Chapter 4
Software Design

Design is not just what it looks like and feels like. Design is how it works.—Steve Jobs, co-founder of Apple Inc.

4.1 ES Software

Software is the program code that runs on a microprocessor or a microcontroller-based system to perform the desired tasks. Software needs to satisfy all objectives and specifications such as functionality, accuracy, stability, and I/O related operations. Embedded system composes of both hardware and software. Thus, an ES designer must also be adept on software design. However, unlike typical software programming on computers which is done in isolation of hardware consideration, embedded software design must be done with consideration of hardware. As Dr. Edward A. Lee of Berkeley summarized: "*Embedded software is software integrated with physical processes. The technical problem is managing time and concurrency in computational systems.*"

A good ES designer will take advantage of the hardware resources available in the system and must meet hardware constraints such as real-time response with guaranteed timing latency. Figure 4.1 shows an example of an embedded system of a temperature sensing system where some portion of the information processing blocks are implemented in hardware (shown in gray rectangles) and the rest are implanted in software code (shown in white ellipses). Dr. Edward A. Lee identified that embedded software is software integrated with physical processes. The technical problem is managing time and concurrency in computational systems [1].

ES software design should be developed with not only functionality in mind, rather also consider quality of programming, efficiency and compactness of codes, and optimization both in software domain and hardware domain. There are two major categories of performance measures for software: qualitative and quantitative. Dynamic efficiency is a measure of how fast a program executes, and can be measured with number of CPU cycles or time in seconds. Static efficiency is the number of memory bytes required for a software code, which is measured in terms of global variables, stack space, fixed constraints, and program object codes.

B. I. Morshed, *Embedded Systems – A Hardware-Software Co-Design Approach*,
https://doi.org/10.1007/978-3-030-66808-2_4

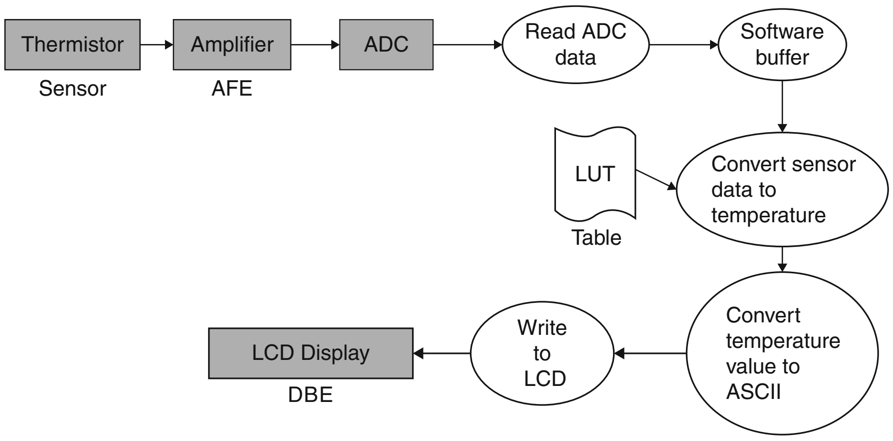


Fig. 4.1 An example of temperature sensing system with hardware (shown in gray) and software (shown in white) blocks

Today's ES are becoming complex, connected, and optimized. Consequently, software is also becoming complex with exponentially increasing complexity of the system. Modern ES systems can have more than 70% of the development cost for complex systems, such as automotive electronics, aviation, phones and other communication systems, are due to software development [2]. Thus, it is out-of-scope here to discuss all of the software choices, languages, and syntaxes for various software that can be used in ES design. Rather, here we briefly indicate some common programming language options and later provide some examples of C programming language with Arduino as the target platform. Readers must refer to relevant programming language books to learn these languages and syntaxes in details.

4.1.1 C Programming Language

C programming is one of the oldest programming languages that is still very popular for ES design due to its low-level hardware access capabilities, fast operation, and ability to run independently. This can be considered as Mid-level programming language. C program is converted to low-level assembly language with a compiler, then converted to machine code (i.e., binary file) with an assembler, along with help of linker if libraries are used. This machine code is downloaded on the MCU for execution of the program. As the machine code can run with or without operating system (OS), this is one of the best choices still for low-power ES design such as wearables or IoTs.

C coding is typically done with the MCU manufacturer provide or supported Integrated Development Environment (IDE). The file that contains the C code is called the source file, which is a text file. With the IDE integrated compiler and assembler, the source file is converted to assembly file, which is linked with other required precompiled files to an executable binary file. To download this binary file (hex) in an embedded processor, a programming chip is needed which might be integrated with the prototype board such as in Arduino or be an external device like In-circuit Debugger (ICD) of PIC microcontroller boards. The final hex file is downloaded in the microcontroller flash memory (which is non-volatile) and can be executed independently of operating system.

Although C program generates assembly codes, and ES programmer can also work on assembly code itself, however, this approach is becoming less popular with increasing complexity of codes. In cases of highly optimality need, such as execution clock cycle or memory usage, assembly code can still be useful to utilize. Most low-power embedded system, like Arduino, uses C programming. Thus, we will primarily focus on C here.

4.1.2 Java or Other High-Level Programming Language

Java is a high-level programming language. Many operating system (such as Android) support Java codes. It is a suitable programming language where the ES uses OS services such as file management, scheduling, and resource allocation. There are many other options for high-level programming languages such as Python, Perl, etc. As these programming languages are not efficient in implemented assembly code, they are not the best choice for efficient implementation. However, for complex codes such as smart devices with machine learning or deep learning techniques, these high-level programming language options are easier to implement. Nonetheless, ultimately these codes in high-level languages also are converted to assembly and machine code, which runs on the microcontroller hardware. Here, we will not focus on these high-level languages, but keep our discussion within C language, suitable for Arduino kit microcontrollers (such as Atmel MCU).

4.1.3 Pseudo-code

Pseudo-code is non-compile-able text representation of the code. It does not follow any exact syntax, and is meant to capture the imagination of the programmer to organize thoughts in common words. It is better suited when the software is more sequential and involves complex mathematical calculations. An example of pseudo-code:

```
Start
Read Name, Age
IF Age = -999 THEN
```

```
    Write "Error Empty File"
  ELSE
    DOWHILE Name <> -999
      Write Name, Age
      Read Name, Age
    ENDDO
  ENDIF
  Stop
```

4.1.4 Flow-chart

Flow-chart is another intuitively understandable representation of the overview of the code. It also does not have any syntax, but typically follows some shape convention for different activities. It is good option when the software involves complex algorithm with many decision points causing control paths.

4.1.5 Assembly Programming Language

Assembly is a low-level programming language. It most closely represents machine code. Typically, each assembly code represents one or a few microprocessor clock cycles, which can be conclusively found from Instruction Set Architecture (ISA) manual. It is also the most efficient implementation of code, but becomes unmanageably difficult as the program becomes complex. Although historically MCUs were programmed with assembly language, newer MCU with complex code requirement and highly-efficient assembler that converts C code to smallest assembly code, use of assembly code is becoming rare. Sometimes C integrated assembly instructions are used. Furthermore, as assembly code is hardware specific, portability becomes a major challenge.

4.1.6 Quality of Programming

There are two major categories for quality considerations of programming: Quantitative performance measurements and Qualitative performance measurements. Quantitative performance measurement can be two types: Dynamic efficiency and Static efficiency. Dynamic efficiency is a measure of how fast a program can execute. It is typically measured in terms of seconds or processor clock cycles. Higher dynamic efficient codes will execute faster or require lesser number of clock cycles. Static efficiency is the number of memory bytes required. It is measured in terms of global variables, stack space, fixed constraints, and program object code. Smaller memory byte usage is preferred in ES as these MCUs are severely memory constrained.

4.1.7 Documentation and Commenting

It is very important to have codes documented well with comments. It makes it easier to understand if the code needs to be revisited later or to be shared with fellow programmer. The code below shows a commented code in C programming language:

```
// square root of n with Newton-Raphson approximation
r = n / 2; // Initial approximation
while ( abs( r - (n/r) ) > t ) { // Iteration loop
  r = 0.5 * ( r + (n/r) ); // Implementation of equation
}
System.out.println( "r = " + r ); // Printing of output
```

A golden rule to remember about commenting in software development is that: "*Write software for others as you wish they would write for you.*"

4.1.8 Modular Software Design

One of the approaches for software development is modular design, where software development is divided into the software problem with distinct and independent modules. In this approach, the systems must be designed from components in modular fashion. A program module is a self-contained software task with clear entry and exit points. It must be easy to derive behavior from behavior of other subsystems. A module can be a collection of subroutines or functions that in its entirety performs a well-defined set of tasks. There are two requirement schemes: Specifications and Connections.

A critical issue in ES software is concurrency where multiple events are occurring in physical world which must be concurrently processed in ES. Modular software is a good mechanism to deal with this issue. Other major issues are synchronization and communication which can also be dealt with relatively easily with modular software design approach.

Some of the advantages of modular software design is easier to design, easier to change, and easier to verify. These are important for ES systems with complex nature. Communication of modules can take advantage of shared variables. This can be done with selecting the appropriate types of variables, such as private, public, local, global, static local, static global, and constant modifier.

4.1.9 Layered Software System

In another approach, software can be developed in a layered system. In layered approach, each layer of modules can call modules of the same or lower level, but not modules of higher level. This is a common approach used in operating system (OS). Here, the top layer is usually the main program layer or application layer. Other examples of layered architecture are Hardware abstraction layer (HAL), and Board-support package (BSP). In layered approach, a key concept used is abstraction, which is the process where data and programs are defined with a representation of similar to its meaning (semantics), while hiding away the implementation details.

4.2 ES Software Environments

Unlike traditional software in computers or servers which must have an OS to run the software, ES software can be deployed in environments with or without OS. Examples of OS-less software environment is Arduino (i.e., Atmel), ESP32, PIC, and PSoC. In fact, historic microcontrollers such as PAL, EEPROM, and FPGA, were always OS-less. With newer generation microprocessor-based micro-controller, there are many new options where an OS exists on the microcontroller that controls execution of the program. Examples of ES with OS are Raspberry Pi, Renesas Electronics Synergy, Beagle-Board, and NVidia Jetson.

4.2.1 MCU with OS

Operating system (OS) enables execution of multiple programs (using slots of processor times known as scheduling) and allocates system resources such as memory and peripherals. OS also handles access to peripherals and device drivers, as well as organizes storage such as file-system. It can also oversee access to other systems and manage communication protocols.

OS-based MCU are most suitable for complex program such as requiring audio/video data processing, machine or deep learning, concurrent processor intensive tasks, and programs requiring file systems. An advantage of OS-based style is that various OS related services and libraries can be available, thus making it easy to access or manage the ports and memory. In this style, control and management is centralized by OS. In case of multitasking and scheduling, OS makes software processes to cooperate and share the processor time. The OS checks periodically if other process requires action or time, and if priorities or deadlines of the processes are maintained. However, OS-based programming requires careful programming skills to achieve these, and can lead to "buggy" process which can stall system.

OS for ES is quite different than traditional computer OS (i.e., standard OS). For instance, many OS for ES do not support disk, network, keyboard, monitor, or mouse by default, rather these functionalities can be added as-per-need by integrating tasks or libraries instead of integrated drivers. Effectively most of these devices do not need to be supported in all variant of OS for ES, except maybe the system timer which is generally required in most OS. Another major difference in OS for ES is that device drivers, middleware, and application software—all might directly access kernel, rather than traditional layered software architecture described earlier.

Many embedded systems are real-time systems, hence the OS used in these systems must be real-time operating system (RTOS). This RTOS is an operating system that supports the construction of real-time systems. The key requirements for RTOS are the timing behavior of the OS must be predictable, and OS should manage the timing and scheduling. An example of RTOS is RT-linux, which is commercially available from fsmlabs (www.fsmlabs.com).

Real-time tasks cannot use standard OS calls, as standard OS calls do not have any mechanism for timing constraint assertion. RTOS classifies various real-time tasks as Critical, Dependent, and Flexible. Critical tasks are those that must meet timing constraints. If the constraints are not met, the system fails. Dependent tasks can allow occasional failures to meet time constraints, but it is good to meet these constraints in most cases. These do not result in system failure; however, subsystems might be impacted. Flexible tasks timing constraints are for improved performance, but missed timing constraints do not cause any failure.

RTOS also must allow real-time scheduling. Scheduling is finding a mapping or a scheme to execute the given tasks such that constraints are met. Typically, these schedules have to respect a number of constraints, including resource constraints, dependency constraints, and deadlines. For example, if a given task graph $G = (V,E)$ is given, a schedule τ of G will be a mapping such that $V \rightarrow D_t$, where V is a set of tasks to start times with domain D_t so that all constraints are met.

4.2.2 OS-less

ES designer must ask the question if OS is needed for the project at hand. It is not essential! Sometimes OS is not needed for the required tasks, especially if the project is relatively simpler. Almost always, OS-less systems will consume less power for the same task compared to OS-based systems, as OS overhead power consumption is not required and OS-less MCUs can be put to deep-sleep state when no activity is required.

One common and easy approach to develop OS-less software program is to use "endless loop." The general structure of this are as follows:

```
loop:
  do the first task;
  do the second task;
```

```
...
do the n-th task;
wait before next round;
repeat loop;
```

The loop code typically precedes with an initialization and setup code that runs once every time the system is restarted or reset. Arduino code library (Sketch) is one of this type of endless loop by default. This style of coding is every efficient and predictable as there is no OS overhead or uncertainty. However, it may not be able to handle time constraints of asynchronous events and might waste clock cycles for various tasks when those are not required. This coding style suits nicely when the tasks are repetitive in periodic manner such as monitoring a vehicle speed.

One of the issues with endless loop is that it is not reactive and runs continuously even if the tasks are not required to be performed. Most recent ES systems such as IoTs and smart devices are generally reactive ES. For this type, another style might be more suitable and more optimal, which is "endless loop with interrupts." This style permits immediate action (real-time) for time constrained events. Interrupt task is performed with software codes known as interrupt service routine (ISR) which triggers when interrupts are registered. ISR is typically atomic, and thus should be kept small in terms of execution time. If more than one type of event is possible within the ISR execution time when the code was atomic, the subsequent event might be missed. This style of coding requires the programmer to be aware of and do event scheduling and guarantee timing deadlines.

4.3 ES Software System Considerations

For development of ES software needs to consider some specific considerations which are not typical for standard computer software development. Key aspects for these considerations are briefly described in this section.

4.3.1 ES System Classification

In terms of task requirements and predictability, ES systems can be classified in 3 types:

1. ***Static:*** The needs of the tasks of these systems are fixed and known at design time. Examples of this type of ES are traffic light system, elevator system, and oven control unit.
2. ***Deterministic:*** The needs of the tasks have variable requirements, but the requirements are either known or at least can be predicted fairly well in advance at design time. Examples of this type of ES are temperature control unit for air condition, drone control unit, and handheld camera controller.

3. ***Dynamic:*** The needs of the tasks have also variable requirements, but there may be no way to know or accurately predict these needs in advance at design time. Examples of this type of ES are intelligent robots, self-driving cars, and airplane control unit.

4.3.2 Real-Time Operation

Real-time with regards to computer system is defined as those systems that update information at the same rate as they receive data, enabling them to direct or control a process. Examples of real-time systems are air condition system controller, elevator controller, self-driving cars, multimedia player, etc. For real-time ES, the response must be relevant when it is generated. According to J. Stankovic [3], real-time systems are "*those systems in which the correctness of the system depends not only the logical results of the computation, but also on the time at which the results are produced.*"

Real-time systems will have some tasks that are real-time constrained. These tasks can be Hard real-time or Soft real-time. Hard real-time tasks are bounded by latency. They must be completed within this bound or else the whole system will fail. An example of hard real-time is braking of self-driving car. If the braking is not complete within certain time (or distance), the car might crash. Another example of hard-real time is flying controller of a drone. If there is a sudden gust of wind, the controller must compensate quickly or else it might crash land.

Soft-real time tasks needs to be executed as soon as possible. The sooner the task completes, the better the experience or performance. However, there is not strict timing or deadline that will result in system failure. An example of this type of system is the elevator control unit. Then a floor button is pressed, it is desirable for the elevator to reach the target floor. However, if it is little late, it does not cause any failure of system. Another example of this type is air condition control unit. When the temperature is too high or too low, air cooling or heating unit will turn on. It is desirable to complete this cooling or heating process soon, but delay of this process is not catastrophic to the system.

In addition, there can be tasks with no real-time constraints. They do not have any dependency of time. Although these types of tasks are rare in ES as most ES are real-time reactive systems. Real-time system design must consider real-time constraint. At design time, latencies (inherent delay) can be considered for timing constraints, but it needs to ensure the system delays at runtime also fits within the timing constraints, especially if it is a hard-real time constraint.

4.3.3 Real-time Reactive Operation

Real-time systems require correctness of result as a function of time when it is delivered. They do not have to be "real fast." In fact, in most cases, consistency is more important than raw speed. These systems should be tested for worst case performance (rather than typical performance). Worst case performance often limits design.

Reactive systems are those whose computation rate is in response to external events. If the events are periodic, they can be scheduled statistically. For example, the temperature monitoring of an air condition control unit can check the temperature once every one minute and take response if the sensed temperature is above or below the temperature range set by the user. For aperiodic events, however, the system must need to be able to statistically predict and dynamically schedule when possible. This will avoid overdesign. Another approach, which is superior, is to design for interrupt to trigger when the aperiodic events occur. For example, an RFID based lock system should only react when an RFID tag is presented, rather than checking for RFID tag periodically.

4.3.4 Time Services

Time plays a critical role in real-time systems. All microcontrollers have timer or counter hardware that can be used for relative timings (or delays). However, actual time is described by real numbers and not available in microcontrollers (unlike computers). If any project requires the exact time (i.e., year, month, day, hour, minute, and second), a Real-Time Clock (RTC) module must be included with the hardware. RTC can be initialized to current time, and can continue keeping time service. Constant power is needed to keep RTC counting, otherwise it will reset. Coin cell batteries are commonly used to only power the RTC that can operate for years. Two discrete standards are used in RTC: International Atomic Time (TAI) and Universal Time Coordinated (UTC). Both are identical on Jan. 1st, 1958. UTC is defined by astronomical standards and is more commonly used in ES. Timing behavior is essential for embedded systems. According Lee et al. (2005), the lack of timing in the core abstraction of computer science, from the prospective of ES, is an inherent flaw. Timing has far-reaching consequences for ES design processes.

4.3.5 Timing Specifications

According to Burns (1990), there are four types of timing specifications related to ES:

1. ***Measure elapsed time:*** This metric indicates how much time elapsed since last call.
2. ***Means for delaying processes:*** This metric indicates a procedure to delay a process relative to another process.
3. ***Possibility to specify timeouts:*** This specification provides a maximum time for a process to stay in a certain state. If the state does not go through a transition to next state within the timeout specification, timeout transitions to a different state (e.g., reset).
4. ***Methods for specifying deadlines:*** This specifies by which time a process must complete. For some processes, it might not exist, or may be specified in a separate control file.

4.4 Advanced Arduino Programming Techniques

As mentioned before, Arduino uses C programming language and is an OS-less software environment. It can either be used in an endless loop or an endless loop with interrupt. In this section, we will explore these with some advanced techniques in the context of Arduino Uno board which has Atmal ATMega328 microcontroller.

4.4.1 Bitmath

Bitwise manipulation provides great strength to programmer when faces with severely processing power constrained systems or extremely high-speed system. Some of the prime advantages of bit manipulation are:

1. When you need multiple pins to be set as input or output exactly at the same time (in the resolution of clock rate), using port bit manipulation allows it. For example, setting *PORTB &= B110011* output HIGH value to pins 8, 9, 12, and 13 at exactly the same clock cycle. Alternatively, if you use *DigitalWrite(8, HIGH)*; *DigitalWrite(9, HIGH)*; *DigitalWrite(12, HIGH)*; *DigitalWrite (13, HIGH)*; will require 4 instructions with slightly different time (in terms of clock cycle) of outputs. For high-speed time-sensitive cases (such as digital parallel communication), this time discrepancy might cause issues.
2. For high-speed communication (serial or parallel), bit manipulation is also powerful. These instructions only take a single clock cycle (typically), whereas *DigitalWrite* or *DigitalRead* instructions might require tens of instructions (depending on the library). In high-speed communication where you need to transmit data or receive data very quickly, bit manipulation is essential. Thus, the communication driver libraries utilize bit manipulation.
3. As bit manipulation requires lesser number of instructions, it is also helpful to reduce instruction code size, especially when the system has a very low program

memory (e.g., tiny microcontrollers) or power budget (e.g., battery or solar powered).

4.4.2 Arduino Uno Digital and Analog I/O Ports

To utilize Arduino Uno interrupt and timers, it is critical to understand and distinguish the onboard microcontroller (ATMega328) and board (Arduino Uno) I/O pins. While Arduino Uno board has 14 digital I/O pins (labeled 0–13) and 6 analog Input pins (labeled 0–5), the microcontroller refers these pins in 3 ports: Port B, Port C, and Port D. These are connected as per table below:

Arduino board I/O pins	Microcontroller I/O pins
Digital pins 0 to 7	Port D [0:7]
Digital pins 8 to 13	Port B [0:5]
Analog pins 0 to 5	Port C [0:5]

Note that Digital pins 0 and 1 are internally connect to RX and TX for serial communication used for USB connection to the computer for downloading the program code and serial port monitor/plotter. If this port is needed for connecting other hardware (such as Bluetooth module or voice recognition module), they must be connected exclusively (i.e., there must be only one communication connection at a certain time). For instance, if a sound recognition module needs to be connected, then one must remove the voice recognition module connection to program the board from the computer with USB cable. Then remove the USB cable, and connect the voice recognition module. As the board is typically powered by USB cable also, thus when USB cable is not connected, the Arduino Uno board must be powered otherwise (e.g., with a battery or a power adapter) using the DC power port.

Another note is the digital pin 13 is also internally connected to an onboard LED. Thus, whenever digital pin 13 is used as an output pin, the LED will correlate with the data being outputted. Also note that microcontroller Port B [6:7] and Port C [6:7] are not connected to any I/O port of the Arduino Uno board (rather used internally).

4.4.3 Bitmath Commands for Arduino Uno Digital I/O Pins

To setup a digital Arduino Uno I/O pin to input or output mode, the command used in Arduino Sketch library is *pinMode(pin,mode)*. This command sets that corresponding pin to INPUT or OUTPUT mode. For example, *pinMode(13,OUTPUT);* sets the digital pin 13 of Arduino Uno to output mode. In fact, the *pinMode* command writes 1 bit in the DDRx register, where x is based on the pin number as given in the table above. Alternatively, using Bitmath approach, the same operation can be done with writing the DDRx directly. The command will be *DDRx = value*.

Here x can be B, C, or D, depending on the target pin number. Binary value of 0 will set that pin as input, and 1 will set that pin as output. Note that DDRx command will write the all pins of that entire register (unless masking method is used as described below). Example: *DDRB = B100000;* command sets the pin 13 as output mode, and pin 8–12 as input mode. Note that "B" (upper case) on the right-hand side denotes the number is provided in binary format. Another notation for binary format is "0b" (smaller case).

To output a logic 0 (low) or 1 (high) using the output pin, Arduino Sketch code is *digitalWrite(pin, value)*. This command sets the pin to LOW (0) or HIGH (1). Example, *digitalWrite(13, HIGH);* will output high logic through pin 13 (and the onboard LED connected to pin 13 will turn on). In fact, this command writes one bit in the PORTx register to LOW or HIGH. Here, x is B, C, or D depending on the pin number. The Bitmath equivalent of this command is *PORTx = value*. This command sets the port bits to High (1) or Low (0) based on the value, which is reflected at the output pins. For example, *PORTB = B100000;* will set the pin 13 to HIGH (and pins 8–12 to low if set as output mode). Again, we can use masking technique described below if we do not want all bits of that port to change.

To read the digital value of an INPUT type pin, the Arduino Sketch code is *int value = digitalRead(pin);* that reads the data into the variable value. This command reads back the pin value (0 or 1). Example, *int data = digitalRead(7);* reads pin value from pin 7 (assuming it is set as Input). The Bitmath equivalent of data reading is a bit more complex. The command is *int value = PINx;* where x is Port B, C, or D depending on the pin. However, this command (PINx) reads the entire port and sets it in value. Thus, bit masking and shifting is essential after reading the port to get the correct value. Example, *int data = PIND;* reads the entire 8-bits of Port D to the variable data.

4.4.4 Bitmath Masking Techniques

Bitmath masking operations are typically done with AND, OR, or XOR gates. AND gate mask is used to Reset (i.e., logic 0) a specific bit. OR gate mask is used to Set (i.e., logic 1) a specific bit. XOR gate mask is used to toggle (i.e., make logic 1 if value was logic 0, and vice versa) a specific bit. To explain these operations, it is important to recall these gates Truth tables:

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

AND gate truth table

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

OR gate truth table

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

XOR gate truth table

AND gate example operation:
Initial portd is B01010011
Operation: **portd &= B00110101;**
Final portd is B00010001
Explanation:

Initial	0 1 0 1 0 0 1 1
AND (&)	0 0 1 1 0 1 0 1
Result	0 0 0 1 0 0 0 1

In general: **x & 0 → 0** and **x & 1 → x**, where x is the initial value.
OR gate example operation:
Initial portd is B01010011
Operation: **portd |= B00110101;**
Final portd is B01110111
Explanation:

Initial	0 1 0 1 0 0 1 1
OR (|)	0 0 1 1 0 1 0 1
Result	0 1 1 1 0 1 1 1

In general: **x | 0 → x** and **x | 1 → 1**, where x is the initial value.
XOR gate example operation:
Initial portd is B01010011
Operation: **portd ^= B00110101;**
Final portd is B01100110
Explanation:

Initial	0 1 0 1 0 0 1 1
XOR (^)	0 0 1 1 0 1 0 1
Result	0 1 1 0 0 1 1 0

In general: **x ^ 0 → x** and **~x ^ 1 → ~x**, where x is the initial value and ~x represents NOT(x).

4.4.5 Port Masking with Bitmath Examples

For further clarification, some examples of port masking with Bitmath operation are given below:

Example 4.1 What is the output for the operation below:

```
portb = portb & B000011;
```

Where the initial value of portb is B101010.

```
Solution: Initial portb:      101010
Masking operator:        &    000011
                              ====
Final portb:                  000010
```

Explanation: This operation Reset the first 4 bits

Example 4.2 What is the output for the operation below:

```
portb = portb | B000011;
```

Where the initial value of portb is B101010.

```
Solution: Initial portb:      101010
Masking operator:        |    000011
                              ====
Final portb:                  101011
```

Explanation: This operation Set the last 2 bits

Note that the operation in example 4.1 can also be written as: *portb &= B000011;*. Similarly, the operation in example 4.2 can also be written as: *portb |= B000011;*.

4.4.6 Direct Port Access with Bitmath Examples

Here are some examples of direct port access with Bitmath beside the corresponding Arduino Sketch codes:

Arduino Sketch code	Bitmath equivalent code
pinMode(8,OUTPUT);	DDRB |= B000001;
digitalWrite(8,HIGH);	PORTB |= B000001;
pinMode(7,OUTPUT):	DDRD |= B100000000;
digitalWrite(7,LOW);	PORTD &= B011111111;
pinMode(9,INPUT);	DDRB &= B111101;
int val = digitalRead(9);	int val = (PINB & B000010) >> 1;

Note that the last code in this table in Bitmath column is performing a masking operation of the second bit (corresponding to digital pin 9), and subsequently shifting the value by 1 bit to the right to align that masked bit to LSB.

Table below gives more example for a partial code of Arduino:

Arduino Sketch code	Bitmath equivalent code

void setup () {
. . .
pinMode(5,OUTPUT);
pinMode(6,OUTPUT);
pinMode(7,OUTPUT);
. . .
}
void loop () {
. . .
digitalWrite(5,HIGH);
digitalWrite(6,HIGH);
digitalWrite(7,HIGH);
. . .
digitalWrite(5,LOW);
digitalWrite(6,LOW);
digitalWrite(7,LOW);
. . .
}
	void setup () {
	. . .
	ddrd |= B11100000;
	. . .
	}
	void loop () {
	. . .
	portd |= B11100000;
	. . .
	portd &= B00011111;
	. . .
	}

From the code snippet above, it is clear that the Bitmath can lead to a much smaller and compact code compared to Arduino Sketch code. Furthermore, each Arduino Sketch code statement might take several clock cycles to complete, whereas each Bitmath code statement takes only one clock cycle to complete. Thus, Bitmath operations, although more complicated to understand and use, leads to more static efficient code (requiring less memory) and more dynamic efficient code (requiring less clock cycles) for the exact same operation.

4.4.7 Advantages of Direct Port Access

Major advantages of using direct port access with Bitmath operations are:

1. ***Execution time:*** Turn pins ON or OFF very quickly (within the same clock cycle if they are in the same port or within a fraction of microseconds if different ports). On the other hand, *digitalRead()* and *digitalWrite()* get compiled into quite a few machine instructions. Each machine instruction requires one clock cycle, which can add up in time-sensitive applications. Direct port access can do the same job in a lot fewer clock cycles.
2. ***Concurrent operation:*** Bitmath operation can Set/Reset multiple output pins at exactly the same time (same clock cycle) when they are on the same port. Calling *digitalWrite(10,HIGH);* followed by *digitalWrite(11,HIGH);* will cause pin 10 to go HIGH several microseconds before pin 11. On the other hand, you can set both pins high at exactly the same moment in time using *PORTB |= B001100;* command.

3. ***Compact code size:*** Bitmath can make your code size smaller. It requires a lot fewer bytes of compiled code to simultaneously write via the port registers than to set each pin separately.

4.5 Dealing with Data Port Queue

Data port queue can be implemented using buffers. The purpose of buffer is to create a “spool” or a “queue” to allow the software routine to operate independently. There are two levels of buffering approach:

1. ***Hardware level buffer:*** This is typically a small (a few characters at most) buffer that is used directly by the subsystem to immediately obtain the data it demands or to immediately store the data it generates.
2. ***Software level buffer:*** This is typically a large buffer used when servicing the subsystem and is also used by the processing routine (main) to interact indirectly with the hardware

   Queueing mechanism is shown in Fig. 4.2.

4.5.1 Polling

The regular monitoring of a number of different service flag bits may be done through a technique called Polling. The polling technique consists of a loop which reads service flags and checks them one by one to see if they are active. Most peripherals provide *service flags,* which reflect I/O status conditions. Typically, these service flag bits get set automatically by the peripheral to indicate that a service routine needs to be launched. The service flags therefore need to be checked regularly to ensure timely service for some I/O task.

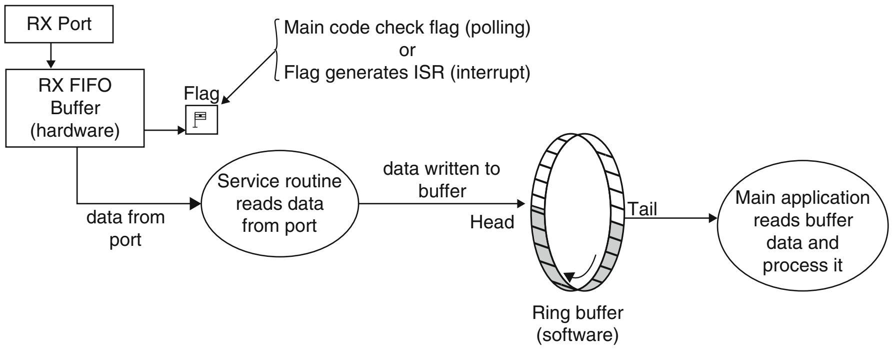

RX Port
RX FIFO Buffer (hardware)
Flag
Main code check flag (polling)
or
Flag generates ISR (interrupt)
data from port
Service routine reads data from port
data written to buffer
Head
Tail
Ring buffer (software)
Main application reads buffer data and process it

Fig. 4.2 Queueing mechanism for data access from MCU I/O port

The programmer needs to consider two things when service flags are used to launch service routines:

1. ***Check the service flag bit regularly:*** This involves periodic checking of the flag bit to see if new data is available at the port queue. Programmer needs to decide at the design time on how often the service flag needs to be checked. If the service flag become active for a short amount of time before new data arrive, the code might miss the first service flag and data loss might occur. If the service flag checking is too fast but data arrives after long intervals, then the clock cycles required to check the service flags are wasted overhead.
2. ***Provide a specific service routine to serve the data:*** When the service flag check finds the service flag bit active, it needs to quickly begin a service routine to serve the data (i.e., collect data from the port and move it to memory buffer or process data immediately). However, the delay might be unpredictable during design time as the exact time of arrival for data and time of checking for data is not known although bounded. Thus, the programmer needs to allow this uncertainty in the code. Also, the programmer needs to think in implementing priorities if multiple ports to be checked so that higher priority data are served appropriately within any timing deadline.

Thus, the polling technique has the following disadvantages:

- The latency interval between the time the service flag gets set and the time the program detects the active service flag depends on how busy the program is when the service request is made by the service flag.
- A large amount of processing time will be wasted polling service flags which are inactive.

4.5.2 Interrupt

An interrupt is an automated method of software execution in response to hardware event that is asynchronous with the current software execution. It allows program to respond to events when they occur and allows program to ignore events until that occurs. Interrupt technique is useful for external events, e.g., UART ready with/for next character, or signal change on pin. The action of the interrupt depends on context as programmed by Interrupt Service Routine (ISR). It can also be used to monitor number of edges (signal value change) arrived on pin. Interrupt can also be generated for internal events, e.g., power failure, arithmetic exception, or timer "tick" or overflow.

When an electronic signal causes an interrupt, the intermediate results have to be saved before the software responsible for handling the interrupt can run. This saving (and restoring process) takes some time, albeit small. The time delay incurred between the instant when the subsystem needs service to the instant the service routine actually provides the service is called *Interrupt Latency time*. Hardware buffer allows the system to cope with the "latency time" by providing a temporary storage. Key factors affecting latency are current CPU critical task, interrupt request processing, CPU registers that need to be saved in the stack, and initialization codes to begin servicing.

Some advantages of interrupts are as followings:

- Since the reaction to the service request is hardware based, there is no need to waste precious processing time polling inactive service flags.
- The latency interval to launch the interrupt handler will be very short when compared to polling.
- It will be easier to establish the priority structure which handles different service requests.

Some disadvantages of interrupt are as follows:

- The program necessary to control and interrupt system is much more complex than a polled system
- Hardware has to be added in order to connect the peripheral service flags to interrupt circuitry.
- Costs may be higher since the peripherals and the processor need additional hardware resources to support interrupt circuitry.

4.5.3 Software Buffer

Often individual data from ports needs to be placed to buffers when individual processing of data is not possible. For instance, to decode an audio signal, individual data samples cannot be processed. Similarly, to detect disease from ECG (electrocardiogram) data, it must be processed with at least one complete heartbeat representing ECG waveform. In some communication setup, especially wireless communication, data should be transmitted in chunks or packets, rather than individual data, as otherwise communication overhead becomes impractical.

For these scenarios, a software buffer as shown in Fig. 4.2 can be implemented. The buffer shown in this figure is "Ring buffer" or circular buffer. Another software buffer option is "Mutex buffer."

4.5.3.1 Ring Buffer

A schematic representation of ring buffer is shown in Fig. 4.3. The ring buffer has a write position (called Head) and a read position (called Tail). The data between Head and Tail are unprocessed data, and rest of the space is available. When new data is pulled from port, it will be placed at Head, and the Head position will be incremented. Similarly, when data needs to be processed, one or more data will be read from Tail position, and the new Tail position will be adjusted accordingly.

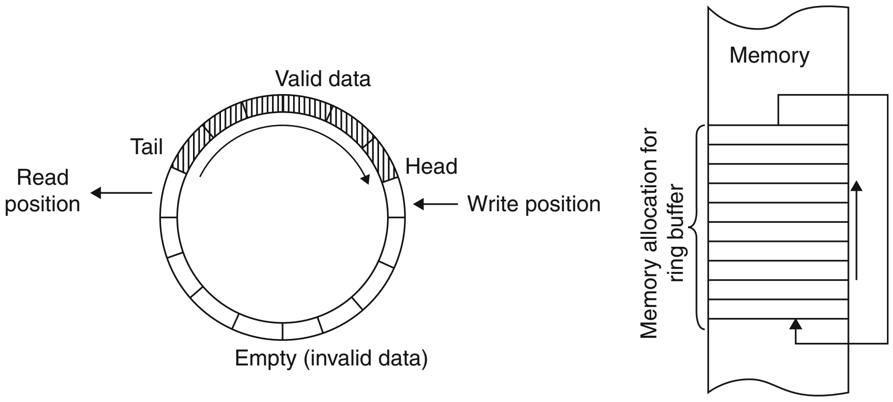


Fig. 4.3 Ring (or circular) buffer representation and implementation

There are a few constraints that needs to be met for Ring buffer, such as there must be available space for new data write (i.e., to increment Head position). As the Ring buffer will be implemented in memory with an allocation of memory, when Head (write process) reaches the top of the allocated memory, it must *Wraparound*, and begin from the bottom of the allocated memory. Similar wraparound must also happen to Tail when ready process reaches the top. Conditional checks must be performed to implement these.

4.5.3.2 Mutex Buffer

Although ring buffer is the most optimized software buffer, it is complex to implement. An easier solution is to employ Mutex buffer approach. Figure 4.4 shows a schematic representation of this approach. In this approach, a multiple of same size buffers are allocated. For instance, if 2-buffer space is used (A and B), the port will start writing data to one of them, say A. When A buffer is full, it can be used by the main code for processing. However, in the time A buffer is being processed, new data can still come in through input port. Thus, the new data needs to be written in buffer B. Again, when buffer B is full, the new data will be written in buffer A while buffer B data is being processed. This also needs to satisfy some constraints such as buffer size has to large enough to allow processing of data from the other buffer. More than 2 buffers can also be used in this approach.

4.5.4 Direct Memory Access (DMA)

Direct Memory Access (DMA) is useful to move large chunk of data to or from I/O port system from or to memory without involving MCU (except initialization).

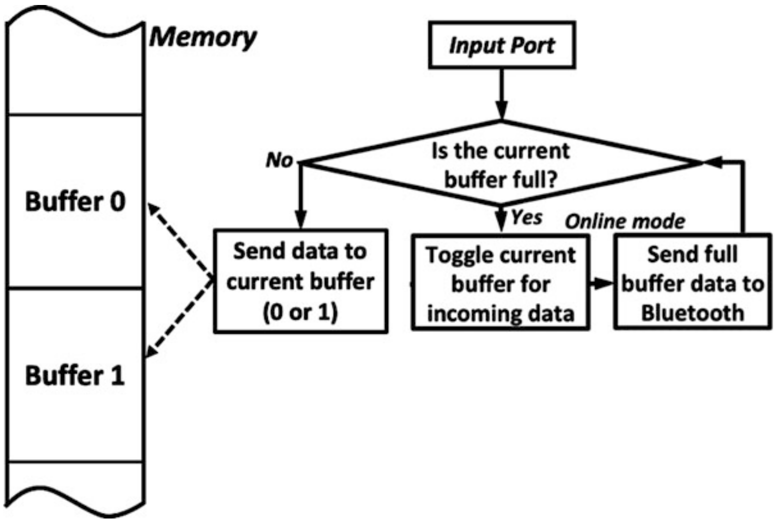

Memory
Buffer 0
Buffer 1
Input Port
Is the current buffer full?
No
Yes
Online mode
Send data to current buffer (0 or 1)
Toggle current buffer for incoming data
Send full buffer data to Bluetooth

Fig. 4.4 Mutex buffer representation with 2-buffer space (A and B)

DMA controller is a hardware device that can control DMA data transfer without processor intervention. However, not all MCU contains DMA controller hardware. For example, the Arduino Uno microcontroller (i.e., ATmega328) does not have DMA hardware. But, Arduino Due MCU has DMA hardware. Most higher-end MCUs contain DMA controller hardware, such as MSP430 series MCUs, STMicroelectronics MCUs, and PSoC MCUs.

To setup DMA data transfer, the processor (MCU) needs to provide the following information to DMA controller:

- Beginning address in memory,
- Block length (i.e., number of Bytes to transfer),
- Direction (memory-to-port or port-to-memory),
- Port ID,
- End of block action (issue interrupt or do nothing).

After this setup process, DMA controller will begin data transfer, and MCU can perform other tasks or sleep, as per code. At the completion of DMA block transfer, DMA controller can raise an interrupt (wake up MCU) if it is set that way, or it can change corresponding status register to indicate completion of the data transfer task.

Example 4.3 Write a C code for transferring a block of ADC data to memory using DMA controller of Arduino Due using 4 mutex buffers.

Solution A code is provided below, where ADC data structure is assumed to consists of ADC_MR, ADC_CHER, ADC_IDR, ADC_IER, ADC_RPR, ADC_RCR, ADC_PNPR, ADC_RNCR, ADC_PTCR, ADC_CR, and ADC_ISR.

```
// For USB (ADC->DMA->Memory buffer; if full->USB)
#undef HID_ENABLED
// Input: Analog in A0
// Output: Raw stream of uint16_t in range 0-4095
// on Native USB Serial/ACM
volatile int bufn,obufn;
uint16_t buf[4][256];  // 4 buffers of 256 readings
void ADC_Handler(){   // move DMA pointers to next buffer
 int f=ADC->ADC_ISR; // Status register, Rx buffer (28)
 if (f&(1<<27)){ // Masking only 28th bit
  bufn=(bufn+1)&3; // Buffer sequence: 0->1->2->3->0 etc.
  ADC->ADC_RNPR=(uint32_t)buf[bufn];
// Data from ADC to new Memory buffer
  ADC->ADC_RNCR=256; // Size
 }
}
void setup(){
 SerialUSB.begin(0);
 while(!SerialUSB); // Wait for USB ready
 pmc_enable_periph_clk(ID_ADC); // Enable peripheral clock
 // Initialize internal ADC module
 adc_init(ADC, SystemCoreClock, ADC_FREQ_MAX, ADC_STARTUP_FAST);
 ADC->ADC_MR |=0x80; // Mode for free running, no trigger
 ADC->ADC_CHER=0x80; // Enable Channel 7 of ADC
 NVIC_EnableIRQ(ADC_IRQn); // Enable interrupt
 // Disable all interrupt except channel 7
 ADC->ADC_IDR=~(1<<27);
 ADC->ADC_IER=1<<27; // Interrupt enable for channel 7
 // Initialize DMA buffer to begin with
 ADC->ADC_RPR=(uint32_t)buf[0];
 ADC->ADC_RCR=256; // Size of data
 // Initialize next DMA buffer to use
 ADC->ADC_RNPR=(uint32_t)buf[1];
 ADC->ADC_RNCR=256; // Size of data
 bufn=obufn=1; // Set current and old buffer as same
 // bufn: next buffer; obufn: overflow of next buffer
 ADC->ADC_PTCR=1;
 ADC->ADC_CR=2; // Control register to start DMA
 // write HIGH to bit-2 to start DMA operation
}
void loop(){
 while(obufn==bufn); // wait for buffer to be full
// send buffer data - 512 bytes = 256 uint16_t
 SerialUSB.write((uint8_t *)buf[obufn],512);
 obufn=(obufn+1)&3;   // select the next buffer
}
```

Example 4.4 Write a C code for reading ADC data using DMA to store in a mutex buffer (2 buffers each of 1024 Bytes) and transfer entire buffer data from memory to serial port (Bluetooth) using DMA when buffer is full for a MSP430 MCU.

Solution A code is provided below.

```
   // MSP430 uC; Read process from ADC12 using DMA
   if (BF1_USE==1) // Which buffer to write data using DMA
    {
   //src
    __data16_write_addr((unsigned short) &DMA0SA, &ADC12MEM0 );   //
dest
    __data16_write_addr((unsigned short) &DMA0DA, &Buffer2[0]);
     BF1_USE=0;
     BF1=1;
    } else {
   //src
   __data16_write_addr((unsigned short) &DMA0SA, &ADC12MEM0 );
   //dest
   __data16_write_addr((unsigned short) &DMA0DA, &Buffer1[0]);
    BF1_USE=1;
    BF2=1;
   }
   // BF1 = 1 -> Buf1 is full;
   // BF2 = 1 -> Buf2 is full;
   // x value increments until all done
    if (BF1==1) {
     UCB0TXBUF = *((unsigned char*)Buffer1+(x-1));
     x++;
     if (x==(1025)) {
      x=0;
     BF1=0; // Buf1 empty, transmission complete
      }
     } else if (BF2==1) {
     UCB0TXBUF = *((unsigned char*)Buffer2+(x-1));
     x++;
     if (x==(1025)) {
       x=0;
      BF2=0;  // Buf2 empty, transmission complete
      }
   }
```

4.6 Using Interrupts

To use interrupts, the peripheral must be instantiated with hardware capability in order to generate an interrupt request. Each peripheral is assigned its own interrupt request number. The initialization software for that peripheral must set the appropriate bits in the peripheral IRQ enable register. Hardware lines (IRQn) bring the different peripheral interrupt requests into the CPU where they are masked (using AND gates) by the iEnable control register. In order for a particular interrupt request to get through, the appropriate iEnable bit must be set by software when the system is initialized. Interrupt requests that get through the AND gates, apply 1's to the corresponding bit in the iPending control register. The iPending bits are then ORed together to generate a processor interrupt request which may be masked by the PIE bit located in the status control register. The PIE bit must be set by software when the system is initialized.

4.6.1 Interrupt Vectors

Interrupts use pre-programmed (ROM) Interrupt Vectors associated with the microcontroller. Interrupt Vector is a table in memory containing the first instruction of each interrupt handler. These are predefined interrupt routines to initiate recommended ISR functions. Programmer can also write custom ISR functions. If interrupts are not used, this memory can be used as part of the program. An example interrupt routine is RESET, which Sets up the stack pointer.

Interrupt exception routines service any interrupt. Before servicing the interrupt, all register values are saved in the stack. Then, the service is done through calling the appropriate Interrupt Handler. A function code called Interrupt service routine (ISR) starts to complete the service. After service is done, all registers are restored from the stack.

4.6.2 What Happens when Interrupt Event Occurs?

When an interrupt event occurs, the following operations takes place in sequence:

1. Processor does an automatic procedure call,
2. CALL automatically done to address for that interrupt,
3. Push current PC, Jump to interrupt address as each event has its own interrupt address,
4. The global interrupt enable bit (in SREG) is automatically cleared, i.e., nested interrupts are disabled, and
5. SREG bit can be set to enable nested interrupts if desired.
6. Interrupt procedure, aka "interrupt handler" or Interrupt Service Routine (ISR) starts,
7. ISR executes the function written inside the function,
8. ISR then returns via RETI,
9. The global interrupt enable bit is automatically set on RETI,
10. One program instruction is always executed after RETI.

4.6.3 ISR Coding Norms

Interrupt Service Routine (ISR) or interrupt handler should be written such that it is invisible to program, except through side-effects, e. g. via flags or variables. ISR will change the normal program execution timing (as ISR codes will need to be executed). Thus, interrupt-based program cannot rely on "dead-reckoning" using instruction timing only.

ISR needs to be written so they are invisible. For instance, ISR cannot stomp on program state, e. g. registers, or save and restore any registers used in the program. ISR code should be small and fast. So, do not put codes that takes long time to complete (such as serial port communication commands). It is also ideal to make ISR code "Atomic" to avoid firing of another ISR while one ISR is being executed.

4.6.4 List of Interrupts and ISRs of ATmega328

ATmega328 has a number of interrupts as shown below (taken from its datasheet [4]).

VectorNo.	Program address	Source	Interrupt definition
1	0×0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0×0002	INT0	External Interrupt Request 0
3	0×0004	INT1	External Interrupt Request 1
4	0×0006	PCINT0	Pin Change Interrupt Request 0
5	0×0008	PCINT1	Pin Change Interrupt Request 1
6	0×000A	PCINT2	Pm Change Interrupt Request 2
7	0×000C	WDT	Watchdog Time-out Interrupt
8	0×000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0×0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0×0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0×0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0×0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0×0018	TIMER1 COMP8	Timer/Coutner1 Compare Match B
14	0×001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0×001C	TIMER0 COMPA	Timer/Counter0 Compare Match A

(continued)

VectorNo.	Program address	Source	Interrupt definition
16	0×001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0×0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0×0022	SPI, STC	SPI Serial Transfer Complete
19	0×0024	USART, RX	USART Rx Complete
20	0×0026	USART, UDRE	USART, Data Register Empty
21	0×0028	USART, TX	USART, Tx Complete
22	0×002A	ADC	ADC Conversion Complete
23	0×002C	EE READY	EEPROM Ready
24	0×002E	ANALOG COMP	Analog Comparator
25	0×0030	TWI	2-wire Serial Interface
26	0×0032	SPM READY	Store Program Memory Ready

A list of defined ISR of ATmega328 is below [4]:

#define	INT0_vect	_VECTOR(1)	/*	External Interrupt Request 0 */
#define	INT1_vect	_VECTOR(2)	/*	External Interrupt Request 1 */
#define	PCINT0_vect	_VECTOR(3)	/*	Pin Change Interrupt Request 0 */
#define	PCINT1_vect	_VECTOR(4)	/*	Pin Change Interrupt Request 0 */
#define	PCINT2_vect	_VECTOR(5)	/*	Pin Change Interrupt Request 1 */
#define	WDT_vect	_VECTOR(6)	/*	Watchdog Time-out Interrupt */
#define	TIMER2_COMPA_vect	_VECTOR(7)	/*	Timer/Counter2 Compare Match A */
#define	TIMER2_COMPB_vect	_VECTOR(8)	/*	Timer/Counter2 Compare Match A */
#define	TIMER2_OVF_vect	_VECTOR(9)	/*	Timer/Counter2 Overflow */
#define	TIMER1_CAPT_vect	_VECTOR (10)	/*	Timer/Counter1 Capture Event */
#define	TIMER1_COMPA_vect	_VECTOR (11)	/*	Timer/Counter1 Compare Match A */
#define	TIMER1_COMPB_vect	_VECTOR (12)	/*	Timer/Counter1 Compare Match B */
#define	TIMER1_OVF_vect	_VECTOR (13)	/*	Timer/Counter1 Overflow */

(continued)

#define	TIMER0_COMPA_vect	_VECTOR (14)	/ *	TimerCounter0 Compare Match A */
#define	TIMER0_COMPB_vect	_VECTOR (15)	/ *	TimerCounter0 Compare Match B */
#define	TIMER0_OVF_vect	_VECTOR (16)	/ *	Timer/Couner0 Overflow */
#define	SPI_STC_vect	_VECTOR (17)	/ *	SPI Serial Transfer Complete */
#define	USART_RX_vect	_VECTOR (18)	/ *	USART Rx Complete */
#define	USART_UDRE_vect	_VECTOR (19)	/ *	USART, Data Register Empty */
#define	USART_TX_vect	_VECTOR (20)	/ *	USART Tx Complete */
#define	ADC_vect	_VECTOR (21)	/ *	ADC Conversion Complete */
#define	EE_READY_vect	_VECTOR (22)	/ *	EEPROM Ready */
#define	ANALOG_COMP_vect	_VECTOR (23)	/ *	Analog Comparator */
#define	TWI_vect	_VECTOR (24)	/ *	Two-wire Serial Interface */
#define	SPM_READY_vect	_VECTOR (25)	/ *	Store Program Memory Road */

4.6.5 Interrupt Enabling and Disabling

By default, interrupt is disabled. Thus, the programmer needs to write code to enable it before using. For this, Global interrupt enable bit in SREG must be set. For Arduino, built in command *sei()* can be used that sets this bit. This allows all interrupts to be enabled with one bit. Similarly, *cli()* can be used to clear the bit, that disables all interrupts with one bit.

Interrupt priority is determined by order in table. Lower addresses have higher priority. The function name format is: *ISR(vector)*, where vector function should define the interrupt routine. To return from ISR, *reti()* function can be used, however it is not required for built in ISR vectors as it is automatically generated for these ISRs.

4.6.6 *Using External Interrupt of ATmega328*

External interrupts monitor changes in signals on pins. To configure an interrupt, the corresponding control registers needs to be setup. Here, we will discuss two types of external interrupt briefly: INT and PCINT. Details of interrupt can be found in the datasheet [4].

The pins used in these external interrupts are listed below:

- INT0 and INT1 – range of event options
  - INT0 – PORT D [2] (i.e., Arduino digital pin 2)
  - INT1 – PORT D [3] (i.e., Arduino digital pin 3)
- PCINT[23:0] – any signal change (toggle)
  - PCINT[0:5] –> PORT B [0:5] (i.e., Arduino digital pins 8-13)
  - PCINT[8:13] –> PORT C [0:5] (i.e., Arduino analog pins 0-5)
  - PCINT[16:23] –> PORT D [0:7] (i.e., Arduino digital pins 0-7)

Note: PCINT for Port B, C, and D are referred to as PCINT0, PCINT1, and PCINT2, respectively. One of the constraints to use these interrupts is that Pulses on inputs must be slower than I/O clock rate.

4.6.7 *INT Interrupts*

To setup the control registers for INT interrupts, below is the information for setting up sense control bits for INT1 (INT0 sensor control is similar) [4].

Bit	7	6	5	4	3	2	1	0	
(0×69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

For the sense control of interrupt 1, see Table 12.1 to select ISC11 and ISC10 proper mode setting. For interrupt 0, similar mode selection needs to be done using ISC01 and ISC00. The external interrupt mask register is shown below. Write 1 to Enable, 0 to Disable corresponding interrupt in this control register.

Table 12.1 Interrupt 1 sense control

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Bit	7	6	5	4	3	2	1	0	
0×1D (0×3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

The external interrupt flag register is given below. Note that interrupt flag bit is set when a change triggers an interrupt request, where 1 represents interrupt is Pending, 0 represents no pending interrupt. Flag is cleared automatically when interrupt routine is executed, thus no explicit action is required when using ISR vectors. Also, flag can be manually cleared by writing a "1" to it.

Bit	7	6	5	4	3	2	1	0	
0×1C (0×3C)	–	–	–	–	–	–	INTF1	IMTF0	EIFR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

For INT interrupts, programmer can choose to write their own custom ISR. To enable, use the instruction: *attachInterrupt(interrupt, function, mode);* where function can be used to write custom ISR function. Here, interrupt value is either 0 or 1 (for INT0 or INT1, respectively). The function represents the custom interrupt function to call. The mode value can be LOW, CHANGE, RISING, or FALLING. To disable the interrupt, use code: *detachInterrupt(interrupt);* where interrupt: Either 0 or 1 (for INT0 or INT1, respectively). Other related functions are *sei()* to enable global interrupt, and *cli()* to disable global interrupt.

4.6.8 PCINT Interrupts

The pin change interrupt (PCINT) control register is:

Bit	7	6	5	4	3	2	1	0	
(0×68)	–	–	–	–	–	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Here, PCIE2 enables interrupts for PCINT[23:16] (i.e., Port D), PCIE1 enables interrupts for PCINT[15:8] (i.e., Port C), and PCIE0 enables interrupts for PCINT [7:0] (i.e., Port B). The corresponding flag register is:

Bit	7	6	5	4	3	2	1	0	
0×1B (0×3B)	–	–	–	–	–	PCIF2	PCIF1	PCIF0	PCIFR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

Status flag of 1 when pending, and 0 when cleared. Note that the flag is cleared automatically when interrupt routine is executed by built in interrupt vector ISRs. The mask register for PCINT2 is given below:

Bit	7	6	5	4	3	2	1	0	
(0×6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	PCMSK2
Read/ Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

The bits 0 to 7 here corresponds to Arduino Pin 0 to 7 (Port D). Each bit controls whether interrupts are enabled for the corresponding pin. Note that change on any enabled pin causes an interrupt (not distinguishable). Mask registers 0 (PCMSK0) and 1 (PCMSK1) are similar.

Example 4.5 For an external interrupt (INT), complete the following partially filled up code.

```
#define pinint0
#define pinint1
     void setup()    (
       pinMode (pinint0,        );
       pinMode(pinint1,        );
       Serial.begin(9600);
       // External interrupts 0 and 1
       // Interrupt on rising edge
       EICRA =
       // Enable both interrupts
       EIMSK =
       // Turn on global interrupts
       sei ();
}
ISR(        _vect) {
}
ISR(        _vect) {
}
// Print out the information
void loop()
{
      Serial.print("X: ");
      Serial.print(percent0);
      Serial.print("    Y:    ");
      Serial.println(percent1);
}
```

Solution
```
#define pinint0 // Defined as Pin 2
#define pinint1 // Defined as Pin 3
int percent0 = 0;
int percent1 = 0;
void setup () {
 pinMode(pinint0, INPUT);
 pinMode(pinint1, INPUT);
 Serial.begin(9600);
 // External interrupts 0 and 1
 // Set interrupt on rising edge
 EICRA |= B00001111;
 // Enable both interrupts
 EIMSK |= B00000011;
 sei(); // Global Interrupt enable
}
ISR(INT0_vect) {
 percent0++;
}
ISR(INT1_vect) {
 percent1++;
}
// Print out the information
void loop () {
 // Put MCU to sleep to save power
 // See subsequent slides for details
 Serial.print("X: ");
 Serial.print(percent0);
 Serial.print("  Y: ");
 Serial.println(percent1);
}
```

Example 4.6 Setup your hardware as follows with an Arduino Uno board:

- From VDD to two push switches in series with two resistors (1 kΩ each).
- Connect Pin 8 to one switch at resistor
- Connect Pin 3 to the other switch at resistor

Write a code that uses PCINT such that when one of the switches connected to Pin 8 is pressed, the *value* increments, but when the switch connected to Pin 5 is pressed, the *value* decrements. Use serial port monitor in Arduino sketch to monitor this data.

Solution A code is provided below to perform the above task.

```
#include <avr/interrupt.h>
volatile int value = 0;
void setup () {
 // Global interrupt disable - Atomic
 cli();
 // Enable PCINT0 and PCINT2
 PCICR |= B00000101;
 // Mask for Pin 8 (Port B)
 PCMSK0 |= B00000001;
 // Mask for Pin 3 (Port D)
 PCMSK2 |= B00001000;
// Global interrupt enable
 sei();
 // Serial port initialize
 Serial.begin(9600);
}
// ISR for Pin 8 interrupt, inc value
 ISR(PCINT0_vect) {
  value++;
 }
 // ISR for pin 3 interrupt, dec value
 ISR(PCINT2_vect) {
  value--;
 }
 // Main loop prints value
 // Note: no port checking statement
 void loop () {
  Serial.println(value);
 }
```

Example 4.7 How can you use interrupt for implementing a ring buffer?

Solution Interrupt can be used to monitor input data ports and the ISR will collect data from the port to the ring buffer. The Head pointer will be incremented for each data read. In the main code, data from ring buffer will be read and corresponding Tail pointed will be moved accordingly. Boolean variables can be used to check the constraints such as overflow condition. A partial code is provided below. Here one block of data can be processed by the main code, thus it must wait until enough data is collected.

```
// Main code --- Read process
...
if(Overflow == 0) {
   Valid_Data = Head - Tail; // Head & Tail as defined
 } else {
   Valid_Data = Head + buffer_size - Tail; // Wraparound
}
if(Valid_Data >= block_size)
   {
 (read data block)
...
```

```
// Check if sufficient data of 1 block has accumulated
if ((Tail + block_size) < buffer_size)
     Tail = Tail + block_size;
    else{
    Overflow = 0;
    Tail = Tail + block_size - buffer_size;
      }
...
// ISR code ---- Write process
...
if(Head == buffer_size){ // Wraparound
    Head = 0;
    Overflow = 1;
  }
// detect error condition
// Tail > Head for Overflow = 1
  if((Overflow == 1) && (Tail < Head))
  hold_Flag = 1; // halts new data to put in Ring buffer
  else
  hold_Flag = 0; //allows new data to put in ring buffer
 ...
  if((Head < buffer_size) && (hold_Flag == 0)){
...
// reads data from analog pin 0
    buffer[Head++] = analogRead(0);
...
```

4.7 Sleep Modes

One of the major advantages of OS-less MCU is the use of deep-sleep modes that drastically reduce power consumptions. However, to wake up the MCU from deep sleep, interrupt needs to be used (leading to reactive ES). Now that we have studied interrupt, it is a good time to look into sleep modes. Here we will focus on ATmega328 microcontroller sleep modes as this is the MCU of Arduino Uno board.

4.7.1 ATmega328 Sleep Modes

The table below shows various sleep modes available in this MCU [4].

	Active clock domains					Oscillators		Wake-up sources							
Sleep mode	Clk_{CPU}	Clk_{FLASH}	Clk_{IO}	Clk_{ADC}	Clk_{ASY}	Main Clock Source Enabled	Timer Oscillator Enabled	INT1, INT0 and Pin Change	TWI Address Match	Timer2	SPM/ EEPROM Ready	ADC	WDT	Other/ O	Software BOD Disable
Idle			x	x	x	x	x[b]	x	x	x	x	x	x	x	
ADC Noise Reduction				x	x	x	x[b]	x[c]	x	x[b]	x	x	x		
Power-down								x[c]	x				x		x
Power-save					x		x[b]	x[c]	x	x			x		x
Standby[a]						x		x[c]	x				x		x
Extended Standby					x[b]	x	x[b]	x[c]	x	x			x		x

[a]Only recommended with external crystal or resonator selected as dock source.
[b]If Timer/Counter2 is running in asynchronous mode.
[c]For INT1 and INT0, only level interrupt.

Note that some hardware units are turned off in some sleep modes. This is critical to understand to use the sleep modes. The objective is to use the lowest power sleep mode provided the needed hardware during the sleep mode is active. ATmega328 microcontroller typical current consumptions in various sleep conditions (high to low) are given below with a list of key hardware units that are available in that sleep mode:

SLEEP_MODE_IDLE	15 mA	All I/O, clk, timers, mem
SLEEP_MODE_ADC	6.5 mA	ADC, EEPROM, clk, Timer2, mem
SLEEP_MODE_PWR_SAVE	1.6 mA	Main clk, clk(asy), Timer2
SLEEP_MODE_EXT_STANDBY	1.6 mA	Clk(asy), Timer osc, Timer2 (no main clk)
SLEEP_MODE_STANDBY	0.8 mA	Main clk
SLEEP_MODE_PWR_DOWN	0.4 mA	Everything off (expt. Interrupt, WDT)

4.7.2 How to Enable Sleep Mode?

To enable sleep mode, the avr/sleep.h library needs to be included. In the setup code, write instruction for *set_sleep_mode(mode);* that initializes the sleep mode. In the loop function, write *sleep_mode();* to active sleeping. This statement is actually a combination of 3 statements in sequence: *sleep_enable(); cpu_sleep(); sleep_disable ();*

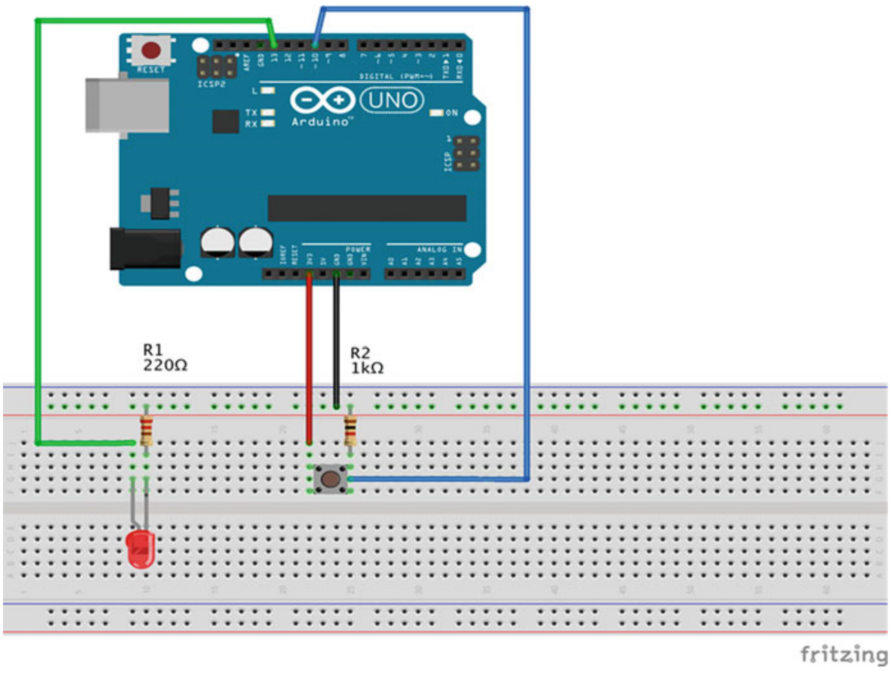
```
#include <avr/interrupt.h>
#include <avr/sleep.h>
void setup() {
 ...
 set_sleep_mode(SLEEP_MODE_IDLE); // select mode
 ...
}
void loop () {
  ...
  sleep_mode();
  ...
}
```

When an interrupt occurs to wake up the MCU, it resumes from the sleep_mode (or sleep_disable) statement.

Example 4.8 Setup your Arduino Uno with hardware as shown below. Notes for hardware setup:

- From VDD (3.3V), connect a push switch and a 1kΩ resistor in series to ground (0V).
- Pin 10 connects to the midpoint of the push switch and 1kΩ resistor.

- From pin 13, connect an LED (red) with a 220Ω (or 330Ω) resistor in series to ground.
- The 4-pin push switch has two sets of pins internally connected. To connect the switch properly, connect the two pins on the same side of the switch.



Now complete the partially filled up code below, so that the LED toggles when the push switch is pressed. Use the highest possible sleep mode (i.e., lowest current during sleep).

```
#include <avr/interrupt.h>
#include <avr/sleep.h>
void setup() {
  cli(); // Clear global interrupt
  // Set Pin 13 as output and 10 as input
  DDRB |=           ;
  DDRB &=          ;
  // Control regs for PCINT
  PCICR |=        ; // Enable PCINT0
  PCMSK0 |=       ; // Select PCINT0 mask
  // Serial.begin(9600); // Only for debug
 sei(); // Set global Interrupt
 // Use an appropriate sleep mode
  set_sleep_mode(        );
}
```

```
  // ISR for pin change interrupt capture
 // Note: triggers both on rising & falling
ISR(       _vect) {
  // Display in serial monitor for debug
  //  Serial.println("Switch pressed");
  // Toggle the LED
  PORTB      ;
}
// Main loop
void loop() {
  // Display in serial monitor for debug
  //  Serial.println("Main loop");
  // Do nothing!
  // Put MCU to sleep
}
```

Solution The completed code is given below:

```
#include <avr/interrupt.h>
#include <avr/sleep.h>
void setup() {
  cli(); // Clear global interrupt
  // Set Pin 13 as output and 10 as input
  DDRB |= B10000000  ;
  DDRB &= B111011   ;
  // Control regs for PCINT
  PCICR |= B00000001 ; // Enable PCINT0
  PCMSK0 |= B00000100  ; // Select PCINT0 mask
   // Serial.begin(9600); // Only for debug
   sei(); // Set global Interrupt
  // Use an appropriate sleep mode
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
}
// ISR for pin change interrupt capture
// Note: triggers both on rising & falling
ISR(PCINT0_vect) {
  // Display in serial monitor for debug
  //  Serial.println("Switch pressed");
  // Toggle the LED
  PORTB ^= B100000 ;
}
// Main loop
void loop() {
  // Display in serial monitor for debug
  //  Serial.println("Main loop");
  // Do nothing!
  // Put MCU to sleep
  sleep_mode();
}
```

4.8 Using MCU Internal Timer/Counter Hardware Unit

Precise time (delay) count requires use of hardware timer/counter internal to the MCU. It is also useful for timed ADC data capture. Never use *for loop* to generate delay in ES, as it is the most inefficient method. Arduino Sketch built in *delay* function uses these internal timer/counter hardware, which is great for precision timing. But MCU cannot be put to sleep with the *delay* function. Thus it is essential for understanding and using the timer/counter hardware directly to take advantage of sleep capabilities of MCU.

4.8.1 Internal Timer/Counter of ATmega328

ATmega328 has three timer/counter units as follows:

- Timer/counter0: 8-bit
- Timer/counter1: 16-bit
- Timer/counter2: 8-bit

Note: Only Timer/counter2 is ON during sleep modes (except IDLE mode). Below is the list of registers related to timer/counter of ATmega328.

TCNTx—Timer/counter count register
OCRxA—Output Compare Register A
OCRxB—Output Compare Register B
TCCRxA—Timer/counter control register A
TCCRxB—Timer/counter control register B
TIMSKx—Timer/counter interrupt mask register
TIFRx—Timer/counter interrupt flag register

Here, x can be 0, 1, or 2 for corresponding timer/counter. The timers/counters can generate two types of interrupts: TOV for overflow, and Computer A&B types for comparing to a set value. In this text, we will learn about TOV as this is simpler to code.

The timer/counter control registers for timer/counter0 are given below. Control registers of Timer/Counter1 and Timer/Counter2 are similar.

Bit	7	6	5	4	3	2	1	0	
0×24 (0×44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/ Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0×25 (0×45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	w	W	R	R	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

These two control registers setup the counter mode using WGM (Waveform Generation Mode) bits and pre-scaler using Clock Source select bits. We will use normal mode for WGM, as given in the table below.

Mode	WGM02	WGM01	WGM00	Timer/counter mode of operation	TOP	Update of OCRx at	TOV Flag Set on[a,b]
0	0	0	0	Normal	0×FF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0×FF	TOP	BOTTOM
2	0	1	0	CTC	OCR A	Immediate	MAX
3	0	1	1	Fast PWM	0×FF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

[a]MAX = 0×FF
[b]BOTTOM = 0×00

As these timers are small (8 or 16 bits), they cannot count long periods if driven directly my internal clock. To provide a better solution, these timer/counter can use pre-scalar hardware that can divide the internal clock by 1 to 1,024 factors by Clock Source (CS) select bits. For CS selection table for Timer/Counter0 is given below. Timer/Counter1 table is similar.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$\mathrm{Clk}_{I/O}$/(No prescaling)
0	1	0	$\mathrm{clk}_{I/O}$/8 (From prescaler)
0	1	1	$\mathrm{clk}_{I/O}$/64 (From prescaler)
1	0	0	$\mathrm{clk}_{I/O}$/256 (From prescaler)
1	0	1	$\mathrm{clk}_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge
1	1	1	External clock source on T0 pin. Clock on rising edge

Here the external clock can be applied with T0 pin which is connected to Port D [4], and T1 pin which is connected to Port D[5]. The CS selection table for Timer/Counter2 is different as given below. Note that this timer/counter does not have external clock source selection option.

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	clk_{T2S}/(No prescaling)
0	1	0	clk_{T2S}/8 (From prescaler)
0	1	1	clk_{T2S}/32 (From prescaler)
1	0	0	clk_{T2S}/64 (From prescaler)
1	0	1	clk_{T2S}/128 (From prescaler)
1	1	0	clk_{T2S}/256 (From prescaler)
1	1	1	clk_{T2S}/1024 (From prescaler)

Output of the timer/counter can be configured to an output pin. Thus, we can use software to generate clock signal of (almost) any frequency. The interrupt mask register for Timer/Counter0 is:

Bit	7	6	5	4	3	2	1	0	
(0×6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Here, TOIE0 allows Timer Overflow Interrupt Enable, and OCIE0A/B is to setup Compare A/B interrupt enable. The corresponding flag register is:

Bit	7	6	5	4	3	2	1	0	
0×15 (0×35)	–	–	–	–	–	OCF0B	OCF0A	TOV0	TIFR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Where TOV0 is for Timer overflow flag, and OCF0A/B is for Compare A/B interrupt flag. Timer/Counter 1 and 2 have similar registers.

4.8.2 OVF Timer Interrupt

To use OVF (Overflow) timer interrupt, we need to set WGM to Normal mode (i.e., 0). In this configuration, timer value increments, and when it reaches the highest value (i.e., 0xFF for 8-bit and 0xFFFF for 16-bit), it wraps around at TOP. At that instance, OVF interrupt is issued. Note that the timer/counter starts again at 0, so must reset to counter value if count must be done for a smaller value that the full range. For example, if using Timer/Counter0, then TOV0 interrupt flag will be set when TCNT0 resets to 0x00 from 0xFF. It can be used to generate periodic signals (like a clock). Note that Timer/Counter2 can be used with sleep mode to develop ultra-low-power ES like battery-powered IoTs. Timers can also be used to generate an interrupt every N time units. For this purpose, set TCNT0/2 to an initial value of (255-N), or for the case of TCNT1, an initial value of (65,535-N).

Example 4.9 What is the value of Timer1 needed for 1 second count in ATmega328 with 16 MHz clock?

Solution Internal clock = 16 MHz. As the needed time is large, we will use the maximum value of pre-scaler, which is 1024. Thus, clock cycles needed = 16 * 10^6/1024 = 15,625. Timer1 is 16-bit, so maximum value (i.e., 0xFFFF) = 65,535. So, Timer1 count value needed = 65,535 − 15,625 = 49,910. Thus, the value of Timer1 needs to be set to 49,910 or 0xC2F6.

Example 4.10 Write a code for 1 second timer for ATmega328 (Arduino Uno). The hardware setup: connect pin 13 to an LED in series of 220 Ω or 330 Ω resistor to ground.

Solution Code for 1 second timer using Timer/Counter1 hardware.

```
void setup () {
 cli(); // Disable global interrupt - atomic
 DDRB |= B100000; // Pin 13 output
 // Set timer 1 to normal mode
 TCCR1A = B00000000;
 // Set pre-scaler to 1024
 TCCR1B = B00000101;
 // Turn ON OVF
 TIMSK1 = B00000001;
 // Initial Timer1 value for 1 sec count
 TCNT1 = 0xC2F6;
 sei(); // Enable global interrupt
}

// Timer1 ISR
ISR(TIMER1_OVF_vect) {
 // Toggle output pin each 1 sec
 PORTB ^= B100000;
 // Reset counter value for next 1 sec
 TCNT1 = 0xC2F6;
}
// Main loop
void loop () {
 // Do nothing
 // If including sleep mode, ensure
 // timer1 is ON while sleep
}
```

Example 4.11 Write a code for 1 second timer for ATmega328 (Arduino Uno) such that a low-power sleep mode can be used (other than IDLE mode). The hardware setup: connect pin 13 to an LED in series of 220 or 330 Ω resistor to ground.

Solution Code for 1 second timer using Timer/Counter2 hardware.

```
#include <avr/sleep.h>
char rep = 0; // Timer repeat count
void setup () {
 // Set pin 13 as output
 DDRB |= B100000;
 // Using Timer2, normal mode
 TCCR2A = B00000000;
 // Pre-scaler 1024 (max)
 TCCR2B = B00000111;
 // Pre-scaled clock rate = 16M/1024
 // = ~16k
 // Timer max count=16k/0.25k=~64
 // Turn on OVF interrupt
 TIMSK2 = B00000001;
 // Turn on global interrupt
 sei();
}
// ISR for TOV that triggers it
ISR(TIMER2_OVF_vect) {
 rep++; // Increment repeat count
 // For 1 sec, 64 repeats needed
 if (rep == 64) {
  rep = 0; // Reset repeat count
   PORTB ^= B100000; // toggle bit 13
 }
}
// Main loop
void loop() {
 // void loop Nothing to do
 // set sleep mode and sleep cpu
 set_sleep_mode(SLEEP_MODE_PWR_SAVE);
 sleep_mode();
}
```

Example 4.12 Complete the partial code for 1/64 second timer interrupt output generator for ATmega328 (Arduino Uno).

```
char timer = 0;
void setup() {
  DDRB =       ; // Pin 13 as output
  // Using timer, Set to Normal mode, Pin OC0A disconnected
  TCCR2A =      ;
  // Prescale clock by 1024, Interrupt every 256K/16M sec = 1/64 sec
  TCCR2B =      ;
  // Turn on timer overflow interrupt flag
  TIMSK2 =    ;
  sei(); // Turn on global interrupts
}
ISR(     _vect) {
  timer++;
  PORTB =      ; // Toggle bit 13
}
void loop() {
 // Do nothing
}
```

Solution The completed code is provided below.

```
char timer = 0;
void setup() {
 DDRB |= B100000  ; // Pin 13 as output
 // Using timer, Set to Normal mode, Pin OC0A disconnected
 TCCR2A = B00000000 ;
 // Prescale clock by 1024, Interrupt every 256K/16M sec = 1/64 sec
 TCCR2B = B00000111 ;
 // Turn on timer overflow interrupt flag
 TIMSK2 = B00000001 ;
 sei(); // Turn on global interrupts
}
ISR(TIMER2_OVF_vect) {
 timer++;
 PORTB ^= B100000 ; // Toggle bit 13
}
void loop() {
 // Do nothing
}
```

Example 4.13 Write a code to generate a 100 Hz waveform through Pin 13 using timer2 hardware of ATmega328 (Arduino Uno).

Solution Using max pre-scaler, modified clock = 16M/1024 = 15,625 Hz. For 100 Hz, we need 10 ms clock period, i.e., 5 ms ON time, 5 ms OFF time. For 5 ms toggle timer, number of modified clock cycles=15,625 $*$ 5 $*$ 10^{-3} = ~78. Count value needed for Timer 2 = 255 − 78 = 177 = 0xB1.

The complete code is provided below.

```
void setup () {
 // Set pin 3 as output (arbitrary)
 DDRD |= B00001000;
 // Using Timer2, normal mode
 TCCR2A = B00000000;
 // Pre-scaler for 1024
 TCCR2B = B00000111;
 // Turn on OVF interrupt
 TIMSK2 = B00000001;
 // Set initial count value
 TCNT2 = 0xB1;
 // Turn on global interrupt
 sei();
}
// ISR for TOV that triggers it
ISR(TIMER2_OVF_vect) {
 // Toggle output: On->Off->On etc.
 PORTD ^= B00001000; // toggle bit 3
 // Re-initialize timer count value
 TCINT2 = 0xB1;
}
// Main loop
```

```
void loop () {
 // Nothing to do
 // CPU can be put to sleep with
 // a proper mode selection
}
```

Example 4.14 Write a code for a traffic light control hardware (Red, Yellow, Green light outputs, and a pedestrian push switch input) using timer hardware of ATmega328 (Arduino Uno).

Solution A complete code is provided below.

```
#include <avr/interrupt.h>
#include <avr/sleep.h>
char tick = 0; // Unit time
void setup () {
 cli();
 // Set pin 11, 12, 13 as output
 DDRB |= B111000;
 // Set pin 10 as input ped switch
 DDRB &= B111011;
 // Timer 1 code, normal mode
 TCCR1A = B00000000;
 // Pre-scaler 1024 (max)
 TCCR1B = B00000101;
 // Turn on OVF interrupt
 TIMSK1 = B00000001;
 // Initialize for 1 sec time tick
 TCNT1 = 0xC2F6;
 // setup input switch interrupt: PCINT0
 PCICR |= B00000001;
 // Pin 10 as input for PCINT0 interrupt
 PCMSK0 &= B111011;
 // For debugging only
 // Serial.begin(9600);
 // Turn on global interrupt
 sei();
}
// ISR for TOV that triggers it
ISR(TIMER1_OVF_vect) {
 tick++; // Increment tick -> elapsed sec
 switch(tick) {
  case 5: PORTB |= B001000; break; // Yellow
  case 6: PORTB |= B010000; break; // Green
  case 10: PORTB |= B100000;
          tick = 0; break; // Red, reset tick
  default: break; // Do nothing
 }
 TCNT1 = 0xC2F6; // Reset counter
}
 // Ped switch capture as soon as pressed
ISR(PCINT0_vect) {
 // if green, only then take action
```

```
    if (tick > 5) {
     // Reset timer for yellow
     tick = 0;
     TCNT1 = 0xC2F6; // Restart time count
     PORTB |= B001000; // Turn ON Yellow
    }
  }
  // Main loop
  void loop () {
   // For debug only
   // Serial.println(tick);
   // Optionally you can set sleep mode
  }
```

4.9 Digital Filter

As mentioned earlier, some hardware blocks might be implemented either in hardware or in software. Filter is one of those types of blocks that can be implemented either in hardware or in software. Previously, we studied implementation of filter block in hardware. Here we discuss implementation of filter block in software.

4.9.1 Digital Filter Types

Digital filters are of two types:

1. Finite Impulse Response (FIR),
2. Infinite Impulse Response (IIR).

FIR filter does not have any feedback from output, whereas IIR filter has feedback from output. The generic forms of these filters are:

FIR filter: $y[n] = b0x[n] + b1x[n-1] + \ldots + bNx[n-N]$
IIR filter: $y[n] = 1/a0\,(b0x[n] + b1x[n-1] + \ldots + bNx[n-N] - a1y[n-1] - a2y[n-2] - \ldots - aQy[n-Q])$

Here *x[n]* is input signal, *y[n]* is output signal, and *a* & *b* are coefficients for various output and input signal samples, respectively.

4.9.2 Implementation of Digital Filters with Arduino Uno

Here are some example implementation of Digital Filters with Arduino Uno codes.

Example 4.15 How you will design a 3rd order FIR filter with the following equation where the streaming data input is coming from ADC 3:

$$y(n) = (x(n) + x(n-3))/2$$

Solution We will need up to input samples of x[3] data, i.e., x[0], x[1], x[2], x[3]. Afterwards, in the loop code, we will need to shift the samples on each cycle by 1 position as follows:

x[3] = x[2]; // Producing sample delay
x[2] = x[1]; // Producing sample delay
x[1] = x[0]; // Producing sample delay
x[0] = analogRead(3); // New data from ADC 3

Next, we will use the following equation to perform the filter operation:

y = (x[0] + x[3]) >> 1; // Right shift by 1 bit is same as div by 2

Note that here we have used right shift by 1 bit to implement division by 2. This right shift operation (typically 1 clock cycle) is much faster that division operation.

Example 4.16 Design a high pass filter (HPF) with the following IIR equation for Arduino Uno where the streaming input data is coming from Analog input 0.

$$y[n] = \propto \left(y[n-1] + (x[n] - x[n-1])\right) \text{ where } \propto = \frac{RC}{RC + \Delta t}$$

Solution A code is provided below.

```
const float alpha = 0.5; // controls filter response
double data_filtered[] = {0, 0};
// Output data, i.e. y[0] and y[1]
double data[] = {0, 0};
// Incoming data storage, i.e. x[0] and x[1]
const int n = 1; // IIR depth
// Analog Pin 0; change to where analog data is connected
const int analog_pin = 0;
void setup(){
 Serial.begin(9600); // Initialize serial port
}

void loop(){
 // Retrieve incoming next data
 data[0] = analogRead(analog_pin);
 // High Pass Filter using the above IIR equation
 data_filtered[n] = alpha * (data_filtered[n-1] + data[n] - data[n-1]);
 // Store the previous data in correct index
 data[n-1] = data[n];
 data_filtered[n-1] = data_filtered[n];
 Serial.println(data_filtered[0]); // Print Data
 delay(100); // Wait before next data collection
}
```

Example 4.17 Design a low pass filter (LPF) with the following IIR equation for Arduino Uno where the streaming input data is coming from Analog input 2.

$y[n] = \propto x[n] + (1 - \propto)y[n-1]$ where $\propto = \frac{\Delta t}{RC + \Delta t}$

Solution A code is provided below.

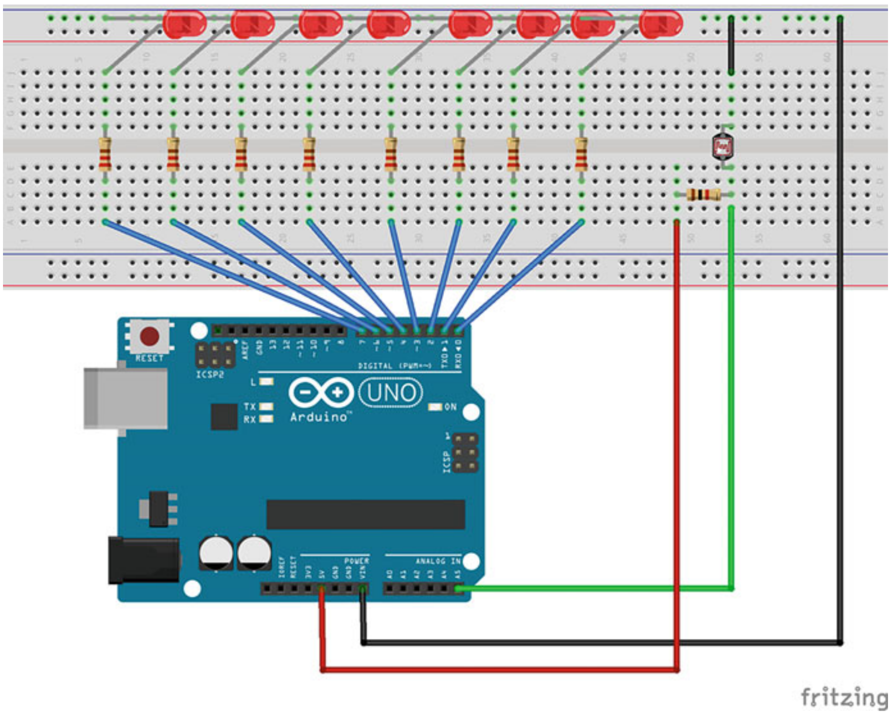
```
const float alpha = 0.5; // controls filter response
double data_filtered[] = {0, 0};
// Output data, i.e. y[0] and y[1]
double data; // Incoming data storage, i.e. x[0]
const int n = 1; // IIR depth
// Analog Pin 2; change to where analog data is connected
const int analog_pin = 2;
void setup(){
 Serial.begin(9600); // Initialize serial port
}
void loop(){
 // Retrieve incoming next data
 data = analogRead(analog_pin);
 // Low Pass Filter using the above IIR equation
 data_filtered[n] = alpha * data + (1 - alpha) * data_filtered[n-1];
 // Store the last filtered data in data_filtered[n-1]
 data_filtered[n-1] = data_filtered[n];
 Serial.println(data_filtered[n]); // Print Data
 delay(100); // Wait before next data collection
}
```

4.10 Artificial Intelligence

Artificial intelligence (AI) is becoming a major driving force in newer generation ES such as IoTs, wearables, smart devices, and self-driving cars. Here, we briefly discuss a few related aspects to introduce the concept. Any AI system would possess some intelligence or learning ability. Non-AI system are designed by programmers to perform task in a certain way. For instance, an ES street-light controller can be designed by the programmer such that the light turns on when the ambient light falls below certain level by hard-coding a threshold value in the program. We can easily make this system adaptive to deployed setting by using a variable that records the maximum light and minimum light setting at the deployed setting, and set the threshold to a percentage of this range between the maximum and minimum. This system will have some adaptive nature or intelligence.

Example 4.18 Design an Arduino Uno to light up eight LEDs one by one when the light becomes lesser than maximum light. All eight LEDs must be on when the ambient light is lowest and none of the light must be on when the ambient light is maximum.

Solution We can include a variable to keep track of maximum light and another variable to keep track of minimum light. Then we can turn on the LEDs one by one when the ambient light falls below certain percentage. A schematic diagram and a code are provided below.



```
// Delay 1 is On and 2 is Off
int i = 0;
long randN;
int sensorPin = A0; // select the input pin for LDR
int sensorValue = 0; // to store the value from sensor
float Val = 0.0; // scaled value
float maxVal = 0.0; // maximum recorded value
float minVal = 0.0; // minimum recorded value
// the setup function
void setup() {
//  Serial.begin(9600);
//sets serial port for comm, if needed
// initialize 8 digital pins as an output.
 pinMode(0, OUTPUT);
 pinMode(1, OUTPUT);
 pinMode(2, OUTPUT);
 pinMode(3, OUTPUT);
 pinMode(4, OUTPUT);
 pinMode(5, OUTPUT);
 pinMode(6, OUTPUT);
 pinMode(7, OUTPUT);
}
void loop() {
// read the value from the sensor
sensorValue = analogRead(sensorPin);
//prints the values coming from
// the sensor on the screen
//Serial.println(sensorValue);
// turn all LED off
for (int i = 0; i <= 7; i++) {
 digitalWrite(i, LOW);
}
// Learning process
if (maxVal<sensorValue)
  maxVal = sensorValue;
if (minVal>sensorValue)
  minVal = sensorValue;
// Linear normalized scaling
Val = (float(sensorValue) - minVal)/(maxVal - minVal);
if (Val > 0.1) digitalWrite(0, HIGH);
if (Val > 0.2) digitalWrite(1, HIGH);
if (Val > 0.3) digitalWrite(2, HIGH);
if (Val > 0.4) digitalWrite(3, HIGH);
if (Val > 0.5) digitalWrite(4, HIGH);
if (Val > 0.6) digitalWrite(5, HIGH);
if (Val > 0.7) digitalWrite(6, HIGH);
if (Val > 0.8) digitalWrite(7, HIGH);
delay(100);
}
```

Although this type of intelligent code has some ability to adapt, but this approach is not applicable for complex decision space where decision boundary needs to be non-linear (Fig. 4.5). For those cases, non-linear algorithms are required to achieve high accuracy and sensitivity. Machine learning (ML) is one of the approach that can produce these types of non-linear boundary space without finding the complex process of detailed mathematical modeling. It is becoming one of most used AI technique for ES. The basic concept evolved in attempt to model human learning process. Furthermore, a relative new technique called Deep learning has emerged that is even more powerful, but requires a very large number of training dataset. All of these are subset of AI as shown in Fig. 4.6.

4.10.1 Machine Learning (ML) and Deep Learning

Machine learning (ML) algorithms generate data clusters by finding connections that may not be obvious or expected. ML approach requires a dataset with known true outcome (called Ground Truth). The ML network is first trained with a portion of this

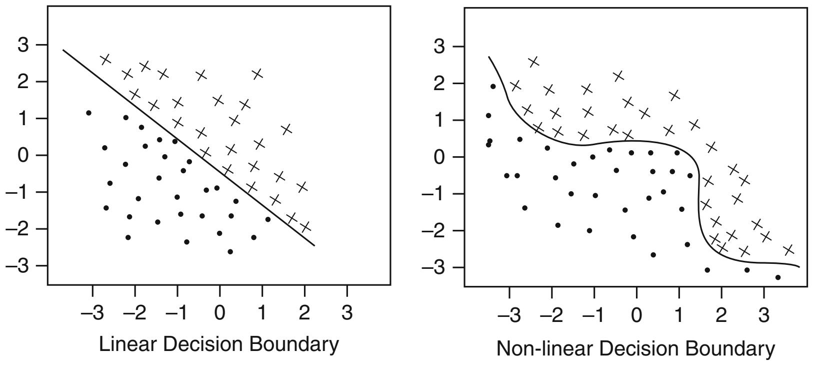
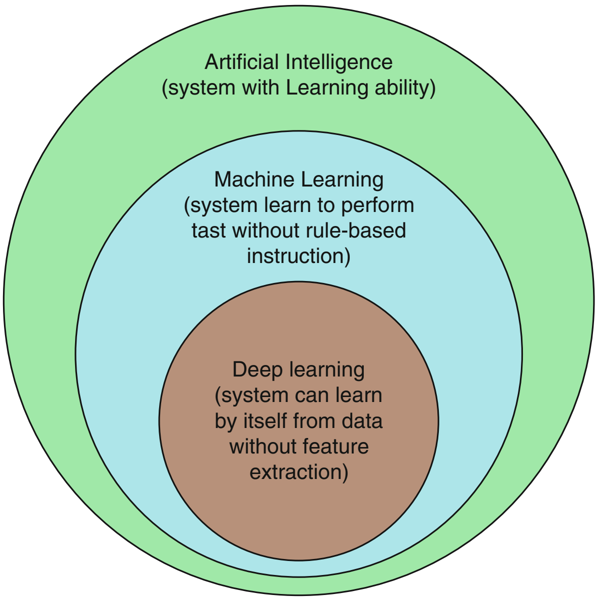


Fig. 4.5 Linear and non-linear decision boundary

Fig. 4.6 Artificial intelligence, machine learning, and deep learning



dataset. After algorithm training, the algorithm is tested with rest of the dataset. If the performance (accuracy, sensitivity, specificity, error rate, false positive, false negative, etc.) is satisfactory, then the algorithm is ready for deployment. The scheme is shown in Fig. 4.7.

Artificial Neural Network (ANN) is one of the commonly used network models for ML. It consists of Input layer, Hidden layer, and Output layer. These layer nodes can be fully or partially connected with various weights. When "extracted features" of interest from the training data is fed to the network, the network evaluates the random initial weightages are re-evaluated using a technique called Back-propagation to reduce error. Next iteration uses the newly calculated values. The iterations continue until error falls below an acceptable value. At that point, the weights represent the trained ANN network, which can be implemented for testing and deployment.

Other than ANN, there are many other ML algorithms. Some are supervised and some are unsupervised. Supervised learning ML can be two types: classification type and regression type. Examples of classification type supervised ML are ANN, Support Vector Machine (SVM), Naive Bayes, and Nearest Neighbor. Examples of regression type supervised ML are Linear Regression, Support Vector Regression, Ensemble, and Decision Trees. Unsupervised ML works on Clustering method. Examples include K-means, K-Medoids, Fuzzy C-means, Hierarchical, Gaussian mixture, and Hidden Markov Model (HMM). Some of them are continuous, such as Linear and polynomial regression, Decision Tress, Random Forest, Principal Component Analysis (PCA) and K-means, while other are categorical such as K-nearest neighbor (KNN), Logistic regression, SVM, Naive-Bayes, and HMM.

Classical ML network cannot learn from deployment, rather the network is pretrained prior to deployment. A newer approach to implement ML to be able to learn from deployment is called Reinforcement Learning. This promising approach can adjust ML network parameters based on feedbacks at deployment.

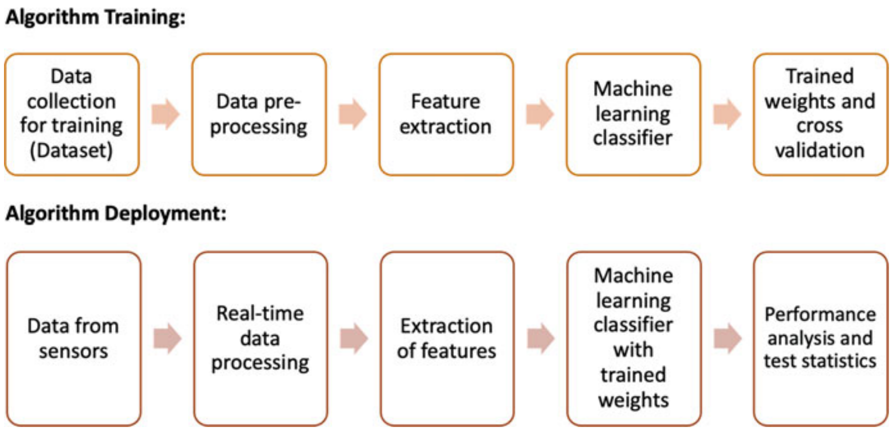

Algorithm Training:
Data collection for training (Dataset)
Data pre-processing
Feature extraction
Machine learning classifier
Trained weights and cross validation
Algorithm Deployment:
Data from sensors
Real-time data processing
Extraction of features
Machine learning classifier with trained weights
Performance analysis and test statistics

Fig. 4.7 Machine learning (ML) algorithm training and testing process

Another very promising AI is Deep Learning. In Deep Learning, the programmer does not need to train the network with extracted features, rather feed the raw data directly to the Deep Learning network. Thus, deep learning network can learn from non-obvious salient features by itself; but it requires a very large amount of data. With the explosion of data (known as Big Data) with large number of IoTs, wearables, and smart devices, deep learning becomes possible and is a prime research interest.

4.10.2 Implementing ANN in Arduino Uno

Implementing ML on OS-less ES is challenging. Currently, approaches are tedious and not suitable. Thus, it is currently recommended to use OS-based ES, such as Raspberry Pi, for ML algorithm implementation. Nonetheless, newer libraries of OS-less platforms are under development and might become suitable in future.

Example 4.19 Write an ANN based machine learning training model for Arduino Uno.

Solution Below is code that implements ANN ML training on Arduino Uno.

```
   //Author: Ralph Heymsfeld
   //28/06/2018
   //With minor modification
   #include <math.h>
   /*************************************************************
****
    * Network Configuration - customized per network
    *************************************************************
***/
   const int PatternCount = 10; // how many training dataset
   const int InputNodes = 3; // feature number
   const int HiddenNodes = 5; // larger than input nodes
   const int OutputNodes = 2; // output controls
   const float LearningRate = 0.3;
   const float Momentum = 0.9;
   const float InitialWeightMax = 0.5;
   const float Success = 0.001; //0.1% error
   // training dataset, synthesized
   // changed "const char" to "const float"
   // e.g. {Peak, Av, std} or {light, humidity, air flow}
   // in normalized form
   const float Input [PatternCount] [InputNodes] = {
    { 1, 0, 0 }, // 0
    { 1, 0.5, 0 }, // 1
    { 1, 0.2, 0.2 }, // 2
    { 0.5, 0.5, 0 }, // 3
    { 1, 0.5, 0.1 }, // 4
    { 0, 0.5, 0.1 }, // 5
    { 0.5, 0.4, 0.2 }, // 6
    { 1, 1, 0 }, // 7
    { 0, 0.9, 0.1 }, // 8
    { 0.5, 0.8, 0.2 }  // 9
   };
   // ground truth to training dataset
   // {Cooler, Heater }
   // data output is Bool
   const byte Target [PatternCount] [OutputNodes] = {
    { 1, 0 }, // 0
    { 1, 0 }, // 1
    { 1, 0 }, // 2
    { 0, 0 }, // 3
    { 0, 0 }, // 4
    { 0, 0 }, // 5
    { 0, 0 }, // 6
    { 0, 1 }, // 7
    { 0, 1 }, // 8
    { 0, 1 }  // 9
   };
   /***************************************************************
****
    * End Network Configuration
    ***************************************************************
***/
   int i, j, p, q, r;
   int ReportEvery1000;
   int RandomizedIndex [PatternCount] ;
   long TrainingCycle;
   float Rando;
   float Error;
   float Accum;
   float Hidden [HiddenNodes] ;
   float Output [OutputNodes] ;
   float HiddenWeights [InputNodes+1] [HiddenNodes] ;
   float OutputWeights [HiddenNodes+1] [OutputNodes] ;
   float HiddenDelta [HiddenNodes] ;
   float OutputDelta [OutputNodes] ;
   float ChangeHiddenWeights [InputNodes+1] [HiddenNodes] ;
   float ChangeOutputWeights [HiddenNodes+1] [OutputNodes] ;
   void setup (){
    Serial.begin(9600);
    randomSeed(analogRead(3));
    ReportEvery1000 = 1;
    for ( p = 0 ; p < PatternCount ; p++ ) {
     RandomizedIndex [p] = p ;
    }
   }
   void loop (){
  /****************************************************************
***
   * Initialize HiddenWeights and ChangeHiddenWeights
   ***************************************************************
***/
    for ( i = 0 ; i < HiddenNodes ; i++ ) {
```

```
    for( j = 0 ; j <= InputNodes ; j++ ) {
     ChangeHiddenWeights[j][i] = 0.0 ;
     Rando = float(random(100))/100;
     HiddenWeights[j][i] = 2.0 * ( Rando - 0.5 ) * InitialWeightMax ;
    }
   }

  /***********************************************************
***
   * Initialize OutputWeights and ChangeOutputWeights
   ***********************************************************
***/
   for( i = 0 ; i < OutputNodes ; i ++ ) {
    for( j = 0 ; j <= HiddenNodes ; j++ ) {
     ChangeOutputWeights[j][i] = 0.0 ;
     Rando = float(random(100))/100;
     OutputWeights[j][i] = 2.0 * ( Rando - 0.5 ) * InitialWeightMax ;
    }
   }
   Serial.println("Initial/Untrained Outputs: ");
   toTerminal();
  /***********************************************************
***
   * Begin training
   ***********************************************************
***/
   for( TrainingCycle = 1 ; TrainingCycle < 2147483647 ; TrainingCycle+
+) {
  /***********************************************************
***
   * Randomize order of training patterns
   ***********************************************************
***/
    for( p = 0 ; p < PatternCount ; p++) {
     q = random(PatternCount);
     r = RandomizedIndex[p] ;
     RandomizedIndex[p] = RandomizedIndex[q] ;
     RandomizedIndex[q] = r ;
    }
    Error = 0.0 ;
  /***********************************************************
***
   * Cycle through each training pattern in the randomized order
   ***********************************************************
***/
    for( q = 0 ; q < PatternCount ; q++ ) {
     p = RandomizedIndex[q];
   /*********************************************************
****
   * Compute hidden layer activations
   ***********************************************************
***/
     for( i = 0 ; i < HiddenNodes ; i++ ) {
       Accum = HiddenWeights[InputNodes][i] ;
       for( j = 0 ; j < InputNodes ; j++ ) {
        Accum += Input[p][j] * HiddenWeights[j][i] ;
       }
       Hidden[i] = 1.0/(1.0 + exp(-Accum)) ;
      }
   /************************************************************
****
   * Compute output layer activations and calculate errors
   ************************************************************
****/
      for( i = 0 ; i < OutputNodes ; i++ ) {
       Accum = OutputWeights[HiddenNodes][i] ;
       for( j = 0 ; j < HiddenNodes ; j++ ) {
        Accum += Hidden[j] * OutputWeights[j][i] ;
       }
       Output[i] = 1.0/(1.0 + exp(-Accum)) ;
       OutputDelta[i] = (Target[p][i] - Output[i]) * Output[i] * (1.0 -
Output[i]) ;
      Error += 0.5 * (Target[p][i] - Output[i]) * (Target[p][i] - Output
[i]) ;
      }
  /*************************************************************
****
   * Backpropagate errors to hidden layer
   ************************************************************
****/
      for( i = 0 ; i < HiddenNodes ; i++ ) {
       Accum = 0.0 ;
       for( j = 0 ; j < OutputNodes ; j++ ) {
        Accum += OutputWeights[i][j] * OutputDelta[j] ;
       }
       HiddenDelta[i] = Accum * Hidden[i] * (1.0 - Hidden[i]) ;
      }
   /************************************************************
****
   * Update Inner-->Hidden Weights
   *************************************************************
***/
      for( i = 0 ; i < HiddenNodes ; i++ ) {
      ChangeHiddenWeights[InputNodes][i] = LearningRate * HiddenDelta
[i] + Momentum * ChangeHiddenWeights[InputNodes][i] ;
       HiddenWeights[InputNodes][i] += ChangeHiddenWeights
[InputNodes][i] ;
       for( j = 0 ; j < InputNodes ; j++ ) {
        ChangeHiddenWeights[j][i] = LearningRate * Input[p][j] *
HiddenDelta[i] + Momentum * ChangeHiddenWeights[j][i];
        HiddenWeights[j][i] += ChangeHiddenWeights[j][i] ;
       }
      }
   /************************************************************
****
   * Update Hidden-->Output Weights
   ************************************************************
****/
```

```
     for( i = 0 ; i < OutputNodes ; i ++ ) {
     ChangeOutputWeights[HiddenNodes][i] = LearningRate * OutputDelta
[i] + Momentum * ChangeOutputWeights[HiddenNodes][i] ;
      OutputWeights[HiddenNodes][i] += ChangeOutputWeights
[HiddenNodes][i] ;
      for( j = 0 ; j < HiddenNodes ; j++ ) {
       ChangeOutputWeights[j][i] = LearningRate * Hidden[j] *
OutputDelta[i] + Momentum * ChangeOutputWeights[j][i] ;
       OutputWeights[j][i] += ChangeOutputWeights[j][i] ;
      }
     }
    }
  /***************************************************************
****
  * Every 1000 cycles send data to terminal for display
  ***************************************************************
***/
    ReportEvery1000 = ReportEvery1000 - 1;
    if (ReportEvery1000 == 0)
    {
     Serial.println();
     Serial.println();
     Serial.print ("TrainingCycle: ");
     Serial.print (TrainingCycle);
     Serial.print (" Error = ");
     Serial.println (Error, 5);
     toTerminal();
     if (TrainingCycle==1)
     {
      ReportEvery1000 = 999;
     }
     else
     {
      ReportEvery1000 = 1000;
     }
    }
  /***************************************************************
****
  * If error rate is less than pre-determined threshold then end
  ***************************************************************
***/
    if( Error < Success ) break ;
   }
   Serial.println ();
   Serial.println();
   Serial.print ("TrainingCycle: ");
   Serial.print (TrainingCycle);
   Serial.print (" Error = ");
   Serial.println (Error, 5);
   toTerminal();
   Serial.println ();
   Serial.println ();
   Serial.println ("Training Set Solved! ");
   Serial.println ("-------");
   Serial.println ();
```

```
    Serial.println ();
    ReportEvery1000 = 1;
   // pause processor until reset
   while(1) {
   }
   }
   void toTerminal()
   {
    for( p = 0 ; p < PatternCount ; p++ ) {
     Serial.println();
     Serial.print (" Training Pattern: ");
     Serial.println (p);
     Serial.print (" Input ");
     for( i = 0 ; i < InputNodes ; i++ ) {
      Serial.print (Input[p][i], DEC);
      Serial.print (" ");
     }
     Serial.print (" Target ");
     for( i = 0 ; i < OutputNodes ; i++ ) {
      Serial.print (Target[p][i], DEC);
      Serial.print (" ");
     }
   /*************************************************************
****
   * Compute hidden layer activations
   *************************************************************
***/
     for( i = 0 ; i < HiddenNodes ; i++ ) {
      Accum = HiddenWeights[InputNodes][i] ;
      for( j = 0 ; j < InputNodes ; j++ ) {
       Accum += Input[p][j] * HiddenWeights[j][i] ;
      }
      Hidden[i] = 1.0/(1.0 + exp(-Accum)) ;   }
  /**************************************************************
***
   * Compute output layer activations and calculate errors
   *************************************************************
***/
     for( i = 0 ; i < OutputNodes ; i++ ) {
      Accum = OutputWeights[HiddenNodes][i] ;
      for( j = 0 ; j < HiddenNodes ; j++ ) {
       Accum += Hidden[j] * OutputWeights[j][i] ;
      }
      Output[i] = 1.0/(1.0 + exp(-Accum)) ;
     }
     Serial.print (" Output ");
     for( i = 0 ; i < OutputNodes ; i++ ) {
      Serial.print (Output[i], 5);
      Serial.print (" ");
     }
    }
   }
```

4.11 Considerations for Memory

As MCU in ES has very limited memory, it is important to consider memory usage for code and data. This is a non-issue for typical computer programming, thus easy to forget in ES coding. A few aspects of memory considerations are discussed here.

4.11.1 Memory Allocations

It is recommended not to use undefined size arrays in ES programming as it might continue to increase during deployment (where ES runs continuously 24/7). Available memory will decrease over time reducing performance of the system. In worst case, the MCU might run out of memory space, and leading to system deadlock. This might need total system reset (use WDT for this purpose), or else not operate at all until manual reset is performed. This will cause disruption of service.

4.11.2 Memory Leak

Another issue to worry about is memory leak. This happens when a program allocates a global memory block for some task; but it does not de-allocate it when existing the program. Next time the program is run, it might consider the previously allocated memory to be unavailable and allocate a new memory block. This is known a memory leak and can reduce performance of the system over time. If this process continues, eventually the MCU will run out of memory and the program will not be able to allocate memory leading to deadlock or improper operation.

Example 4.20 The pseudo-code below has memory lead issue. Identify where, and provide a solution of memory leak.

```
When a button is pressed:
Get some memory to remember the floor number
  Put the floor number into the memory
  Floor:  Are we already on the target floor?
    If so, open the door and wait for passengers
      Nothing else to do, so...
      Finish
    Else:   Wait until the lift is idle
      Go to the (next) required Floor
Release the memory we used to remember the floor number
Finish
```

Solution The memory leak occurs as the IF condition does not free up allocated memory for floor number before "Finish." The code only release allocated memory if the code does not fulfill IF condition. Each time IF is executed, a small memory will lead that was used to remember the floor number. A solution to this memory leak can be as follows.

```
When a button is pressed:
Get some memory to remember the floor number
  Put the floor number into the memory
  Floor:   Are we already on the target floor?
    If so, open the door and wait for passengers
      go to the Cleanup
  Else:   Wait until the lift is idle
      Go to the (next) required Floor
Cleanup:
  Release the memory used to remember the floor number
Finish
```

4.12 Computation and Communication Models of Software

Computation models define components and an execution model for computations for each component. Communication model describes the process for exchange of information between components.

4.12.1 FSM Computation Model

One of the computation models commonly used in ES is Finite State Model (FSM). FSM are models of the behaviors of a system with a limited number of defined conditions or modes, where mode transitions change with circumstances. FSM has four elements: States, Transitions, Rules or Conditions, and Input events. There are two main types of FSM: Moore and Mealy. Moore FSM outputs depend on current state only. Mealy FSM outputs depend on current state and inputs. This model describes various states and data flow where data flow of this model can be flow of data in different computation units of the same system or communication among a distributed system. In a distributed system, models also need to be synchronized.

In case of distributed system or communication with external devices, microcontroller must serve peripheral before the next data appears. This requires synchronization for data transfer without corruption. This can utilize control pins. One simple approach is Baud rate where both transmitter and receiver settle at a predefined data transfer rate. Another approach is blind cycle counting synchronization. Gadfly, Busy-wait, or handshaking mechanism can also be used. Handshaking can be two types: Single handshaking, and Double handshaking.

Asynchronous or non-blocking message passing might be needed if synchronization is not possible (e.g., no control pins or rate not negotiated). In this case, sender does not have to wait until message has arrived. Receiver uses FIFO buffer to receive data. However, buffer overflow might occur if the receiver is slow to retrieve data from FIFO buffer. Alternatively, this can be implemented without buffer, called Rendezvous, where sender will wait sometime for receiver to retrieve data before sending the next data. This is simpler as no buffer required, but reduced performance as the transmission must have some delay between transmissions. Another approach is Extended Rendezvous, where explicit acknowledgment from receiver is required. Sender waits until it receives acknowledgment from receiver of the previous message reception before it transmits the next data.

Synchronization can also be important for internal purpose (within the ES system). This occurs with one master clock. The specification for communication hardware synchronization should be <10 μs, for OS kernel 10 μs to 100 μs, and for application level 500 μs to 5 ms.

4.12.2 Recursion

Recursion is process where a program calls itself. There are three types of recursion: Linear, Tail, and Binary. In order for recursive function to finish, there must a situation where a direct result is generation, called End Condition. It is important to carefully consider termination condition of recursion, otherwise it might lead to race condition or out of memory issue.

4.13 Considerations for ES with OS

For complex systems and processes, OS-based ES is more suitable. There are some additional considerations for this type of system. Some important aspects of this type of ES are discussed in this section.

4.13.1 Thread Scheduler

If multiple programs need to be executed simultaneously, multitasking approach is commonly used by OS. OS uses thread scheduler to manage this. Scheduler type can be Round Robin, Weighted Round Robin, or Priority based. Performance of thread scheduler is measured with utilization of CPU clocks and resources (such as memory), latency for tasks, and bandwidth. Based on thread scheduler operation, threads can be put in Run state where the thread is executing currently, Active state where the thread is ready to run, but waiting it's turn, and Blocked state where the thread is waiting for an external event or availability of needed resource. Each thread has its own stack. Thread control block (TCB) contains the needed information such as stack pointer (SP) value, next or previous links, ID number, sleep counter, blocked pointer, and priority. Thread scheduler controls which thread will be run in sequence, which threads will be blocked, and if there is any thread dependency.

Thread scheduler can use Round Robin Scheduling. One approach is to assign thread times in a Rate Monotonic manner, where each thread requires to be executed are given equal amount of time in round robin fashion. This is a static type scheduling. Priority can be given based on period of time for data arrival, or maximum execution time. For priority-based approach, thread with the highest priority is executed first. There is no synchronization between tasks. For real-time ES, priority can also be assigned to tasks requiring hard real-time operation, then tasks with soft real-time operation, and then the remaining tasks. This minimizes the latency on real-time tasks. One approach to implement this is to assign dollar cost for delay and minimize cost. Another approach is earliest deadline first, which is a dynamic scheduling. Scheduling can also be based on smallest slack time first, which is also dynamic. Slack time is calculated by subtracting work left on a task from time to deadline.

One issue that might result from priority scheduling is that some tasks might never be allocated. This is called "Starvation." One solution for starvation is to assign a weightage of aging. As threads wait for scheduled, aging value will increase and higher aged threads will have increased priority.

4.13.2 Multithreading

Multithreading is a common process in OS. Thread based multiprocessing sometimes access shared global variables. This access to global variables by multiple threads might lead to race conditions. To avoid these, mutual exclusion (or Mutex) is required. However, mutex might lead to deadlocks also. Thus, careful consideration of mutex usage and avoiding deadlocks is essential, however might lead to performance penalties.

4.13.3 Multitasking

OS-based system can allow multitasking by utilizing thread scheduler. All programs, by nature of programming language, has threads. In consideration of thread execution, threads can be foreground or background. In terms of time intervals, threads can be Periodic which executes at regular intervals (e.g., ADC, motor control), Aperiodic which executes without any regular interval, but execution is frequent (e.g., car speed detection from wheel turning), and Sporadic which executes without any regular interval and infrequently (e.g., keypress interrupt, fault, error condition). Multitasking thread scheduler can be pre-emptive or cooperative.

4.13.3.1 Pre-emptive Multitasking:

In this type, OS has the flexibility to shift from process to process by time-slicing approach. Threads are suspended by a periodic interrupt. Intervals for each thread is determined by the OS based on some other factors, such as priority for access to system resources, and intervals are implemented with timer interrupts. OS maintains the context for each process, i.e., registers, memory allocation, etc. The thread scheduler chooses a new thread to run. Code below shows 3 threads, where OS can decide to switch thread wherever it wants (based on timer interrupt).

void Thread1(void){	void Thread2(void){	void Thread3(void){
Initialization1();	Initialization1();	Initialization1();
while(1){	while(1){	while(1){
Task1();	Task2();	Task3();
}	}	}
}	}	}

4.13.3.2 Cooperative (Non-Preemptive) Multitasking

In this type, threads themselves decide when to stop running and allowing other threads to be scheduled. An example is shown below where *OS_Suspend()* indicates only when scheduler is allowed to switch to a different thread.

void Thread1(void){	void Thread2(void){	void Thread3(void){
Initialization1();	Initialization1();	Initialization1();
while(1){	while(1){	while(1){
Task1();	Task2();	Task3();
OS_Suspend();	**OS_Suspend();**	**OS_Suspend();**
}	}	}
}	}	}

4.13.4 Reentrant

A program segment is reentrant if it can be executed by two (or more) threads. As it might use the same variables in registers and stack, it might lead to deadlock. A non-reentrant subroutine can have a section of code with atomic locking which is a vulnerable window or a critical section. However, it is imperative that reentrants must not have atomic write sequence, otherwise deadlock might occur. Still, error can occur when one thread calls the non-reentrant subroutine, which the critical section (if non-atomic) is interrupted by a second thread, and the second thread calls the same subroutine.

4.13.5 *Thread Dependency*

Threads might have dependency with one or more threads. Dependent thread must be scheduled after the predecessor thread is complete and sequence constraints are met. A dependency graph shows these tasks dependency. If a task *v2* is dependent directly after the task *v1*, then *v2* is dependent task and *v1* is immediate predecessor of *v2*, while *v2* is an immediate successor of *v1*.

In addition to task dependency, a dependency graph can also show timing dependencies [5]. This additional timing information can specify arrival time, deadline, and other timing information. Dependency graph can also show shared resources such as shared memory or shared bus.

4.14 Shared Memory Issues

Multithread programs can communicate though shared memory. They can be in the form of shared global variables or pointers, mailbox, FIFO queues, or messages. However, shared memory can cause race conditions, producing inconsistent results. To resolve this Mutex, atomic, and semaphores techniques need to be utilized.

4.14.1 *Atomic Sections*

Critical sections of the program must be made atomic such that sections are not interrupted or resources are locked while the critical section is being executed. In this sections, exclusive access to shared resource (e.g., shared memory) must be guaranteed. As write access can change the value of the shared resource, it is important to have the section atomic. This can be applied to shared variables or I/O ports. However, read access creates two copies of shared information, one is the original in memory and the other is temporary copy in register. They need to be consistent.

To make a critical section atomic, disable all interrupts at the beginning of the section. In addition, lock the scheduler itself so that no other foreground threads can run. However, this might not be possible in many cases where mutex semaphore needs to be used that blocks other threads trying to access the shared resource or information. This will inevitably delay other non-related operations.

4.14.2 *Semaphore*

Atomic section with disabling interrupt might not be possible for many cases. Semaphores are used to lock resources for this purpose. To protect race-free access to shared memory S, P(S) and V(S) are semaphore operations to lock and unlock the resource, respectively. These allows mutex operation and perform lock on resources. Tasks requiring S might issue these locks. The tasks will implement semaphore like this:

```
        task a {
 ..
 P(S) //obtain lock
         ..  // critical section
 V(S) //release lock
}
        task b {
 ..
 P(S) //obtain lock
         ..  // critical section
 V(S) //release lock
}
```

P represents WAIT (from Dutch word proberen). V represents SIGNAL (from Dutch word verhogen). These are executed with *OS_WAIT* and *OS_SIGNAL*, respectively. Semaphores can be Binary type or Counter type. Binary type represents 1 (Free or event occurred), or 0 (Busy or event not occurred). Counter type represents a number corresponding to available copy of resources. Each thread decrements the number representing the number of threads locking the resource. If the counter is 0, then all of the resources are busy. It can be used for space left in FIFO or number of printers available.

Example 4.21 Using pseudo-code, describe Readers-Writers problem with semaphore.

Solution The reader threads (1) Execute ROpen(file), (2) Read information from file, and (3) Execute RClose(file). The writer threads (1) Execute WOpen(file), (2) Read information from file, (3) Write information to file, and (4) Execute WClose(file). Below is a pseudo-core representation of the Readers-Writers problem, where ReadCount = 0 is a number, mutex = 1 is a semaphore, and wrt = 1 is another semaphore.

```
ReadCount, number of Readers that are open
mutex, semaphore controlling access to ReadCount
wrt, semaphore is true if a writer is allowed access
ROpen
 wait(&mutex);
 ReadCount++;
 if(ReadCount==1) wait(&wrt);
 signal(&mutex);
RClose
 wait(&mutex);
 ReadCount--;
 if(ReadCount==0) signal(&wrt);
 signal(&mutex);
```

```
WOpen
 wait(&wrt);
WClose
 signal(&wrt);
```

Semaphores can be implemented with Spin-lock Binary, Spin-lock Counting, Cooperative Spin-lock, or Blocking Semaphore approaches. Blocking semaphore recaptures time lost in spin operation of spin-lock, thus eliminates wasted time running threads that are not doing any useful work. It is implemented with bounded waiting where once thread calls WAIT and is not services, there is only a finite number of threads that will go ahead.

4.14.3 Waiting and Timeouts

Another important items to consider is waiting and timeouts. The waiting should be bounded. Otherwise, timeout should occur. This provides a solution to deadlock detection. It can be implemented with Wait-for-graph or resource allocation graph. This approach works well for single instance resources.

4.15 Software Performance Measure

Some metrics used for software performance measures are:

1. ***Breakdown Utilization (BU):*** This represents the percentage of resource utilization below which the RTOS can guarantee that all deadlines will be met.
2. ***Normalized Mean Response Time (NMRT):*** This is the ratio of the "best case" time interval a task becomes ready to execute and then terminates, and the actual CPU time consumed.
3. ***Guaranteed ratio (GR):*** For dynamic scheduling, the number of tasks whose deadlines can be guaranteed to be met versus the total number of tasks requesting execution.

Exercise

Problem 4.1: Write an Arduino Uno code for Morse Code of "SOS" by blinking the internal LED (connected to Pin 13).

Problem 4.2: Write an Arduino Uno code for Temperature measurement with a thermistor. The analog data from the sensor must be converted to Fahrenheit and Centigrade. Display the temperature readings on an LCD display.

Problem 4.3: What are different queueing mechanisms in ES? Discuss relative advantages and disadvantages. Provide two examples where each of them is more suitable.

Problem 4.4: Describe the 6 sleeping modes of Arduino Uno in order of lower current consumption. Why interrupt is suitable for sleeping modes?

Problem 4.5: What is time service? What module you will need for time service in Arduino Uno?

Problem 4.6: Write a bitmath code to set the digital pin 5 to output mode without changing any other pin modes. Write another code using bitmath to toggle the pin 5 output value.

Problem 4.7: Write a code for a traffic light (using Red, Yellow, Green LEDs) and a pedestrian switch (using a push button). First implement the code with *delay* function, then implement with Timer2 and PCINT interrupt, with lowest current sleep mode.

Problem 4.8: What are two main type of software buffer? Describe relative advantages and disadvantages.

Problem 4.9: Write a C code that utilizes timer/counter hardware of ATmega328 of Arduino Uno, employs extended standby sleep mode, and generates an output pulse of 10 Hz through pin 13.

Problem 4.10. Write a code to implement a IIR LPF for a push button input that turns on an LED.

Problem 4.11: Write a C code that implements ANN machine learning (ML) algorithm to first train with a set of given data in the code, then runs the ANN ML to monitor 3 analog input values (A0–A2) to turn ON/OFF two LEDs (connected to D6–D7).

References

1. E.A. Lee, "Cyber physical systems: design challenges," 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), Orlando, FL, 2008, pp. 363–369
2. B. Tabbara, A. Tabbara, A. Sangiovanni-Vincentelli, *Function/Architecture Optimization and Co-Design of Embedded Systems* (Kluwer Academic Publishers, 2000)
3. J. Stankovic, Misconceptions about real-time computing. IEEE Computer **21**, 10 (1988)
4. Atmel ATmega328 datasheet. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf
5. P. Marwedel, Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems (2011)