "a", then c will be 'a'
// If the input is "b", then c will be 'b'
// If the input is "c", then c will be 'c'
// Otherwise, the function returns 0
```

This behaves similarly to a regular expression, and can be negated by using the caret (^), ranges can be specified by using a hyphen (-), and the backslash (\) can be used to escape special characters.
If a width is specified for a scanset, then the function will read input until it reaches a character that is not in the scanset or until the maximum width is reached. This character will be left in the buffer.

Chapter 6

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.

6.1 Operation Precedence

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

6.2 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).

6.3 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, `? :`.

6.4 Assignment

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

```
int x = 5;
```

6.5 Multiple Assignment

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

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

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

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

6.6 Compound Assignment

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

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

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

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

Chapter 7

The Preprocessor

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

7.1 Includes

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

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

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

7.2 Header Files

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

```c
// add.c
int add(int x, int y)
{
    return x + y;
}
```

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

```c
// add.h
int add(int x, int y); // The variable names are not required in the prototype
```

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

```c
// main.c
#include <stdio.h> // Include printf definition (and other definitions)
#include "add.h" // Include add function definition

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

```

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

    return 0;
}
```

7.3 Definitions

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

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

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

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

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

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

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

    return 0;
}
```

Note that the semi-colon is omitted at the end of the macro definition, as it would also be substituted into the program. Only a single preprocessor 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).

Chapter 8

Pointers

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

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

This code declares a variable `ptr` that points to a `uint8_t`. Internally, a pointer contains a **memory address**, which on the ATtiny1626 is 16-bit.

8.1 Addressing

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

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

A more common usage of pointers is to *reference* **other variables**.

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

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

8.2 Dereferencing

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

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

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

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

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

8.3 Strings

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

```
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. We will see how to index into strings in the section on arrays.

8.4 Qualifiers

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

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

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

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

```
uint8_t a = 100; // Variable
const uint8_t *ptr = &a; // Points to `a` but treats it as constant

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

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

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

const uint8_t *ptr = &a; // Points to `a` but treats it as constant
ptr = &b; // Valid: `ptr` is not constant
```

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

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

uint8_t *const ptr = &a; // Points to `a` but cannot be reassigned
*ptr = 200; // Valid: `ptr` points to `a` which is not constant
ptr = &b; // Error: Cannot reassign `ptr`
```

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

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

const uint8_t *const ptr = &a; // Points to `a` but cannot be reassigned nor modified
ptr = &b; // Error: Cannot reassign `ptr`
*ptr = 200; // Error: Cannot modify `a` because `ptr` is constant
```

8.4.1 Pointers to Pointers

Pointers can also point to other pointers.

```
uint8_t a = 100; // Variable

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

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

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

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

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

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

```
uint8_t a = 100;               // Variable
uint8_t *ptr = &a;             // Points to `a`
const uint8_t **ptr1 = &ptr;   // Pointer to pointer to constant uint8_t
uint8_t * const *ptr2 = &ptr; // Pointer to constant pointer to uint8_t
uint8_t ** const ptr3 = &ptr; // Constant pointer to pointer to uint8_t
```

8.4.2 Pointer Arithmetic

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

```
uint8_t a = 100; // Variable
uint8_t *ptr = &a; // Points to `a`
ptr++; // Increment by 1 byte (size of uint8_t)
// ptr now points to the next byte after `a`
```

8.4.3 Void Pointers

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

```
void *ptr;
```

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

```
uint8_t a = 100;
void *ptr = &a; // Pointer to uint8_t
uint8_t *ptr2 = ptr; // Error: Cannot assign void pointer to uint8_t pointer
```

8.4.4 Size-of

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

```
uint8_t a = 100;
uint16_t b = 200;
sizeof(a); // Returns 1
sizeof(b); // Returns 2
```

8.5 Arrays

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

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

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

8.5.1 Character Arrays

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

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

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

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

8.5.2 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 is now {0, 10, 2, 3, 4, 5, 6, 7, 8, 9}
```

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

```
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;
}
```

8.5.3 Pointers and Arrays

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

```
uint8_t a[10];
uint8_t *ptr = a; // Equivalent to `uint8_t *ptr = &a[0]`
*ptr = 100; // `a` is now {100, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

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

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

uint8_t a[10];
func(a); // `a` is now {100, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

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

```c
uint8_t a[10];
a++; // Error: Cannot change the address of an array
```

8.5.4 Array Length

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

```c
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.

8.5.5 Copying Arrays

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

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

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

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

8.5.6 Multidimensional Arrays

Multi-dimensional arrays (or multiple subscript arrays) are used to hold multi-dimensional data.

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

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

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

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

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

```
    {7, 8, 9}
};

uint8_t rows = 3;
uint8_t cols = 3;

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

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

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

8.6 Functions

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

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

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

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

8.6.1 Parameters

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

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

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

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

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

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

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

```

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

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

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

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

8.6.2 Return Values

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

8.6.3 Function Prototypes

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

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

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

    uint8_t c = add(a, b);
}

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.

8.6.4 Passing by Reference

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

```
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);
}
```

8.6.5 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 set to that address when the function returns.
Local variables inside functions do not increase the explicit SRAM usage reported by the compiler. Rather, this memory will be allocated on the stack when the function is called. Therefore it is important to ensure that the stack does not overflow, through recursive functions or large local variables.

8.7 Scope

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

8.7.1 Global Scope

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

```c
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.

8.7.2 Local Scope

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

8.7.3 Block Scope

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

8.7.4 Static Variables

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

Chapter 9

Types

9.1 Accessing Registers

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

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

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

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

9.2 Type Casting

Some type conversions are implicit, such as converting a `uint8_t` to a `uint16_t`. However, some implicit type conversions generate warnings usually because of 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;
```

This does not make code more portable, but it tells the compiler that the programmer is aware of the conversion and that it is intentional.

9.2.1 Types of Type Casting

Numeric Types

Numeric types (signed or unsigned) will expand or narrow the type, resulting in the value being truncated or zero extended.

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

Floating Point Types

Conversion from floating point to integer will truncate the fractional part.

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

Pointer Types

Conversion between pointers will change the pointer type but will not affect the data. Note that these conversions are not portable across platforms.

```
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 make the pointer contain the address in that integer.

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

Likewise, casting a pointer to an integer will make the integer contain the address in the pointer.

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

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

Modfying Qualifiers

Casting can also be used to add/remove qualifiers such as `const` and `volatile`, however the following will not work on the ATtiny1626 because the `const` qualifier usually results in that variable being stored in SRAM or read-only memory locations.

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

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

Avoiding Truncation

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

```c
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
```

9.3 Floating Point Types

In C, 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 $-2^{127} \dots 2^{127}$. A floating point number f can be represented 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. 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`
- $e = 0 \implies 2^{-126}$ (denormalised):
  - $m = 0$ (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 AVR does not have an FPU, floating point operations must be handled using the ALU instructions which can be significantly slower than integer operations. In addition, floating point operations require the `avr-libc` floating point library to be linked which increases the size of the program.

9.3.1 Fixed Point Math

Fixed point math 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.

Chapter 10

Objects

10.1 Structures

Structures are a way 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 initialized 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
```

10.1.1 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.

10.1.2 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`.

10.1.3 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 } };
```

10.1.4 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.

10.1.5 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 possible to use the struct name to declare variables.

```c
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 the amount of typing.

10.2 Unions

Unions are similar to structures, however the members of a union share the same overlapping memory location. While structs have capacity to store multiple values, unions only have the capacity to store its largest value.

```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 use allows us to access the same memory location interpretted as a different type without the need of casting pointers. When used with structs (or other aggregates), the order 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' }
```

10.3 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 `x` and `y` members are 4 bits each. The base type of each member must be able to store the specified number of bits.

10.3.1 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, this will 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
```

Chapter 11

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.

11.1 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.

11.1.1 Interrupt Vectors

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

11.1.2 Interrupt Service Routine

The code that handles the interrupt is called the **interrupt service routine** (ISR) (or *interrupt handler*). The ISR is a function that is executed as a result of the 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.

11.1.3 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 cleared by writing a 1 to the corresponding bit.

11.1.4 Peripheral Interrupts

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

Port Interrupts

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

```
ISR(PORTA_PORT_vect) {
    // Interrupt service routine for PORTA
    VPORTA.INTFLAGS = PIN4_bm;
}

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

11.1.5 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 with multithreaded programming. 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.

Chapter 12

Compilation

C source code is translated into machine code through a **compiler**, whereas assembly code is translated into machine code through an **assembler**. While some compilers emit assembly code, others emit machine code directly.

12.1 Assembler

Program code run directly on CPUs is not designed to be written by humans. The assembler prioritises performance and size efficiency, and accounts for simplified chip design.
They allow us to write programs in plain text without needing to memorise opcodes, or manually keep track of memory locations. Modern assemblers also provide features such as macros that assist in writing code. While the scope of an assembler is limited, the programmer decides exactly which instructions are used and the assembler simply translates them into machine code.

12.2 Compiler

A compiler is a program that translates source code written in a high-level language such as C. In a high level programming language, the desired program is described algorithmically by the programmer and the compiler produces an equivalent program which results in the same **side effects** when executed.

12.2.1 Compilation Process

Compilers can be configured to produce code in various ways. Compiling without optimisation will produce code which closely resembles the source code, so that it is easier to debug. This also means that code generation is faster. Compilers can also be configured to optimise code for minimum code size or maximum speed (or a combination of both).

12.2.2 Advantages of Compilers

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. Compilers also allow us to write efficient code through the use of compiler optimisations, which can be difficult to achieve manually in assembly code.

12.2.3 Disadvantages of Compilers

Some disadvantages of compilers are that hardware-specific features may only be available from assembly requiring the use of inline assembly or assembly language macros. Precise timing of code is also difficult due to the inability to predict how the compiler generates code.

12.3 Object Files

An object file is the output from the compilation or assembler phase. Object files mostly contain machine code, and also contain 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 external references unresolved. This allows the linker to resolve the references and produce a single executable file.

12.4 Linker

The linker is a program that combines multiple object files and links them together to produce a single executable file. The linker resolves all external references and addresses are updated as required. The linking step is extremely fast compared to

the compilation step, as only addresses need to be updated. The linker also performs 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.

12.4.1 Linker in Assembly

```
// 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.

12.4.2 Linker in C

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.

12.5 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 (names can be anything):

```
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 use the following blocking functions to read and write characters:

```
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.

Chapter 13

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.

13.1 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.

13.2 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, generate periodic signals on a pin that are frequency of pulse width modulated etc.

13.2.1 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, changing pin state etc.

13.2.2 Timer Counters

The ATtiny1626 has two timers, Timer Counter A and B (TCA/TCB) that are both 16-bit counters. As both timers are highly configurable, the datasheet should be consulted for more information.
In general, the counter, `CNT`, increments by 1 each clock cycle. The clock cycle can be configured with a prescaler to increase the duration of a timer.
The capture compare register `CCMP` can be used to generate an interrupt when the counter reaches a certain value.
The period register `PER` can also be used to set the maximum value of the counter. This register can also generate interrupts when the counter reaches the maximum value (overflow).

13.2.3 Timer Periods

The period T is the duration of a timer cycle, that is, the time before the counter resets to 0. To configure the timer counter to cycle according to this period, we must configure the register `PER`, which is the **number of counts** n of the timer clock, depending on the **clock frequency** f_{clk} of the timer clock.
One period of the timer clock is given by:

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

so that the number of timer clock cycles in a period T is given by:

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

The prescalared used to configure the timer clock leads to a trade-off between the timer period and the timer resolution, as the interval between timer clock cycles T_{clk} is increased.

13.2.4 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 period to some value
    TCB0.INTCTRL = TCB_CAPT_bm;      // Invoke the CAPT ISR when the counter reaches CCMP
    TCB0.CTRLA = TCB_ENABLE_bm;      // Enable TCB0
}

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

    while (1);
}
```

13.3 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% means that the signal net is always LOW, while a duty cycle of 100% means that the signal net 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.

13.3.1 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:

$$D_{\text{PWM}} = \frac{\texttt{TCA0.CCMP}}{\texttt{TCA0.CNT} + 1}$$

13.3.2 PWM Brightness Control Example

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

void tca_init() {
```

```
    // DISP EN
    PORTB.DIR = 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)
{
    sei();
    tca_init();
    cli();

    while (1);
}
```

To dynamically change the brightness of the display, the compare value can be changed using the buffered register `TCA0.SINGLE.CMP1`. The same applies to the period.

13.4 Analog to Digital Conversion

Analog 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.

13.4.1 Quantisation

When we **discretise in amplitude**, this is referred to as **quantisation**.
Each amplitude is assigned a digital **code**. The **code width** determines the **amplitude resolution** and introduces **quantisation error**.

13.4.2 Sampling

When we **discretise in time**, this 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.

13.4.3 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.

13.4.4 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 @ 3.3 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;
```

```
18
19      // Enable ADC
20      ADC0.CTRLA = ADC_ENABLE_bm;
21  }
22
23  int main(void)
24  {
25      sei();
26      adc_init();
27      cli();
28
29      while (1)
30      {
31          printf("%u\n", ADC0.RESULT0);
32      }
33  }
```

13.5 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** used (how the data is arranged, timing etc.) and the **physical interface** used (i.e., voltage level used to represent bits). For two devices to communicate, they must both use the same protocol and physical interface.

13.5.1 Serial Communication Terminology

- **Transmit**: to send data. Often abbreviated to **Tx**.
- **Receive**: to receive data. Often abbreviated to **Rx**.
- **Full-duplex**: bidirectional communication, occurring simultaneously.
- **Half-duplex**: bidirectional communication, occurring in one direction at a time.
- **Simplex**: Unidirectional communication.
- **Synchronous**: Communication relying on a shared clock.
- **Asynchronous**: Communication that does not rely on a shared clock.

There are many serial interfaces used in embedded systems, including:

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

13.5.2 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 Mbaud maximum).
UART is a frame based protocol, where each frame is signalled by a start bit (always LOW), and is fixed 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).

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.

USART0 on the ATtiny1626

The USART (Universal Synchronous/Asynchronous Receiver/Transmitter) peripheral is used to implement UART 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 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.

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
}
```

13.5.3 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.

SPI0 Example Configuration

```
void spi_init(void) {
    VPORTC.OUT = PIN0_bm | PIN2_bm; // Drive output to LOW on SPI CLK and SPI MOSI
    VPORTC.DIR = 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
}
```

13.5.4 Other Serial Protocols

- **I^{2}C**: Inter-integrated circuit.
  - Very common on microcontrollers, suitable for short distances only (typically $< 300\,\text{mm}$). Widely used for external peripherals.
  - Typically up to 400 kbaud.
  - 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 Mbaud
  - Half-duplex, asynchronous, 2-wire bus (one differential pair)
  - Very precise timing requirements compared with UART
  - Complex protocol and controller

13.5.5 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.

13.6 Serial Communications on the QUTy

13.6.1 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 (host, e.g., computer) side, 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 the 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 share the USB-UART bridge with the UART interface, which necessitates a switch to select between the two.

13.6.2 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 into the output register (and consequently update the state of Q0-Q7) requires a **rising edge** on the DISP LATCH net.

13.6.3 Time Multiplexing

As only one side of the 7-segment display can be illuminated at a time, we need to multiplex the display to show both digits. This is done by using a **time multiplexing** scheme, where we illuminate each digit for a short period of time, and then switch to the other digit. As this is done rapidly, 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.
If we choose a refresh rate of 50 Hz, we must choose a period of 20 ms for both digits. This means that we must switch digits every 10 ms which will ensure that each digit is displayed for an equal amount of time.

```
// Bytes latched onto 7-segment display
volatile int8_t left_byte = DISP_0 | DISP_LHS;
volatile int8_t right_byte = DISP_0;

// Current display side (alternates between left and right)
volatile uint8_t current_display_side = 0;

// 10ms interrupt
ISR(TCB0_INT_vect)
{
    if (current_display_side == 1)
        SPI0.DATA = left_byte;
    else
        SPI0.DATA = right_byte;

    // Toggle display side (flip lsb)
    current_display_side ^= 1;

    TCB0.INTFLAGS = TCB_CAPT_bm;
}

ISR(SPI0_INT_vect)
{
    // Latch the digit
    PORTA.OUTCLR = PIN1_bm; // Drive HIGH
    PORTA.OUTSET = PIN1_bm; // Drive back to LOW

    // Clear the interrupt flag (undocumented behaviour)
    SPI0.INTFLAGS = SPI_IF_bm;
}
```

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

13.7 Pushbutton Handling

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

- When the pushbutton is pressed, the pin is pulled low, corresponding to a `0` value.
- When the pushbutton is released, the pin is pulled high, corresponding to a `1` value.

The state of pushbuttons can be read through a bitwise AND operation 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.

```
uint8_t pb_prev = 0xFF;
uint8_t pb_state = 0xFF;

while (1)
{
    pb_prev = pb_state;
    pb_state = PORTA.IN;
    uint8_t pb_edge = pb_state ^ pb_prev;

    uint8_t pb_falling_edge = pb_edge & pb_prev;
    uint8_t pb_rising_edge = pb_edge & pb_state;
}
```

To additionally detect a falling/rising edge, we can use the AND operator (`&`) to detect a change in the signal.

13.7.1 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; what is acceptably small depends on what magnitude of latency is perceptible and is specific to the application. Latency as low as 2 ms is perceptible in particular user input applications, however latency betwen 20 ms to 60 ms is acceptable for most applications.

13.7.2 Switch Bounce

Switch bounce is an artefact of electromechanical switches where the switch contacts bounce back and forth when the 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 can **debounce** the pushbuttons and either by using a debouncing circuit, or software. To implement this in software, we can take multiple samples and only detect a change in state if the samples are consistent over multiple samples. Due to this, it is better to use an ISR to capture the state of the pushbutton at regular intervals.

```
volatile uint8_t pb_debounced_state = PIN4_bm;
volatile uint8_t pb_falling_edge = 0;
volatile uint8_t pb_rising_edge = 0;

// Periodic 4ms interrupt
ISR(TCB0_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)
        {
            // Save previous debounced state
            uint8_t pb_previous_state = pb_debounced_state;

            // Update debounced state
```

```
            pb_debounced_state = pb_sample;

            // Update falling/rising edge
            pb_falling_edge = pb_edge & pb_previous_state;
            pb_rising_edge = pb_edge & pb_debounced_state;

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

    TCB0.INTFLAGS = TCB_CAPT_bm;
}
```

The above implementation only debounces a single pushbutton, as the counter only applies to S1. To debounce multiple pushbuttons, it is possible to use a counter for each pushbutton, however this will lead to a large amount of code. Instead, we can utilise vertical counting.

13.7.3 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 will reduce the maximum value of the counter, however this is not a problem as we only require the counter to reach a value of 3. The following code implements a vertical counter for 4 pushbuttons.

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

// Periodic 4ms interrupt
ISR(TCB0_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;

    // Save previous debounced state
    uint8_t pb_previous_state = pb_debounced_state;

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

    // Update falling/rising edge
    pb_falling_edge = pb_edge & pb_previous_state;
    pb_rising_edge = pb_edge & pb_debounced_state;

    TCB0.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`. This variable updates if the state of the pushbutton is consistent over 3 samples, or if a falling edge is detected.

Chapter 14

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 can perform actions during transitions.
A state machine is fully defined by its list of states, initial state, and the conditions for transitioning between states.

14.1 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 type that allows 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.

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

// Initial state
state_t state = START;

while (1)
{
    // State machine
    switch (state)
    {
        case START:
            if (condition1)
                // Transition if condition is met
                state = STATE1;
            break;
        case STATE1:
            if (condition2)
                // Transition if condition is met
                state = STATE2;
            break;
        case STATE2:
            if (condition3)
                // Go back to start if condition is met
                state = START;
            break;
        default:
            // Invalid state, reset to start
            state = START;
            break;
```

```
    }
}
```

14.2 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
```

14.3 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.

Chapter 15

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.

15.1 Serial Protocol Design

15.1.1 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

15.1.2 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.

15.1.3 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.

15.1.4 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 desireable 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

15.1.5 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

15.1.6 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.

15.1.7 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

15.1.8 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.

15.1.9 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.

15.1.10 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 preferrable to split a large payload into multiple messages.

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.

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.

15.1.11 Escape Sequences

When there is potential ambiguity as to 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.
In a serial protocol, this may not appropriate as the backslash may be missed during transmission, and the next character may be treated as a start sequence for example. To address this, the escape sequence should not contain the symbol it is escaping. Instead, an alternate sequence is 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 also account for this.

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

  In this situation, the sender may retransmit the message, or send a different message.

15.1.13 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.
This simple checksum detects many transmission errors however, it is not guaranteed to handle symbols sent out of order, or symbols that are corrupted without changing the checksum.

15.1.14 Flow Control

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

15.2 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 the 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, and then use 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. The state machine can be implemented to consider the following:

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 (could be separate states for various identifiers)
5. **Checksum**: The parser is receiving the checksum