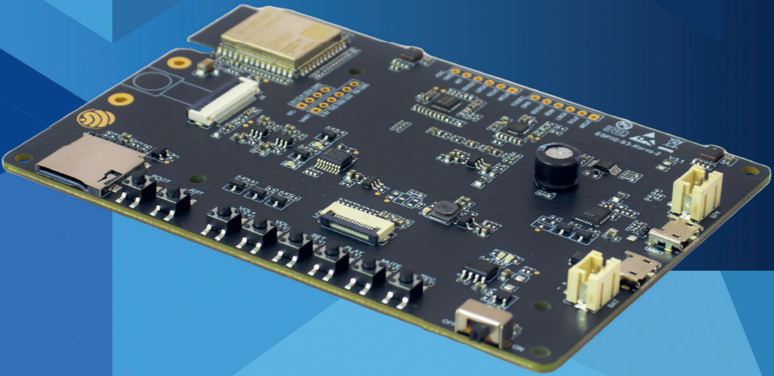
SECOND EDITION

Introduction to the IoT



iot-open.eu RELOADED

Co-funded by the European Union

Introduction to the IoT

(Internet of Things)

Coursebook
Second Edition

RTU Press
Riga 2025

Introduction to the IoT. Coursebook. Second Edition. Riga: RTU Press, 2025. 416 pp.

This book and its offshoots have been prepared to provide comprehensive information about the Internet of Things. It is intended to introduce IoT to undergraduate and graduate students, technology enthusiasts, and engineers who want to expand their current knowledge with the latest hardware and software achievements in the field of IoT.
This book is also intended for teachers and educators who want to prepare a course on IoT.

Scientific Editor	Agris Ņikitenko
Technical Editor	Marta Ņikitenko, Piotr Czekalski
Graphic Design	Blanka Czekalska, Piotr Czekalski
Images	Blanka Czekalska, Piotr Czekalski
Technical Support	Eryk Czekalski
Published by	RTU Press

https://www.roboticlab.eu/homelab

This book is compiled by DokuWiki and proofread using Grammarly. This approach speeds up the publishing process and more than likely will be generally used in future to prepare books for publishing.

ISBN 978-9934-37-137-0 (print)
ISBN 978-9934-37-138-7 (pdf)

Table of Contents

Authors

IOT-OPEN.EU Reloaded Consortium partners proudly present the 2nd edition of the Introduction to the IoT book. The complete list of contributors is listed below.

ITT Group

- Raivo Sell, Ph. D., ING-PAED IGIP
- Rim Puks, Eng.
- Mallor Kingsepp, Eng.

Riga Technical University

- Agris Nikitenko, Ph. D., Eng.
- Karlis Berkolds, M. sc., Eng.
- Anete Vagale, M. sc., Eng.
- Rudolfs Rumba, M. sc., Eng.

Silesian University of Technology

- Piotr Czekalski, Ph. D., Eng.
- Krzysztof Tokarz, Ph. D., Eng.
- Godlove Suila Kuaban, Ph. D., Eng.
- Oleg Antemijczuk, M. sc., Eng.
- Jarosław Paduch, M. sc., Eng.

Tallinn University of Technology

- Raivo Sell, Ph. D., ING-PAED IGIP
- Karl Läll, B. sc., Eng.

SIA RobotNest

- Karlis Berkolds, M. sc., Eng.

IT Silesia

- Łukasz Lipka, M. sc., Eng.

University of Messina

- Salvatore Distefano
- Rustem Dautov
- Riccardo Di Pietro
- Antonino Longo Minnolo

ITMO University

- Aleksandr Kapitonov, Ph. D., Assoc. Prof.
- Dmitrii Dobriborsci, M. sc., Eng.
- Igor Pantiukhin, M. sc., Eng.
- Valerii Chernov, Eng.

Graphic Design and Images

- Blanka Czekalska, M. sc., Eng., Arch.
- Piotr Czekalski, Ph. D., Eng.

Technical Correction

- Eryk Czekalski

Reviewers (1st edition)

- Fabio Bonsignorio, Ph. D., Eng.– Professor at Scuola Superiore Sant'Anna, Institute of Biorobotics
- Artur Pollak, M. sc., Eng. – CEO at APAGroup
- Ivars Parkovs, M. sc., Eng. – R&D Senior Engineer at “SAF Tehnika” Ltd.
- Janis Lacaunieks, M. sc., Eng. – R&D Engineer at “SAF Tehnika” Ltd.

Preface

This book and its offshoots were prepared to provide comprehensive information about the Internet of Things on the engineering level.
Its goal is to introduce IoT to bachelor students, master students, technology enthusiasts and engineers willing to extend their current knowledge with the latest hardware and software achievements in the scope of the Internet of Things.
This book is also designated for teachers and educators willing to prepare a course on IoT.

We (Authors) assume that persons willing to study this content possess some general knowledge about IT technology, e.g. understand what an embedded system is, know the general idea of programming (in C/C++) and are aware of wired and wireless networking as it exists nowadays.

This book constitutes a comprehensive manual for IoT technology; however, it is not a complete encyclopedia nor exhausts the market. The reason for it is pretty simple – IoT is so rapidly changing technology that new devices, ideas and implementations appear daily. Once you read this book, you can quickly move over the IoT environment and market, easily chasing ideas and implementing your IoT infrastructure.

We also believe this book will help adults who took their technical education some time ago to update their knowledge.

We hope this book will let you find brilliant ideas in your professional life, see a new hobby, or even start an innovative business.

Playing with real or virtual hardware and software is always fun, so keep going!

Project Information

This content was implemented under the following projects:

- Strategic Partnerships in the Field of Education, Training, and Youth – Higher Education, 2016, IOT-OPEN.EU – Innovative Open Education on IoT: Improving Higher Education for European Digital Global Competitiveness, project number: 2016-1-PL01-KA203-026471,
- Cooperation Partnerships in higher education, 2022, IOT-OPEN.EU Reloaded: Education-based strengthening of the European universities, companies and labour force in the global IoT market, project number: 2022-1-PL01-KA220-HED-000085090,
- Horizon 2020 Research Innovation and Staff Exchange Programme (RISE) under the Marie Skłodowska-Curie Action, Programme H2020-EU.1.3.3. - Stimulating innovation by means of cross-fertilisation of knowledge, Grant Agreement No 871163: Reactive Too - Reliable Electronics for Tomorrow’s Active Systems.
- International project co-financed by the program of the Minister of Science and Higher Education entitled “PMW” in the years 2021 - 2025; contract no. 5169/H2020/2020/2

1. Introduction

🎓P 🎓B 🎓E

Here comes the Internet of Things. The name that recently makes red-hot people in business, researchers, developers, geeks and … students. The name that non-technology related people consider a kind of magic and even a danger to their privacy. The EU set the name as one of the emerging technologies and estimated the worldwide market will hit well over 500 billion US dollars in 2022, while the number of IoT devices in 2030 is expected to be around 3.2 billion.

What is IoT (Internet of Things), then? Surprisingly, the answer is not straightforward.

Content classification hints

The book composes a comprehensive guide for a variety of education levels. A brief classification of the contents regarding target groups may help in a selective reading of the book and ease in finding the correct chapters for the desired education level. To inform a reader about the proposed target group, icons are assigned to the chapters level 1 (top) and 2nd level chapters. The list of icons and their reflection on the target groups is presented in the table 1.

Table 1: List of icons presenting content classification and corresponding target groups

Icon	Target group
🎓P	General **P**ublic audience: all those who want to get familiar with basic concepts but do not necessarily step into technical details.
🎓B	**B**achelor and Engineering level students
🎓M	**M**asters students
🎓E	**E**nterprise, VETS and technical

1.1. Definition of IoT

Let us roll back to the 1970s first. In 1973, the first RFID device was patented. This device was the key enabling technology even if it does not look nor remind modern IoT devices. The low power (actually here passive) solution with a remote antenna large enough to collect energy from the electromagnetic field and power the device brought an idea of uniquely identifiable items. That somehow mimics well-known EAN barcodes and the evolution used nowadays, like QR codes, but every single thing has a different identity here. In contrast, EAN barcodes present a class of products, not an individual one. The possibility to identify a unique identity remotely became fundamental to the IoT as it's known today. RFID is not the only technology standing behind IoT. In the 1990s, the rapid expansion of wireless networks, including broadband solutions like cellular-based data transfers with their consequent generations, enabled connecting devices in various, even distant, geographical locations. Parallelly, an exponential increase in the number of devices connected to the global Internet network was observed, including the smartphone revolution that started around the first decade of the XXI century. On the hardware level, microchips and processors became physically smaller and more energy efficient yet offering growing computing capabilities and memory size increase, along with significant price drops. All those facts drove the appearance of small, network-oriented, cheap and energy-efficient electronic devices. In recent years, the development of efficient AI technologies has even boosted IoT applications.

What is IoT?

The phrase “Internet of Things” was used for the first time in 1999 by Kevin Ashton – an expert on digital innovation. Formally, IoT was introduced by the International Telecommunication Union (ITU) in the ITU Internet report in 2005 [1]. The understanding and definitions of IoT have changed over the years, but now all agree that this cannot be seen as a technology issue only. According to IEEE “Special Report: Internet of Things” [2] released in 2014, IoT is:

IEEE Definition of IoT
A network of items – each embedded with sensors – connected to the Internet.

It relates to the physical aspects of IoT only. The Internet of Things also addresses other aspects that cover many areas [3]:

- enabling technologies,
- software,
- applications and services,
- business models,
- social impact,
- security and privacy aspects.

IEEE, as one of the most prominent standardisation organisations, also works on standards related to the IoT. The primary document is IEEE P2413™ [4]. It covers the technological architecture of IoT as three-layered: sensing at the bottom, networking and data communication in the middle, and applications on the top. It is essential to understand that IoT systems are not only small, local-range systems. ITU-T has defined IoT as:

ITU-T Definition of IoT
A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies.

In the book [5] by the European Commission, we can read a similar description of what IoT is: "The IoT is the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment." IoT impacts many areas of human activity: manufacturing, transportation, logistics, healthcare, home automation, media, energy saving, environment protection and many more. In this course, we will consider the technical aspects mainly.

Thing

In the IoT world, the "thing" is always equipped with some electronic element that can be as simple as the RFID tag, an active sensor sending data to the global network, or an autonomous device that can react to environmental changes. In CERP-IoT book "Visions and Challenges" [6] in the context of "Internet of Things" a "thing" could be defined as:

CERP-IoT Definition of "Thing"
A real/physical or digital/virtual entity that exists and moves in space and time and can be identified. Assigned identification numbers, names and location addresses commonly identify things.

It is quite easy to find other terms used in the literature like "smart object", "device", or "nodes" [7].

Passive Thing

One can imagine that almost everything in our surroundings is tagged with an RFID element. They do not need a power supply; they respond with a short message, usually containing the identification number. Modern RFID can achieve 6 to 7 meters of the range. Using the active RFID reader, we can quickly locate lost keys and know if we still have the butter in the fridge and in which wardrobe there is our favourite t-shirt.

Active Thing

If the "thing" includes the sensor, it can send interesting data about current conditions. We can sense environmental parameters like temperature, humidity, air pollution, pressure, localisation data, water level, light, noise, and movement. Using different methods and protocols, this data can be sent to the central collector that connects to the Internet and the database or cloud. There, the data can be processed, and Artificial Intelligence algorithms can be used to decide actions that could be taken in different situations. Active things can also receive control signals from the central controller to control the environment: turn on/off the heating or light, water flowers, and turn on the washing machine when there is enough sunlight to generate the required electricity or charge your electric car.

Autonomous Thing

This thing does not even require the controller to make the proper decision. An autonomous vacuum cleaner can clean our house when it detects that we aren't home and the floor needs cleaning. The fridge can order our favourite beverage once the last bottle is empty.

Sensor Network

Sensor Networks are a subset of the IoT devices used as a collaborative solution to grab data and send it for further processing. Opposite to the general IoT devices, Sensor Network devices do not have any actuators that can apply an action to the external world. The data flow is unidirectional, then.

IoT vs Embedded Systems

IoT systems and embedded systems share almost the same domain. They frequently use the same microcontrollers, sensors and actuators, development software and even programming models. What differs between IoT and embedded systems is that IoT, on its principles, uses communication to send and receive data outside of its instance. Embedded systems do not have to be network-enabled or have a unique identity, while IoT devices do. Moreover, IoT systems are complex and multilayered, often introducing cloud-based parts, while embedded systems are stand-alone devices. Indeed, one can say that an IoT device is a network-enabled embedded system.

1.2. Future of IoT

While the IoT, as a technical solution paradigm to certain problems, is relatively well understood and applied, it keeps evolving to address a more complex and diverse spectrum of tasks. It follows the trends of AI integration and advancement into hardware systems, providing close-to-problem AI capabilities - edge AI. Therefore, the new AI-enabled IoT systems provide low-latency intelligent solutions capable of online or offline training through applications of machine learning techniques. Regarding model flexibility, the microprocessors with dedicated AI co-processing or GPU (Graphical Processing Unit) have limited memory and processing speed; they significantly extend application possibilities. A good example is modern cell phones, capable of complex AI-drive tasks like photo or video editing, content generation, and other functions. The same techniques might be applied for real-time data analysis, decision-making and other IoT-specific applications.

1.3. Enabling Technologies

In this chapter, there is an approach to describe modern technologies that appeared in the last few years, enabling the idea of IoT to be widely implementable. In the [8], one can read that “The confluence of efficient wireless protocols, improved sensors, cheaper processors and a wave of startups and established companies made the concept of the IoT mainstream”. Similar analysis has been done in [9] where authors write that “the latest developments in RFID, smart sensors, communication technologies and Internet protocols enable the IoT”. RFID and smart sensors need the microprocessor system to read, convert the data into digital format, and send it to the Internet using the communication protocol. This process can be done by small- and medium-scale computer (embedded) systems. These are essential elements of technologies used in IoT systems.

Edge class devices

In recent years, one can observe rapid growth in microprocessors. It includes not only the powerful desktop processors but also microcontrollers – elements that are used in small-scale embedded systems. We can also notice the popularity of microprocessor systems that can be easily integrated with other factors, like sensors and actuators, connected to the network. Essential is also the availability of programming tools and environments supported by different companies and communities. An excellent example of such a system is Arduino. Those devices are low-power, constrained devices, usually battery-powered and, in most cases, communicating wirelessly.

Fog class devices

The same growth can be observed in the advanced constructions comparable to low-end computers. They have more powerful processors, memory and networking connectivity built in than small-scale computer systems. They can work under the control of multitasking operating systems like Linux and Windows and embedded or real-time operating systems like FreeRTOS. Having many libraries, they can successfully work as hubs for local storage, local controllers and gateways to the Internet. Raspberry Pi and the nVidia Jetson series are examples of such systems. This category of devices frequently contains hardware accelerated (such as GPU) AI-capable solutions, e.g. nVidia Jetson Nano or Xavier series. Those devices can be battery or mains-powered. Often, they are green energy powered: e.g. with a larger backup battery and energy harvesting solution (such as solar panel).

Access to the Internet

Nowadays, the Internet is (almost) everywhere. There are lots of wireless networks available in private and public places. The price of cellular access (3G/4G/5G) is low, offering a suitable data transfer performance. Connecting the “thing” to the Internet has never been so easy.

IP Addressing Evolution

The primary paradigm of IoT is that every unit can be individually addressed. With the addressing scheme used in IPv4, it wouldn't be possible. IPv4 address space delivers “only” 4 294 967 296 of unique addresses (2^{32}). If you think it's a considerable number, imagine that every person in the world has one IP-connected device – IPv4 covers about half of the human population. The answer is IPv6 with a 128-bit addressing scheme that gives 3.4×10^{38} addresses. It will be enough even if everyone has a billion devices connected to the Internet.

Data Storage and Processing

IoT devices generate the data to be stored and processed somewhere. If there are a couple of sensors, the amount of data is not very big, but if there are thousands of sensors generating data hundreds of times every second. The cloud can handle it – the massive place for the data with tools and applications ready to help with data processing. Some big, global clouds are available for rent, offering storage, Business Intelligence tools, and Artificial Intelligence analytic algorithms. There are also smaller private clouds created to cover the needs of one company only. Many universities have their own High-Performance Computing Centre.

Mobile Devices

Many people want to be connected to the global network everywhere, anytime, having their “digital twin” with them. It is possible now with small, powerful mobile devices like smartphones. Smartphones are also elements of the IoT world, being together sensors, user interfaces, data collectors, wireless gateways to the Internet, and everything with mobility features.

The technologies we mentioned here are the most recognisable. Still, there are many others, more minor, described only in the technical language in some standard description document, hidden under the colourful displays between large data centres, making our IoT world operable. In this book, we will describe some of them.

A special note on Fog class and Edge class devices

Technology development instantly shifts devices between categories. A border between Fog and Edge class devices is conventional; many can share both worlds. It depends on their purpose, application and performance configuration; thus, Raspberry Pi can be an end-node (Edge) class device and a Fog class, working as a data aggregator and analytical device.

1.4. Mobility – New Paradigm for IoT Systems

IoT is a network of physical things or devices that might include sensors or simple data processing units, complex actuators, and significant hybrid computing power. Today, IoT systems have transitioned from being perceived as sensor networks to smart-networked systems capable of solving complex tasks in mass production, public safety, logistics, medicine and other domains, requiring a broader understanding and acceptance of current technological advancements, including advanced AI data processing.

Since the very beginning of sensor networks, one of the main challenges has been data transport and data processing, where significant efforts have been put by the ICT community towards service-based system architectures. However, the current trend already provides considerable computing power, even for small mobile devices. Therefore, the concepts of future IoT already shifted towards more innovative and more accessible IoT devices, and data processing has become possible closer to the Fog and Edge.

Cloud Computing

Cloud-based computing is a relatively well-known and adequately employed paradigm where IoT devices can interact with remotely shared resources such as data storage, data processing, data mining, and other services are unavailable to them locally because of the constrained hardware resources (CPU, ROM, RAM) or energy consumption limits. Although the cloud computing paradigm can handle vast amounts of data from IoT clusters, the transfer of extensive data to and from cloud computers presents a challenge due to limited bandwidth[10]. Consequently, there is a need to process data near data sources, employing the increasing number of smart devices with enormous processing power and a rising number of service providers available for IoT systems.

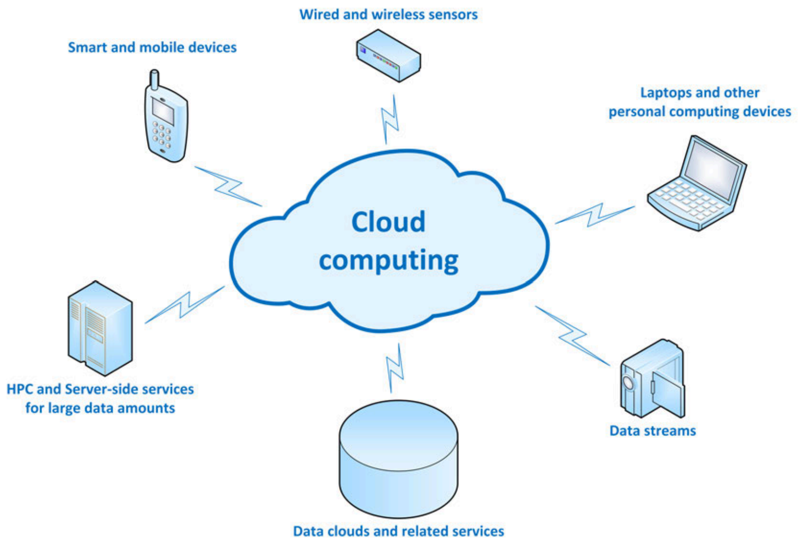

Wired and wireless sensors
Smart and mobile devices
Laptops and other personal computing devices
Cloud computing
HPC and Server-side services for large data amounts
Data streams
Data clouds and related services

Figure 2: Cloud IoT system' architecture

Fog Computing

Fog computing addressed the bottlenecks of cloud computing regarding data transport while providing the needed services to IoT systems. Fog computing is a new trend in computing that aims to process data near the source. It pushes applications, services, data, computing power, and decision-making away from the centralized nodes to the logical extremes of a network. Fog computing significantly decreases the data volume that must be moved between end devices and the cloud. Fog computing enables data analytics and knowledge generation at the data source. Furthermore, the dense geographic distribution of fog helps to attain a better-localised accuracy for many applications than the cloud processing of the data [11].

The recent development of energy-efficient hardware with AI acceleration enters the fog class of the devices, putting Fog Computing in the middle of the interest of IoT application development and extending new horizons to them. Fog Computing is more energy efficient than raw data transfer to the cloud and back, and in the current scale of the IoT devices, the application is meant for the future of the planet Earth. Fog computing usually also has a positive impact on IoT security, e.g., sending preprocessed and depersonalized data to the cloud and providing distributed computing capabilities that are more attack-resistant.

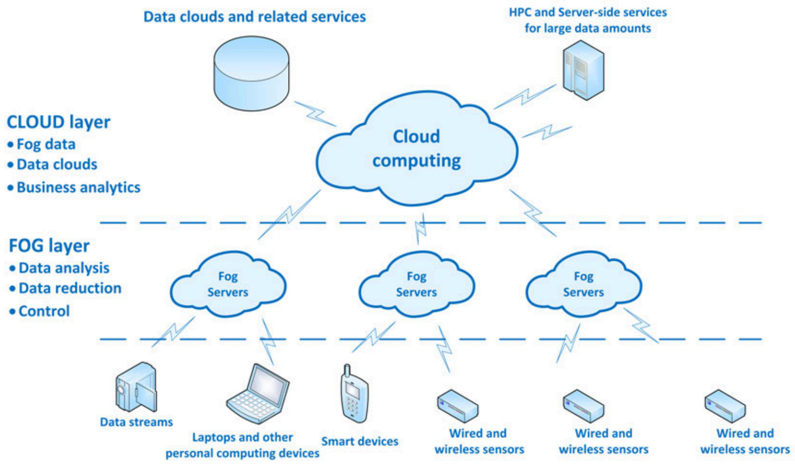

Data clouds and related services
HPC and Server-side services for large data amounts
CLOUD layer
• Fog data
• Data clouds
• Business analytics
Cloud computing
FOG layer
• Data analysis
• Data reduction
• Control
Fog Servers
Fog Servers
Fog Servers
Data streams
Laptops and other personal computing devices
Smart devices
Wired and wireless sensors
Wired and wireless sensors
Wired and wireless sensors

Figure 3: Fog IoT system' architecture

Edge Computing

Recent developments in hardware, power efficiency, and a better understanding of IoT data nature, including privacy and security, led to solutions where data is processed and preprocessed right to their source in the Edge class devices. Edge data processing on end-node IoT devices is crucial in systems where privacy is essential and sensitive data is not to be sent over the network (e.g. biometric data in a raw form). Moreover, distributed data processing can be considered more energy efficient in some scenarios where, e.g. extensive, power-consuming processing can be performed during green energy availability.

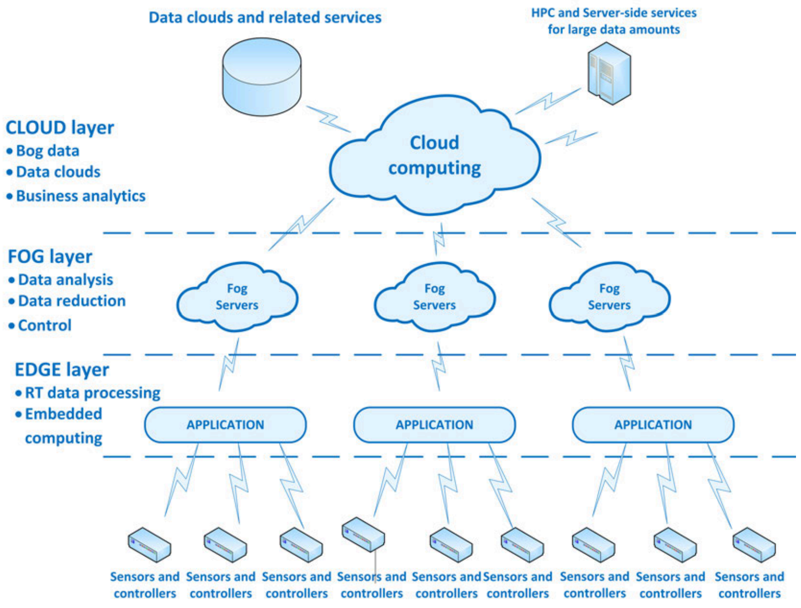
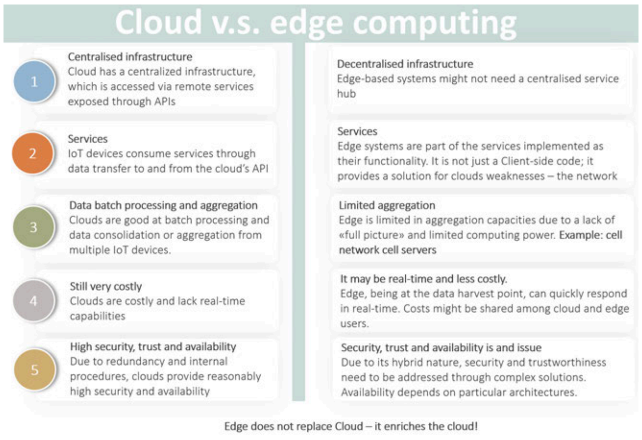


Figure 4: Edge IoT system' architecture

While Cloud, Fog, and Edge systems might seem the same to the end user from a functionality perspective, they are very different and provide different performance, scalability, and computing capabilities, which are emphasized in the following comparison.



Cloud v.s. edge computing

1	Centralised infrastructure Cloud has a centralized infrastructure, which is accessed via remote services exposed through APIs	Decentralised infrastructure Edge-based systems might not need a centralised service hub
2	Services IoT devices consume services through data transfer to and from the cloud's API	Services Edge systems are part of the services implemented as their functionality. It is not just a Client-side code; it provides a solution for clouds weaknesses – the network
3	Data batch processing and aggregation Clouds are good at batch processing and data consolidation or aggregation from multiple IoT devices.	Limited aggregation Edge is limited in aggregation capacities due to a lack of «full picture» and limited computing power. **Example: cell network cell servers**
4	Still very costly Clouds are costly and lack real-time capabilities	It may be real-time and less costly. Edge, being at the data harvest point, can quickly respond in real-time. Costs might be shared among cloud and edge users.
5	High security, trust and availability Due to redundancy and internal procedures, clouds provide reasonably high security and availability	Security, trust and availability is and issue Due to its hybrid nature, security and trustworthiness need to be addressed through complex solutions. Availability depends on particular architectures.

Edge does not replace Cloud – it enriches the cloud!

Figure 5: Differences between Cloud and Edge IoT systems

Cognitive IoT Systems

According to [12], Cognitive IoT, besides a proper combination of hardware, sensors and data transport, comprises cognitive computing, which consists of the following main components:

- **understanding** – in the case of IoT, it means systems' capability to process a significant amount of structured and unstructured data, extract the meaning of the data – produce a model that binds data to reality,
- **reasoning** – involves decision-making according to the understood model and acquired data,
- **learning** – creating new knowledge from the existing, sensed data and elaborated models.

Usually, cognitive IoT systems or C-IoT are expected to add more resilience to the solution. Resilience is a complex term and is differently explained under different contexts; however, there are standard features for all resilient systems. As a part of their resilience, C-IoT should be capable of self-failure detection and self-healing that minimises or gradually degrades the system's overall performance. In this respect, the non-resilient system fails or degrades in a step-wise manner. In case of security issues, that system should be able to change its security keys and encryption algorithms and take other measures to cope with the detected threats. Self-optimisation abilities are often considered part of the C-IoT feature list to provide more robust solutions. Recent developments in the Fog and Edge class devices and the efficient software leverage cognitive IoT Systems to a new level.

All three approaches, from cloud to cognitive systems, focus on adding value to IoT devices, system users and related systems on-demand. Since market and technology acceptance of mobile devices is still growing, and the amount of produced data from those devices is growing exponentially, mobility as a phenomenon is one of the main driving forces of the technological advancements of the near future.

1.5. Data Management Aspects in IoT

Data management is a critical task in IoT. Due to the high number of devices (things) already available, that is tens of billions. Considering the data traffic ge nerated by each of them through, e.g. sensor networks, infotainment (soft news) or surveillance systems, mobile social network clients, and so on, we are now even beyond the ZettaByte (ZB 2^{70}, 10^{21} bytes) era. This opened up several new challenges in (IoT) data management, giving rise to data sciences and big data technologies. Such challenges have not to be considered as main issues to solve but also as significant opportunities fuelling the digital economy with new directions such as Cloudonomics [13] and IoTonomics, where data can be considered as a utility, a commodity to manage, curate, store, and trade appropriately. Therefore, properly managing data in IoT contexts is not only critical but also of strategic importance for business players as well as for users, evolving into **prosumers** (producers-consumers).

From a technological perspective, the main aspects of dealing with IoT data management are:

- **Data source** - data generation and production is a relevant part of IoT, involving sensors probing the physical system. In a cyber-physical-social system view, such sensors could be virtual (e.g. software) or even human (e.g. citizens, crowdsensing). The main issues in data production are related to the type and format of data, heterogeneity in measurements and similar issues. Semantics is the key to solving these issues through specific standards such as Sensor Web Enablement and Semantic Sensor Networks.
- **Data collection/gathering** - once data are generated, these should be gathered and made available for processing. The collection process needs to ensure that the data collected are defined and accurate so that subsequent decisions based on the findings are valid. Some types of data collection include census (data collection about everything in a group or statistical population), sample survey (collection method that provides for only part of the total population), and administrative byproduct (data collection is a byproduct of an organization’s day-to-day operations). Usually, wireless communication technologies such as Zigbee, BlueTooth, LoRa, Wi-Fi and 3G/ 4G networks are used by IoT smart objects and things to deliver data to collection points.
- **Filtering** - is a specific preprocessing activity, usually performed at data source or data collector (IoT) nodes (e.g. motes, base stations, hotspots, gateways), aiming at cleaning noisy data, filtering noise and not helpful information.
- **Aggregation/fusion** - to reduce bandwidth before sending data to processing nodes, these are further elaborated, compressed, aggregated and fused (sensor/data fusion) to reduce the overall volume of raw data to be transmitted and stored.
- **Processing** - once data are adequately collected, filtered, aggregated, and fused, they can be processed. Processing can be local and remote and usually includes preprocessing activities to prepare data for actual processing. Local processing, when possible, is mainly tasked with a fast, lightweight computation on edges (Edge computing) and in the Fog layer, wherever possible, quickly providing results and local analytics. More complex computations are usually demanded to remote (physical or virtual) servers provided by local nodes (e.g. communication servers, cloudlets) in a Fog computing fashion or by Cloud providers as virtual machines

hosted in data centres. This kind of computation can also involve historical data, providing global analytics, but hardly meets time-constrained applications and real-time requirements.

- **Storage/archive** - remote servers are also used for permanently storing and archiving data, making these available for further processing, even to third parties. The database is often used for that, mainly based on distributed, NoSQL key-store technologies to improve reliability and performance.
- **Delivering/presentation/visualization** - processing activity results must then be delivered to requestors and users. These have to be, therefore, adequately organized and formatted, ready for end-users. IoT data visualization is becoming an integral part of the IoT. Data visualization provides a way to display this avalanche of collected data in meaningful ways that clearly present insights hidden within this mass amount of information.
- **Security and privacy** - data privacy and security are among the most critical issues in IoT data management. Good results and reliable techniques for secure data transmission, such as TLS and similar, are available. This way, IoT data security issues mainly concern [14] **securing IoT devices**, since they are usually resource-constrained and therefore do not allow to adopt traditional cryptography scheme to data encryption/decryption. **Data privacy and integrity** should also be enforced in remote storage servers, anonymizing data and allowing owners to properly manage (monitoring, removing) them while ensuring availability. Indeed, security and privacy issues vertically span the whole IoT stack. A promising technique to address IoT security issues, attracting growing interest from both academic and business communities, is blockchain [15].

1.6. IoT Application Domains

There is a rapid increase in the adoption of IoT in the various sectors (e.g., intelligent transport systems, smart health care, smart manufacturing, smart homes, smart cities, smart agriculture, and smart energy) of the society or economy. IoT technologies are being applied in the various sectors of the economy to increase efficiency, solve technical challenges, and create value to increase companies' earnings and improve user experience. The increasing adoption of IoT technologies in the various sectors of the economy or industry has made IoT technology the pillar of the fourth and fifth industrial revolutions (industry 4.0 and Industry 5.0).

Application domains of the Internet of Things solutions are vast. Some of the applications of IoT include the following[16]:

- building and home automation,
- smart water,
- internet of food,
- smart metering,
- smart city (including logistics, retail, transportation),
- industrial IoT,
- precision agriculture and smart farming,
- security and emergencies,
- healthcare and wellness (including wearables),
- smart environment,
- energy management,
- robotics,
- smart grids.

Smart Homes are one of the first examples that come to mind when discussing the Internet of Things domain applications. Smart home benefits include reduced energy wastage, the quality and reliability of devices, system security, reduced cost of basic needs, etc. Some home automation examples are environmental control systems that monitor and control heating, ventilation, air conditioning and sunscreens; electrical charging of vehicles; solar panels for electrical power and hot water; ambient lighting control, smart lighting for aquaria; home cooking and food ordering; access control (doors, garage, gate); smart plant irrigation systems (both indoors and outdoors); baby monitoring; timed pet food dispensers; monitoring perishable goods (for example, in the refrigerator); household items remote monitoring (for instance, of washer cycle status); tracking and proactive maintenance scheduling (such as e.g. electric car charging); event-triggered task execution. Home security also plays a significant role in smart homes. Examples of applications are automatic door locks, sensors for opening doors and windows, pressure, motion and infrared sensors, security cameras, notifications about security (to the owner or the police) and fitness-related applications.

In **Smart City**, multiple IoT-based services are applied to different areas of urban settings. The aim of the smart city is the best use of public resources, improvement of the quality of resources provided to people and reduction of operating costs of public administration [17]. A smart city can include many solutions like smart buildings, smart grids for improving energy management, smart tourism, monitoring of the state of the roads and occupation of parking lots, public transportation optimisation, public safety, environment monitoring, automatic street lighting, signalling with smart power devices, control of water levels for hydropower or flood warnings, electricity-generating devices like solar panels and wind turbines, weather monitoring stations.
Transportation in smart cities may include aviation, monitoring and forecasting of traffic slowdowns, timetables and current status, navigation and route planning, as well as vehicle diagnostics and maintenance reports, remote maintenance services, traffic accident information collection, fleet management using digital tachographs, smart parking, car/bicycle sharing services [18]. IoT in transportation makes cars interconnected, particularly in the approaching autonomous vehicles era.

Smart Grid is a digital power distribution system. This system gathers information using smart meters, sensors and other devices. After these data are processed, power distribution can be adapted accordingly. Smart grids deliver sustainable, economical and secure electricity supplies efficiently.

In **Precision Agriculture** and **Smart Farming** IoT solutions can be used to monitor the moisture of the soil and conditions of the plants, control microclimate conditions and monitor the weather conditions to improve farming [19]. The goal of using IoT in agriculture is maximising the harvest, reducing operational costs, being more efficient, and reducing environmental pollution using low-cost automated solutions. An interaction between the farmer and the systems can be done using a human-machine interface. In the future smart precision farming can be a solution for such challenges as increasing worldwide demand for food, a changing climate, and a limited supply of water and fossil fuels [20].

Internet of Food integrates many of the abovementioned techniques and encompasses different stages of the food delivery chain, including smart farming, food processing, transportation, storage, retail, and consumption. It provides more safety and improved efficiency at each food production and consumption stage, including reduced waste and increased transparency.

Like precision agriculture, which is part of IoT in industry, **Smart Factories** also tend to improve manufacturing by monitoring pollutant gas emissions, locating employees and with many other solutions.

Industrial IoT and **smart factories** are part of the Industry 4.0 revolution. In this model, modern factories can automate complex manufacturing tasks, thanks to the Machine-To-Machine communication model, which provides more flexibility in the manufacturing process to easily enable personalised, short-volume product manufacturing.

In the **healthcare and wellness**, IoT applications can monitor and diagnose patients and manage people and medical resources. It allows remote and continuous monitoring of the vital signs of patients to improve medical care and wellness [21]. An essential part of smart welfare is wearables, including wristbands and smartwatches that monitor the activity level, heart rate and other parameters. Smart healthcare includes remote monitoring, care of patients, self-monitoring, smart pills, smart home care, Real-Time Health Systems (RTHS) and many more. Medical robotics can also be part of the healthcare IoT system that includes medical robots in precision surgery or distance surgery; some robots are used in rehabilitation and hospitals (for example, Panasonic HOSPI [22]) for delivering medication, drinks, etc. to patients.

Wearables used in IoT applications should be highly energy efficient, ultra-low power and small-sized. Wearables are installed with sensors and software for data and information collected about the user. Devices used in daily life like Fitbit [23] are used to track people's health and exercise progress in previously impossible ways, and smartwatches allow to access smartphones using this device on the wrist. But wearables are not limited only to wearing them on the wrist. They can also be glasses equipped with a camera, a sports bundle attached to the shoes, a camera attached to the helmet, or a necklace [24].

Smart supply chains integrate IoT and other modern information and communication technologies to manage supply chain systems and facilitate the flow of raw materials and finished goods, increasing efficiency and productivity in manufacturing, transportation, retail, distribution, shipping, planning and management. IoT sensors are installed throughout the supply chain infrastructure to collect monitoring data, sent to data analytic platforms for advanced analytics. The analysis results are used for the various stakeholders to make quick decisions and react or respond quickly when necessary. Some tasks are automated using IoT actuators controlled by commands from data analytics platforms that analyse sensor data and then carry out control measures or responses. Some of the IoT use cases in supply chains include:

- Location tracking -the tracking of the location of raw materials and finished goods throughout the supply chain.
- Monitoring the products' physical condition or state during transportation and storage throughout the supply chain.
- Asset monitoring and management (e.g., fleet management) -monitoring the various assets deployed throughout the supply chain to facilitate the smooth functioning of the supply chain.
- Stock management -managing the available warehouse stocks, deliveries and orders.

Integrating IoT into supply chains and other technologies such as AI and blockchains transforms supply chains, increasing efficiency and productivity. The supply chain bottlenecks experienced during and after the COVID-19 period demonstrate the need to increase the efficiency of supply chains even though they are getting more complex. The deployment of IoT and other modern technologies to automate some of the processes to increase efficiency and productivity is very important.

IoT-supported retail stores, used to automate some of the processes in supermarkets and small and medium-sized shops. It is driven by the shortage of workers to work in retail stores, the need to reduce costs, and to reduce waiting lines in retail stores. Using IoT technologies to automate some processes increases efficiency and productivity, especially in inventory management, supply chain management, and customer service. Some of the IoT use cases in retail stores include:

- Automated checkout points where customers can serve themselves without needing customer service agents.
- Surveillance monitoring of the entire supermarket or store.
- Monitoring of the products in the supermarkets and control of the environmental

conditions to prolong the shelf life of perishable products.

- Smart shelves for tracking the stock on the shelves.
- Robots to automate some of the tasks that can be executed repeatedly without human intervention.
- IoT-based shopping assistant that monitors the stock of the consumers at home and then reminds them of what they need to buy (and could even order them online).

Supermarkets and shops are becoming smarter with the increased deployment of IoT technologies to automate some of their processes to increase efficiency and productivity and decrease cost. With the gradual decrease in the cost of IoT technologies, supermarkets and small and medium-sized stores will adopt IoT technologies to automate some of their processes.

IoT-based Intelligent Transport Systems (IoT-ITS), that integrate modern Information and Communication Systems and modern technologies into transportation systems to increase productivity and efficiency. It involves using IoT sensors to collect real-time data, which enables real-time monitoring and control to increase the productivity and efficiency of transportation systems and to satisfy some design goals (e.g., reduction of emissions and accidents, improvement of user experiences). Some of the benefits of intelligent transportation systems include:

- Reduction of road traffic, which increases user experience, reduces energy consumption and lowers emissions.
- Enable optimal use of critical resources in transportation systems and increase efficiency and productivity.
- Reduces accidents and facilitates timely emergency response, increasing the safety and security of users.
- Reduces emissions, enabling the transition into cleaner and sustainable transportation systems.
- Increases productivity and efficiency in transportation systems, increasing returns on investments.
- Increase user experience, e.g., reduce the time users spend waiting in traffic (efficient traffic management), reduce the waiting time of users of public transport systems (reduces delays).

In IoT-based Intelligent transport systems (IoT-ITS), IoT sensors are used to gather data sent to computing platforms at a control centre when the data is processed and analysed. The analysis results inform various stakeholders for quick decision-making and timely response. The results of the computations can be sent to manipulate actuators to control some systems within the intelligent transportation system. Some of the use cases of Intelligent Transport Systems are:

- Optimal traffic routing based on real-time traffic monitoring.
- Providing relevant information (weather reports, state of the roads, traffic) to road users to ensure their safety and security and to increase their user experience.
- Assist drivers in searching for available parking places, including cheaper and free parking spaces in their vicinity.
- Timely detection and response to traffic incidents (accidents).

- Real-time traffic rerouting when necessary, especially when the condition of the roads is unsuitable or when there is an emergency.
- Automatic control of speed limits.
- Monitoring of structural properties of the public transport infrastructure to inform users to be aware, ensuring their safety.

Internet of Military Things (IoMT), also known as the Military Internet of Things (M-IoT) or Battlespace IoT (B-IoT), is the integration of IoT sensor and actuator devices into military weapons and battlefield infrastructure for information gathering and automation of some processes, increasing the efficiency of intelligence gathering and combat. Some battlefield assets such as ships, aircraft, battle tanks, weapons, munitions, drones, tucks, soldiers, and operating bases are connected to enable seamless interoperability and efficient cooperation between the various units and systems on the battlefield. The massive amount of data gathered by the sensors embedded within the different military systems provides the relevant stakeholders within the military chain of command a comprehensive situational awareness, improving the efficiency of the command and control and combat operations, especially in complex and diverse conflict zones.

Using sensor networks, actuators and robots on the battlefield to increase situational awareness, risk assessment, response time, and precision is not new. Still, the rapid evolution of IoT technologies and artificial intelligence (AI) will radically transform the future battlefields. The combination of IoT, robotics, and AI will automate some military operations, increasing flexibility and precision during combat and reducing the number of casualties in terms of the number of soldiers killed during combat operations. A significant challenge with M-IoT or B-IoT is cyber security. Incorporating IoT sensors and actuator networks within the military systems and infrastructure exposes them to cyber security risks. A cyber security breach could compromise or disrupt command, control, and combat operations.

Green and sustainable IoT is the application of IoT technologies to reduce pollution and the impact of climate change on the environment and livelihoods. It also involves the application of IoT for resource management and conversations. Sensors are deployed to collect data from the environment. The data collected is analysed for rapid and timely decision-making and control, reducing pollution and conserving critical resources required to sustain ecosystems and human progress. Some of the green and sustainable IoT applications include the following:

- Smart agriculture: One use case example is IoT-based smart irrigation systems designed to reduce water usage in agricultural operations.
- Smart energy: Applying IoT technologies to reduce energy consumption (reducing the carbon footprint) and improve electricity infrastructure efficiency.
- Environmental monitoring: Using IoT to monitor and control the level of pollution in the environment and to detect environmental disasters and prepare for them before they occur.
- Resource management: Application of IoT to conserve critical resources like water and wildlife.
- Supply chains: Increasing the efficiency of supply chains to reduce their carbon emission and other forms of pollution.

The IoT applications discussed above are just a few of the IoT applications that are being developed and adopted in various industries. New IoT application use cases are being designed, and a detailed discussion of almost all IoT applications is out of the scope of this book. However, the IoT applications presented above are a broad category of IoT applications.

2. Introduction to the IoT Microcontrollers

At the perception layer of IoT systems usually some kind of computer operates. Depending on the complexity of the device the computational power of it does not need to be very high. Its design is similar to the Embedded System with the difference that the IoT node device should be equipped with some communication module. We can denote the IoT end-device as the constrained computer, built with four modules: a **sensing unit**, a **processing unit**, a **communication unit**, and a **power supply unit**, as presented in the figure 6. An essential element of the IoT node is the software, capable of performing a set of functionalities.

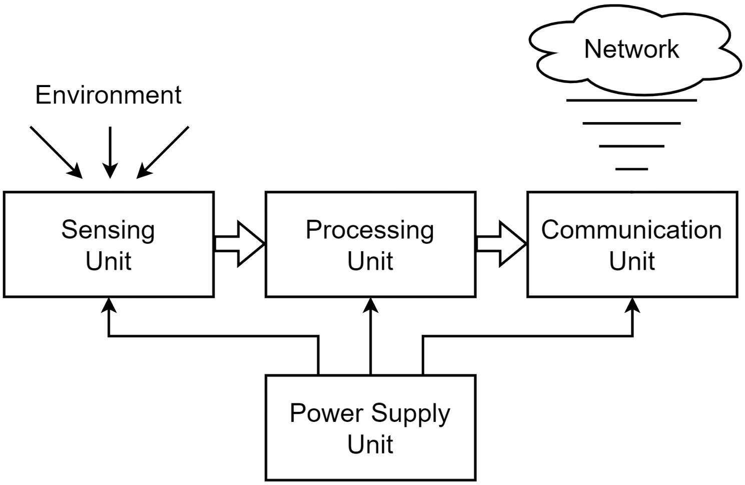


Figure 6: IoT node in the Perception layer

Every computer consists of at least the central processing unit (processor), memory, and peripheral devices, all connected with address, data and control buses. In modern embedded systems architectures usually, the mentioned elements are integrated into one chip forming the single-chip computer. There exists a variety of different kinds of processors with capabilities adjusted to the target system requirements. The most universal is the general-purpose processor which can be a microprocessor, microcontroller, or embedded processor. The **microprocessor** as the device that requires connection of external peripherals and memory is not currently used in embedded or IoT systems design. It gave the leading place to **microcontrollers**, which are representatives of one-chip computers. **Embedded processor** being the extended version of microcontroller is also the popular choice. If such an integrated circuit also contains expansion modules, eg, a radio module, it is called **System on Chip** (SoC). IoT nodes can also be built using the **Digital Signal Processor** (DSP), especially if they implement some audio or video signal processing. For high-volume device production Application Specific Integrated Circuits can be used, designed for specific, specialised use. One type of ASIC can be an **Application Specific Processor** (ASP). There exist also technologies of programmable hardware design implemented as the matrices of universal logic units called **Field Programmable Gate Array**. FPGA technology allows for flexible reconfiguration of the hardware enabling implementation of different microprocessors with other elements of the computer in one universal integrated circuit. This can be achieved with the use of ready descriptions of electronic units written in special hardware description language (Verilog, VHDL). Currently, the most popular choice to design the IoT node is some kind of single-chip computer: microcontroller, embedded processor or System on Chip. The element of this type is an integrated circuit that incorporates all units required to function as the computer. It includes a central processing unit (CPU), memory for programs, memory for data, inputs, outputs, timers, serial communication ports and other peripherals. Complex microcontrollers, called embedded processors, can include more processor cores, display controllers, advanced internal data transfer mechanisms (like DMA), programmable connections between modules, specialised coprocessors for ciphering and deciphering, compression and decompression, video and audio coding and decoding, and other modules. Wireless networking capability makes Microcontrollers even more complex in the IoT world. A complex microcontroller equipped with an internal radio communication module is also known as a System on Chip (SoC).

Although many microcontrollers or SoCs are called processors, historically, the processor is the name of the element of the CPU functionality only. It must be connected to memory and peripherals to form a fully functional computer. On the other hand, a microcontroller, embedded processor or System on Chip can work without any external elements; it just requires the power supply to operate.

The typical microcontroller includes general-purpose units like:

- CPU core,
- Program memory,
- Data memory,
- Timers, Counters,
- Interrupt controller,
- I/O ports,
- Serial synchronous and asynchronous communication ports,
- Analog to Digital converter,
- PWM (Pulse width Modulation unit for Digital to Analog conversion),
- DMA controller,
- Supervisory units (Watchdog, Reset, Brownout).

Embedded Processor or System on Chip can contain also:

- Network interface,

- USB controller,
- Memory interface module,
- Floating point unit (FPU),
- Cryptographic module,
- Other application-specific extensions.

The **CPU core** is the unit that executes the main program. It controls program flow, executes general-purpose instructions, calculates addresses, and processes integer values. For fast floating point calculations, an FPU coprocessor is built-in. It executes instructions that perform calculations on real numbers and advanced mathematical functions. The program instructions are fetched from **program memory**, usually implemented as internal or external flash memory. Data is stored in internal **data memory** implemented as static RAM. If more memory is needed, some microcontrollers have a memory management unit that allows them to connect external DRAM memory. Flash memory is often used as a place for file storage. **Timers and counters** are units that help to generate pulses of specified length and square signals of selected frequency. They can also measure delays and synchronise the work of other modules like serial ports, converters, and displays. Timers can generate pulse width modulated signals to control the speed of motors and light brightness. Microcontrollers have **digital input and output ports** to connect other elements of the systems. Connecting external sensors to collect information from the surroundings and output devices to manipulate environmental parameters is possible. **Analogue inputs** can read the voltage value generated by simple sensors. **Serial communication ports** are used to connect more complex sensors and displays to communicate with the user or another computer system. An **interrupt controller** is a unit that automatically executes subroutines responsible for handling tasks specific to the hardware that signalled the situation that needs the processor's attention. The processor doesn't have to waste execution time by periodically checking if there is a need to take care of the device. It helps to make the code more efficient and reliable. **Supervisory units** help to recover from some abnormal situations. Watchdog resets the processor in case the software hangs up. Brownout detector constantly monitors the power supply voltage. It stops the processor if the voltage is too low for proper operation to avoid execution errors, flash write errors, and other malfunctions. Supervisory interfaces like JTAG allow writing the programs into flash memory and debugging the code. **Direct Memory Access** (DMA) module performs memory operations without processor intervention. It is usually used for copying data blocks between memory and other peripheral units. For example, data from the network unit is stored automatically in the buffer, and the CPU is informed while the data transfer is complete.

Details of the internal construction and operation of many internal modules of popular microcontrollers are described in further chapters of this book.

3. Introduction to Embedded Programming

IoT systems share programming paradigms with embedded systems. Each microcontroller manufacturer has its own set of tools (called SDK or Development Framework) that frequently contain an IDE dedicated to the platform. There are some cross-platform solutions and frameworks, however.
Programming languages include:

- C/C++ - undoubtedly the most popular, versatile, yet demanding programming language. With modern supporting tools such as syntax highlights, code samples, code generators (AI-based) and instant syntax checking, C/C++ programming became relatively easy but still requires solid software development foundations. On the other hand, it is probably the only programming language that is natively supported with hardware debugging features. C/C++ bare metal programming allows the developer to control all MCU features on the lowest level and implement energy-efficient, fast and compact solutions.
- Java and Javascript - with low entry-level for developers, usually represented by the variation of NodeJS, limited and applicable to beginners. Within the constraints of the interpreter, it provides rapid prototyping and the fastest market delivery but the lowest flexibility and extensibility beyond what the manufacturer plans. Also, the Java development framework implemented in the microcontroller is compact because of the constrained resources. Usually, it does not keep standards, so the feature of the portability of the code is somewhat limited.
- Python (Micropython) - similarly to Java, offers an easy start but low flexibility and control over the hardware. Acceptable for prototyping.
- Other.

3.1. IoT and Embedded Systems Programming Models

IoT device programming can be done on a variety of levels. Below are the most popular models and a brief discussion of their pros and cons.

Bare Metal Programming

The bare metal programming model is where the software developer builds firmware (usually from scratch or based on a stub generated by the SDK) and flashes it to the MCU. The MCU usually does not contain software other than technical ones necessary for starting and updating the device, e.g. a bootloader. The developer must implement all algorithms, communication, interfacing, storage, etc., on a low level. They may use 3rd party libraries to implement it, which speeds up development significantly. There is no operating system running in the background. Eventually, it comes with the firmware as part of it, as included by the developer, e.g. FreeRTOS [25].

Bare metal programming applies first to the Edge class devices, rarely to the Fog class.

Bare metal programming requires a good understanding of the hardware configuration of the IoT device as well as the configuration of the software development toolchain. The MCU manufacturer usually provides SDK and related tools, but there do exist middleware solutions (such as PlatformIO [26]) that significantly simplify installation.

In most cases, source code is written in C or C++ language or their combination (e.g. in the case of the STM). The development process for bare metal programming is present in the following figure 7 and its features are discussed in table 2. In short, it requires developing, compiling and uploading the firmware to the device's flash memory. Programming uses a programmer (physical or Over-the-air - OTA, virtual interface). The bare metal model usually provides hardware development capabilities.

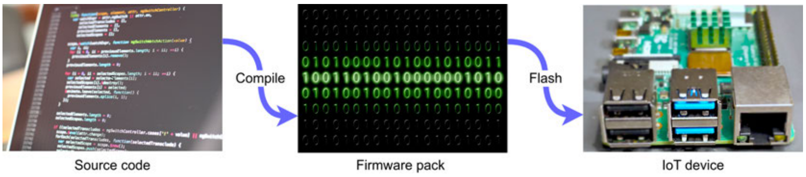


Figure 7: Bare metal IoT firmware development process

The bare metal programming model is considered the only one that enables developers to have absolute control over the hardware on a very low level. On the one hand, it brings opportunities to implement non-standard solutions and optimal code in terms of compactness and efficiency; on the other, it increases time-to-market delivery. Recent advances in development supporting tools (e.g. AI-based code generation), wide availability of the libraries, standardisation of their presence and automated management, such as, e.g. in PlatformIO Library Management [27] significantly lower this time.

Table 2: Bare metal programming pros and cons

Pros	Cons
Absolute control over hardware	Need to implement all from scratch
Secure, low vulnerability	Requires good hardware understanding
No bottlenecks	Requires advanced programming skills
Efficient and compact code	Requires complex development environment
Fastest, no overhead of the middleware	Possibility to brick the device during flashing
Good community support	Time consuming implementation
Highly flexible, enables the developer to prepare non-standard solutions	
Provides hardware debugging capabilities	
Energy efficient, it gives control over low-level, energy-saving mechanisms (waitstates, sleep modes, radio power, etc.)	

In the IoT world, it is common to distribute firmware remotely (OTA - Over The Air). For this reason, it is pretty frequent that the flash memory of the IoT device is split in half into two partitions, each containing a different version of the firmware (old and new). OTA mechanism flashes an inactive partition, and then the bootloader swaps them during the reboot of the device once flashing is done. If new firmware fails to boot, the bootloader swaps the partition back to run the old version, reboots the device, and notifies about the update error.

Script Programming with Middleware

Opposite to bare metal programming, script programming does not involve compilation or firmware burning into the flash memory. This programming model uses interpreted languages such as Python (actually Micropython: an edition of Python for microcontrollers dedicated to constrained devices), NodeJS, Javascript, Java, C#, etc. A virtual machine middleware (programming language interpreter) running bare metal (installed as firmware) or as a part of the operating system (if any), and the developer prepares an algorithm as a script, usually in a textual form, later uploaded and executed on the device. The middleware brings an overhead on execution; thus, this solution is intended for not-so-constrained IoT devices, still acceptable for Edge and quite common for Fog class. It requires much more CPU, RAM and storage than bare metal programming, has limitations from the interpreter implementation and only indirectly accesses hardware. It is not suitable for real-time solutions.

Scripting programming is common for more powerful Edge devices and almost the first choice for Fog class devices.

The development process for scripting programming is present in the following figure 8 and its features are discussed in table 3. In short, it requires limited SDK (or none), but debugging is complex, if possible.

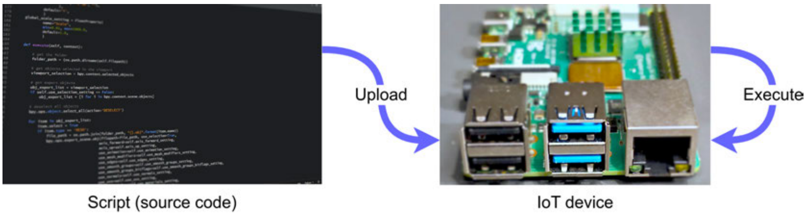


Figure 8: Scripting IoT programming process

This programming model is suitable wherever standard solutions are implemented and where code execution efficiency is not critical, and there is no demand for real-time operations; eventually, the IoT device is unconstrained, providing developers with decent CPU (e.g. modern ARM), RAM and storage. Note those solutions are usually less energy efficient than bare metal programming; still, they offer great flexibility in algorithm implementation, far beyond a predefined list of choices or limited configuration as presented in the following section. On the other hand, it speeds up delivery time to the market because of the ease of implementation, the lack of need to install the complex software development environment and the high level of abstraction.

Table 3: Scripting programming pros and cons

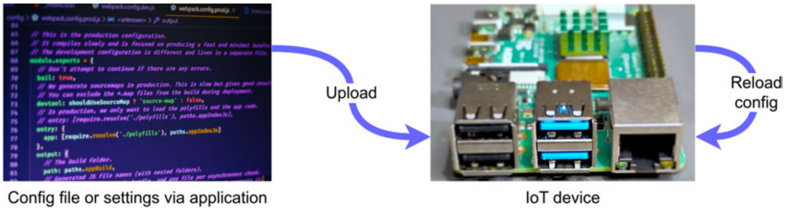
Pros	Cons
Better suitable for beginners	Not optimal because of the middleware overhead on execution
Uses higher abstraction level	Not suitable for real-time operations
Uses high-level programming languages	Limited hardware interfacing and features to those exposed by the middleware
Usually does not involve complex SDK/development environment	Limited and non-optimal energy efficiency management
Flexible enough to implement complex algorithms	Low extendibility
Rapid development	Middleware updates used to cause the need to adapt script with algorithm
	Hardware debugging is tricky or not possible at all

Configuring Firmware

Several configurable firmware (IoT frameworks) are available for various IoT devices. This development model focuses on reconfiguring the ready-to-use firmware delivered “as is” using some configuration interface or script (or both). Eventually, modifying and recompiling it yourself is possible if it is open source. Still, the recompilation process is usually very complex, and understanding all relations and development toolchains is sometimes more complicated than developing a solution from scratch as a bare metal. Some open-source firmware (like Tasmota, ESPhome, and OpenBeken) offer high flexibility and configureability, making their use the simplest and fastest way to develop IoT devices. In contrast, proprietary firmware does not bring this opportunity at all and is delivered "as is" with a predefined set of features. Software modifications are not allowed, and configuration is limited to changing the state of the elements from simply switching them on and off to setting up access and credentials. This usually does not bring capabilities to modify the algorithm, eventually to choose a behaviour from the predefined list proposed by the firmware author. Such a model does not bring debugging capabilities; finally, simple tracking with error codes and log files (if at all). Moreover, in many scenarios, firmware operation is dependent on some external resources (e.g. authorisation via a cloud or firmware updates delivered with this channel).

Firmware configuration model is applied to both Edge and Fog class devices, exposed via IoT frameworks. Sometimes, it also involves the cloud part of the solution.

The development process for the firmware configuration model is present in the following figure 9 and its features are discussed in table 9.


webpack.config.dev.js
webpack.config.prod.js
config > webpack.config.prod.js > <unknown> > output
Upload
Reload config
Config file or settings via application
IoT device

```
// This is the production configuration.
// It compiles slowly and is focused on producing a fast and minimal bundle
// The development configuration is different and lives in a separate file.
module.exports = {
  // Don't attempt to continue if there are any errors.
  bail: true,
  // We generate sourcemaps in production. This is slow but gives good resul
  // You can exclude the *.map files from the build during deployment.
  devtool: shouldUseSourceMap ? 'source-map' : false,
  // In production, we only want to load the polyfills and the app code.
  // entry: [require.resolve('./polyfills'), paths.appIndexJs],
  entry: {
    app: [require.resolve('./polyfills'), paths.appIndexJs]
  },
  output: {
    // The build folder.
    path: paths.appBuild,
    // Generated JS file names (with nested folders).
```

Figure 9: Firmware configuration process

Configuration r a nge v a ries a m ong I o T f r ameworks b u t c o mmonly r e quires compatible hardware. Proprietary firmware p r ovides s e aled c o nfiguration so ft ware an d encryption; thus, it virtually excludes any non-standard modifications o r m a kes t h em v e ry complex. Configuration i n p r oprietary fi rm ware sc en arios ca n be pr ov ided in di rectly vi a a cloud solution that raises serious questions about privacy (e.g. configuring y o ur p r ivate WiFi router credentials via a public or 3rd party cloud, not directly to the device). It is worth mentioning that IoT hardware used to be compatible with more than one firmware, and proprietary ones can be replaced with alternative open-source firmware, e . g. Tasmota [28], ESPHome [29], OpenBeken [30], ESPEasy [31], ESPurna [32]. Unfortunately, the replacement process usually requires specialised skills like soldering because, in most cases, first-time reflashing needs a physical connection with the chip.

WARNING! The reflashing process usually needs specialised skills and requires high care. Flashing an alternative firmware can lead to unexpected behaviour of the device and can make the device unusable. Never connect anything or touch the device while it is opened and powered by an electric line!

Table 4: Middleware configuration model pros and cons

Pros	**Cons**
Easy to use even for beginners	Limited number of use scenarios
Consistent environment (configuration, use) common look and feel	Problems appearing hard to solve in case of failure
No need for SDK, configuration tools use plain text, browsers or apps	Low flexibility - limited support for hardware (only proprietary or limited compatibility in the case of the open source)
Manufacturer's support (for proprietary) but usually for a limited time and shorter compared to open source solutions	Doubtful privacy, in particular when a public cloud is in use
Usually reliable	Lack of help once the Manufacturer's maintenance period is finished
During the maintenance period, updates are given by the vendor	

In this book, we focus on the bare metal programming model using the C/C++ model, but we also present some aspects of scripting programming and review some IoT frameworks that are exposed with the alternative firmware configuration model.

3.2. Introduction to the Programming Frameworks

In the beginning, it is essential to distinguish an IoT Framework that is a set of tools, firmware for a variety of devices, sometimes also hardware, delivered as is and providing developers with configuration capabilities on the high abstraction level from the Programming Framework that is related to the low-level programming, here in C/C++, referred to as an SDK. SDK tends to be a narrower definition than a programming framework as the former contains both SDK and tools, development toolchain and code organisation rules.
This chapter presents and discusses programming frameworks (SDKs and source code organisation) that define how the IoT code is organised on the low level in the Bare Metal programming model for Edge class devices.

Almost every MCU (microchip/microcontroller) vendor develops its own SDK, providing programmers with a specific programming framework. It is worth nothing to mention that, in many cases, it follows the general programming construction of the source code for C or C++, such as below:

```
int main() {
    std::cout << "Hello IoT!";
    return 0;
}
```

A common approach is to use a GUI to automate the generation of the source code stub that contains the hardware-specific configuration, e.g. timers, GPIOs, and interrupts, to avoid monotonous and complex tasks and speed up time to market.
Still, as hardware differs, it is particular for each platform, and usually, software development requires a rigorous approach to inject user-specific code only in predefined locations. Otherwise, it may break source code or even delete it when re-generating configuration using SDK tools and automation. Sample `main()` function for the STM32 MCU is presented below. Developers are intended to fill their code only in predefined areas, such as starting from `USER CODE BEGIN Init` and finishing before `USER CODE END Init`; otherwise, the source code will be gone when updating the configuration:

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */
  HAL_Init();
  /* USER CODE BEGIN Init */

  /* USER CODE END Init */
  SystemClock_Config();
  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */
  MX_GPIO_Init();
  MX_LPUART1_UART_Init();
  MX_NVIC_Init();
  /* USER CODE BEGIN 2 */
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      nUARTBufferLen = sprintf((char*)tUARTBuffer, "Hello World!\n\r");
      HAL_UART_Transmit_IT(&hlpuart1, tUARTBuffer, nUARTBufferLen);
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

Studying specific platforms is time-consuming, and as each vendor has its approach, knowledge and source codes are usually not portable between microcontrollers.
Specific frameworks for hardware vendors are (among others):

- Espressif ESP8266:
  - ESP8266 RTOS SDK,
  - ESP8266 Non-OS SDK,
  - Arduino.
- AVR/Atmel:
  - AVR Studio (Atmel Studio),
  - Arduino.
- Espressif ESP32:
  - ESP-IDF,
  - Arduino.
- Nordic Semiconductors nRF52:
  - Mbed,
  - Zephyr RTOS,
  - nRF5 SDK,
  - Arduino.
- ST Microelectronics STM32 series:
  - Mbed,
  - CMSIS,
  - Zephyr RTOS,
  - Registers programming model (RAW),
  - STM32Cube (HAL),
  - Arduino.

Typical C++ code, as presented above, is a single-pass execution. On the other hand, IoT devices used to work infinitely, handling their duties such as reading sensors, communicating over the network, sending and receiving data, routing messages and so on, thus requiring setting up an infinite `while (1)` loop for processing. Many tasks need to be done in parallel, so it is expected to include a task scheduling mechanism to run multiple tasks asynchronously. A common is to use the FreeRTOS [33] or its modified versions for the specific hardware platform provided by the hardware vendor, e.g. as in the case of the ESP32 [34] to provide support for multicore MCUs.

Name FreeRTOS may be misleading because it can be understood as a general purpose operating system (GPOS), suggesting it runs in the background before your application starts as Windows or Linux does. FreeRTOS as an Embedded Operating System (OS for embedded systems and microcontrollers) is included as a C/C++ library in the source code and built into the firmware and algorithms. It provides similar functionalities as the GPOS kernel with task handling, memory management, file system, etc.

3.2.1. Arduino Framework

Observing the list of software frameworks above, one can easily find that many platforms have common frameworks, but the Arduino framework is present for all of them. Arduino framework is a cross-platform approach providing a slightly higher level of abstraction over dedicated software frameworks, and it is the most popular among hobbyists, students, professionals and even researchers at the moment. Arduino Framework is a reasonable balance between uniform code organisation and elements of cross-hardware HAL, still bringing opportunities to access hardware on a low level and get the advantage of the advanced features of modern IoT microcontrollers such as, e.g. power management. Most hardware vendors support this framework natively, and it has become almost an industry standard. Some advanced hardware controls may require integration or other native frameworks, anyway. Still, the Arduino framework has real-time capacity. It is powerful and flexible enough to handle most IoT-related tasks, and most of all, it has excellent community support with dozens of software libraries, examples and applications worldwide.

A dummy C/C++ code for the Arduino framework looks as follows:

```
void setup()
{

}

void loop()
{

}
```

The `void setup()` function is executed only once after the microcontroller reboots. Its

purpose is to initialise, instantiate objects, read configuration, check working conditions, and so on: generally, all tasks that are to be executed only once in a work cycle of the IoT device.
The `void loop()` function is executed in a loop automatically and infinitely once a single pass is finished. Its purpose is to implement repeating tasks such as periodic reading of a sensor and sending the data to the cloud. There is no need to implement a dummy `while(1)` inside the `loop()`; moreover, it is usually not advised or even forbidden. It is because, for every execution of the `loop()` statement, many other tasks, such as handling communication, may be executed once. Making a single pass of the `loop()` function infinite (e.g. implementing an infinite `while(1)` loop could cause starvation of the other underlying processes the framework handles, such as network communication, embedded protocols handling, etc.).

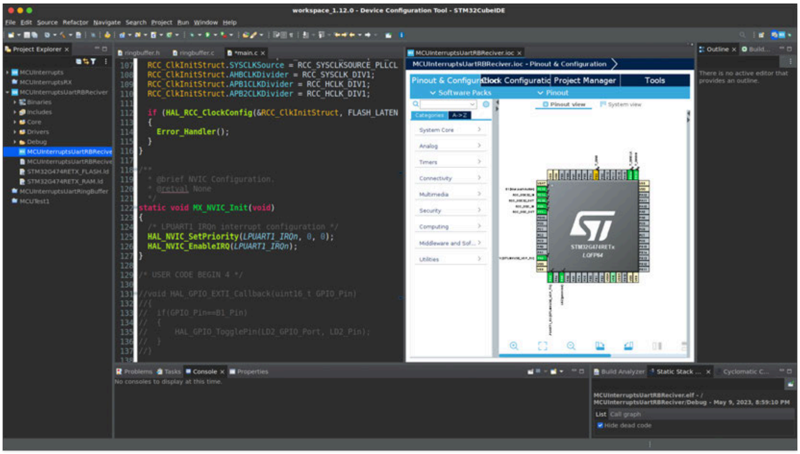
The book presents code and examples in the Arduino framework context for edge-class devices and Fog-class devices (scripting). Wherever other framework is used, it will be clearly stated. Note, following introduction to the C and C++ programming and task handling contents are universal and can be applied to the other frameworks, whether directly or indirectly, with some adaptation on the code level.

3.3. Software Development Tools and Platforms

Software development in the bare metal model requires a development toolchain installed on the developer's computer. The vendor of the MCU usually provides a set of tools. This set frequently includes a dedicated compiler, linker, library management tools, configuration tools, debugger software, etc. These tools are command lines in most cases. Using a GCC [35] C/C++ compiler is also quite common. On top of it, a GUI with a rich UI interface is built to simplify software development. Some vendors provide their own GUIs, such as STMicroelectronics' STM32CubeIDE (image 10). In contrast, others use already available universal code editing solutions and integrate with them, e.g. in the form of plugins or extensions.

Vendors barely develop their own GUI solutions from scratch; instead, they adapt existing open-source ones, e.g. STM32CubeIDE is built on top of the Eclipse IDE [36].

Detailed instructions on the tools necessary to set the development environment, e.g., the STM32 WB chip family, are presented on the STM's YouTube channel [37].

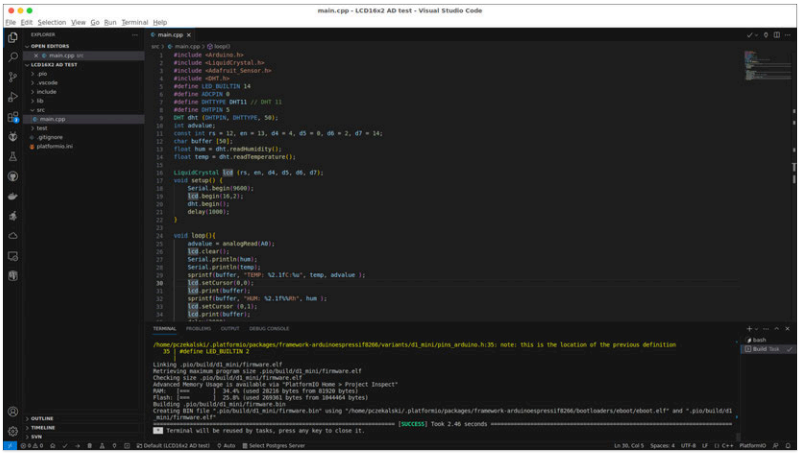

```
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCL
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATEN
  {
    Error_Handler();
  }
}

/**
  * @brief NVIC Configuration.
  * @retval None
  */
static void MX_NVIC_Init(void)
{
  /* LPUART1_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(LPUART1_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(LPUART1_IRQn);
}

/* USER CODE BEGIN 4 */

//void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
//{
//  if(GPIO_Pin==B1_Pin)
//  {
//      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
//  }
//}
```

No consoles to display at this time.

There is no active editor that provides an outline.

Figure 10: STM32CubeIDE: Eclipse for STM32 developers

Because documentation for the command line tools composing SDK is usually available, there are also universal solutions that enable developers to use a single GUI environment for various tasks and microcontrollers, switching among them quickly, such as Visual Studio Code (figure 11). Each platform requires its dedicated toolchain, anyway, and integration with universal code editors such as the aforementioned VS Code may be tricky. Luckily, there are tools to help with the automated installation of all required components, such as PlatformIO [38]), which we describe below.

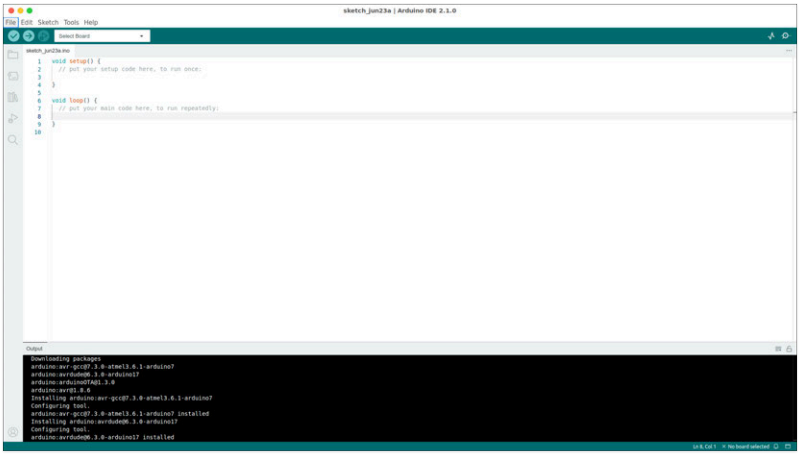

main.cpp - LCD16x2 AD test - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
LCD16X2 AD TEST
.pio
.vscode
include
lib
src
main.cpp
test
.gitignore
platformio.ini
OUTLINE
TIMELINE
SVN
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
Default (LCD16x2 AD test)
Auto
Select Postgres Server
Ln 30, Col 5 Spaces: 4 UTF-8 LF C++ PlatformIO

```
#include <Arduino.h>
#include <LiquidCrystal.h>
#include <Adafruit_Sensor.h>
#include <DHT.h>
#define LED_BUILTIN 14
#define ADCPIN 0
#define DHTTYPE DHT11 // DHT 11
#define DHTPIN 5
DHT dht (DHTPIN, DHTTYPE, 50);
int advalue;
const int rs = 12, en = 13, d4 = 4, d5 = 0, d6 = 2, d7 = 14;
char buffer [50];
float hum = dht.readHumidity();
float temp = dht.readTemperature();

LiquidCrystal lcd (rs, en, d4, d5, d6, d7);
void setup() {
    Serial.begin(9600);
    lcd.begin(16,2);
    dht.begin();
    delay(1000);
}

void loop(){
    advalue = analogRead(A0);
    lcd.clear();
    Serial.println(hum);
    Serial.println(temp);
    sprintf(buffer, "TEMP: %2.1fC:%u", temp, advalue );
    lcd.setCursor(0,0);
    lcd.print(buffer);
    sprintf(buffer, "HUM: %2.1f%%Rh", hum );
    lcd.setCursor (0,1);
    lcd.print(buffer);
```

```
/home/pczekalski/.platformio/packages/framework-arduinoespressif8266/variants/d1_mini/pins_arduino.h:35: note: this is the location of the previous definition
   35 | #define LED_BUILTIN 2
      |
Linking .pio/build/d1_mini/firmware.elf
Retrieving maximum program size .pio/build/d1_mini/firmware.elf
Checking size .pio/build/d1_mini/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [===       ]  34.4% (used 28216 bytes from 81920 bytes)
Flash: [===       ]  25.8% (used 269361 bytes from 1044464 bytes)
Building .pio/build/d1_mini/firmware.bin
Creating BIN file ".pio/build/d1_mini/firmware.bin" using "/home/pczekalski/.platformio/packages/framework-arduinoespressif8266/bootloaders/eboot/eboot.elf" and ".pio/build/d1
_mini/firmware.elf"
======================================= [SUCCESS] Took 2.46 seconds =======================================
 * Terminal will be reused by tasks, press any key to close it.
```

Figure 11: VS Code: a universal development environment

As the Arduino programming framework became a cross-platform standard, vendors provided low-level libraries implementing standard functionalities such as embedded communication protocols (Serial, SPI, I2C, 1Wire) and networking communication. Arduino, a manufacturer of popular development boards, provides an IDE (figure 12: Arduino IDE [39]) that is intended to be an entry-level development environment. It can be extended beyond genuine Arduino boards, e.g. with the Espressif toolchain for ESP8266 and ESP32. This software, however, is very limited in features and is suitable only for simple projects.


sketch_jun23a | Arduino IDE 2.1.0
File Edit Sketch Tools Help
Select Board
sketch_jun23a.ino
Output
Ln 8, Col 1 × No board selected

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

```
Downloading packages
arduino:avr-gcc@7.3.0-atmel3.6.1-arduino7
arduino:avrdude@6.3.0-arduino17
arduino:arduinoOTA@1.3.0
arduino:avr@1.8.6
Installing arduino:avr-gcc@7.3.0-atmel3.6.1-arduino7
Configuring tool.
arduino:avr-gcc@7.3.0-atmel3.6.1-arduino7 installed
Installing arduino:avrdude@6.3.0-arduino17
Configuring tool.
arduino:avrdude@6.3.0-arduino17 installed
```

Figure 12: Arduino IDE: an entry-level IDE for beginners

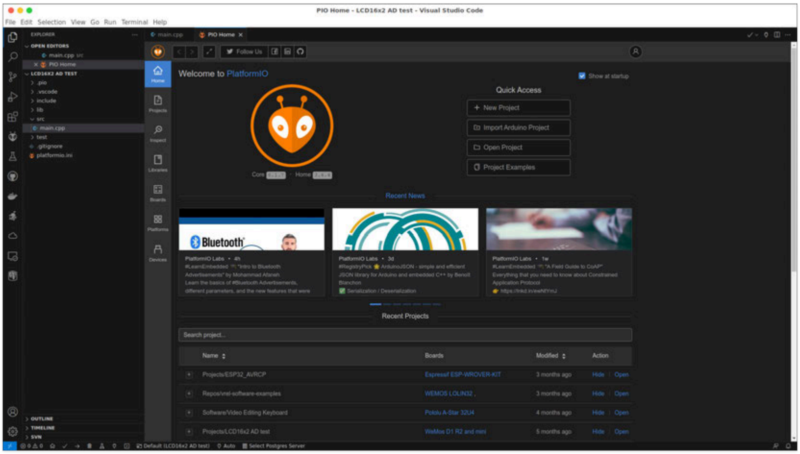
We suggest starting with VS Code and PlatformIO over Arduino IDE, even if you're a beginner.

3.3.1. Developers Middleware and Support Tools

Several additional tools usually come with the development toolchain provided by the hardware vendors. They include programmers (flashers, injecting firmware into the IoT device), configuration tools, power consumption calculators, etc. Installation is not always straightforward, and updating is tricky. Developers who use a variety of platforms (MCUs) struggle with instant updates and browsing the web for tools, samples and libraries. Moreover, handling libraries they use for development is time-consuming and involves instant monitoring of changes, manual copy-paste operations on files, etc. Moreover, it has to be done individually for every project.

PlatformIO

The solution is a developer's middleware that integrates with selected IDE and helps to install, configure and maintain toolchains for hardware, software development libraries, and also contains a set of additional tools (e.g. serial port monitor, JTAG debugger, code repository integration, collaboration tools, remote development, etc.). As mentioned above, one example of a handy middleware for IoT and embedded development is PlatformIO. It is a command-line toolset that provides a whole ecosystem for virtually any hardware platform; it still uses the vendor's proprietary toolchains. It perfectly integrates with Visual Studio Code (among others) via VS Code's extension (plugin) systems. VS Code also works as a GUI for PlatformIO. In the following figures, we present its look and UI when integrated with Visual Studio Code (figures 13, 14 and 15).



Name	Boards	Modified	Action
Projects/ESP32_AVRCP	Espressif ESP-WROVER-KIT	3 months ago	Hide Open
Repos/vrel-software-examples	WEMOS LOLIN32 ,	3 months ago	Hide Open
Software/Video Editing Keyboard	Pololu A-Star 32U4	4 months ago	Hide Open
Projects/LCD16x2 AD test	WeMos D1 R2 and mini	5 months ago	Hide Open

Figure 13: VSCode with PlatformIO: starting page

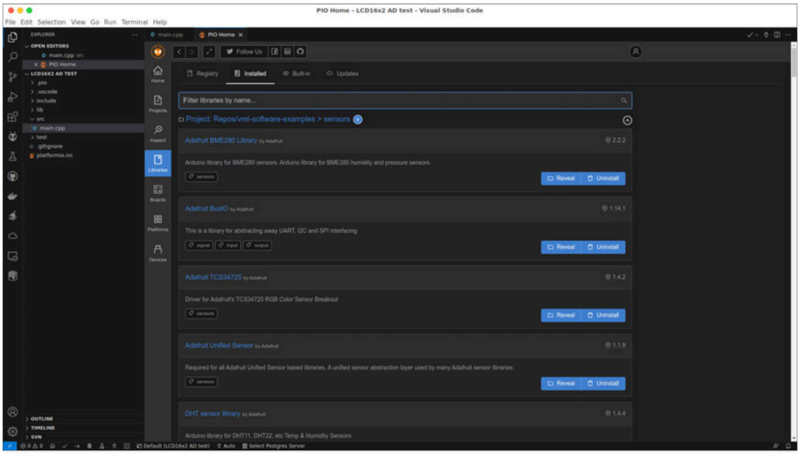
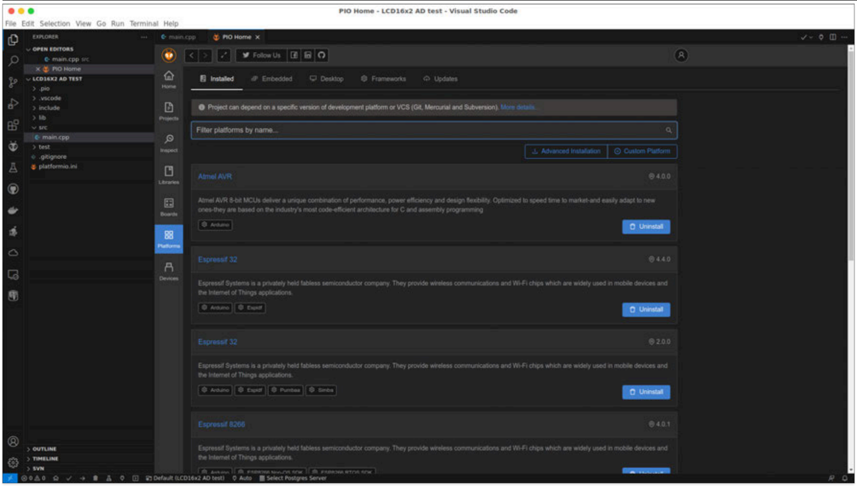

PIO Home - LCD16x2 AD test - Visual Studio Code
Registry
Installed
Built-in
Updates
Filter libraries by name...
Project: Repos/vrel-software-examples > sensors
Adafruit BME280 Library by Adafruit
Arduino library for BME280 sensors. Arduino library for BME280 humidity and pressure sensors.
Adafruit BusIO by Adafruit
This is a library for abstracting away UART, I2C and SPI interfacing
Adafruit TCS34725 by Adafruit
Driver for Adafruit's TCS34725 RGB Color Sensor Breakout
Adafruit Unified Sensor by Adafruit
Required for all Adafruit Unified Sensor based libraries. A unified sensor abstraction layer used by many Adafruit sensor libraries.
DHT sensor library by Adafruit
Arduino library for DHT11, DHT22, etc Temp & Humidity Sensors
Reveal
Uninstall

Figure 14: VSCode with PlatformIO: library management



Project can depend on a specific version of development platform or VCS (Git, Mercurial and Subversion). More details...

Filter platforms by name...

Atmel AVR 4.0.0

Atmel AVR 8-bit MCUs deliver a unique combination of performance, power efficiency and design flexibility. Optimized to speed time to market-and easily adapt to new ones-they are based on the industry's most code-efficient architecture for C and assembly programming

Arduino

Espressif 32 4.4.0

Espressif Systems is a privately held fabless semiconductor company. They provide wireless communications and Wi-Fi chips which are widely used in mobile devices and the Internet of Things applications.

Arduino Espidf

Espressif 32 2.0.0

Espressif Systems is a privately held fabless semiconductor company. They provide wireless communications and Wi-Fi chips which are widely used in mobile devices and the Internet of Things applications.

Arduino Espidf Pumbaa Simba

Espressif 8266 4.0.1

Espressif Systems is a privately held fabless semiconductor company. They provide wireless communications and Wi-Fi chips which are widely used in mobile devices and the Internet of Things applications.

Arduino ESP8266 Non-OS SDK ESP8266 RTOS SDK

Figure 15: VSCode with PlatformIO: toolchain management

A PlatformIO-enabled IoT project is a set of files with a `platformio.ini` file in the root folder and `main.cpp` in the `./src/` subfolder (as, e.g. in the figure 13, project folder tree is to the left)). The `platformio.ini` file describes all technical parts of the project: the hardware platform, the method of uploading the firmware (usually via a serial port), software libraries that are included in the code and should be automatically pulled from the libraries repository during compilation and many other options [40]. Sample `platformio.ini` file is presented in the code below:

```
[env:d1_mini]
platform = espressif8266
board = d1_mini
framework = arduino
upload_port = /dev/ttyUSB0
upload_speed = 9600
monitor_port = /dev/ttyUSB0
lib_deps =
        arduino-libraries/LiquidCrystal@^1.0.7
        adafruit/Adafruit Unified Sensor@^1.1.7
        adafruit/DHT sensor library@^1.4.4
```

This code configures the ESP8266 (Espressif) hardware project, with specific developer board D1 Mini and programming done in the “Arduino” framework development model.
Communication with the IoT device is via serial port (here `/dev/ttyUSB0` for Linux or, e.g. `COM3` for Windows) and uses the same port for monitoring (serial port monitor for tracing messages from the MCU and code).
It uses three libraries registered in the library registry for PlatformIO: `LiquidCrystal`, `Adafruit Unified Sensor` and `DHT sensor library`, with explicit versions. PlatformIO's Library Manager automatically checks for updates and proposes to update libraries to the latest available if a version is not explicitly stated. The Library Registry in the PlatformIO is a repository of the Gitlab project, available online [41]. Libraries are currently held per project instead of shared between projects.
At the start of the PlatformIO GUI and then periodically, it checks for PlatformIO updates and development toolchain updates, proposing to update them when a new version is available.

3.4. C/C++ Language Embedded Programming Fundamentals

The following sub-chapters cover programming fundamentals in C/C++, which comply with most C/C++ notations. Those who feel comfortable in programming will find these chapters somewhat introductory, while for those having no or little experience, it is highly recommended to cover this introduction. This chapter and its sub-chapters target the basics and general syntax of C/C++ programming for different platforms, including Arduino, Espressif, Nordic, STM32, and partially for Raspberry Pi devices; however, in any case, the programming environment configuration is different for every platform. The Arduino programming framework is common for many MCU manufacturers of the IoT Edge class devices in bare metal programming mode, even if it brings some overhead and does not let the developer push the devices to their limits. To enjoy full power, efficiency and control of the specific device, one needs to use a dedicated SDK and Framework, but for teaching purposes and many even professional applications, Arduino Framework is suitable and a good balance between the cost of the development and the result.

This manual refers to the particular versions of the software available at the moment of writing this book. Accessing specific features may change over time along with the evolution of the platform. Please refer to the attached documentation (if any) and browse Internet resources to find the latest guidance on configuring specific development platforms when in doubt.

3.4.1. Data Types and Variables

Almost every computer program manipulates the data. Data representation in the program is variable. In C/C++, the variable needs to be defined before using it, giving it some name and assigning a chosen type, dependent on the kind of data. Some common data types and how to use variables are shown below.

Data Types

Data type specifies how it is encoded and represented in the computer memory. For example, integer numbers are binary-encoded, the texts are represented as a series of ASCII-encoded characters, and real numbers have a particular encoding scheme that consists of two binary numbers - mantissa and exponent. Other data types are tables that consist of elements of the same type or structures with elements of different types. There are plenty of different data types; some of them are predefined, but user-own types can be defined based on existing ones. Creating a variable requires specifying its type, which determines its place in the memory of the microcontroller and also the way it can be used. Further, the most used ones together with examples of defining variables.

- **byte** – a numeric type of 8 bits that stores numbers from 0 to 255.

```
byte exampleVariable;
```

The example above defines a variable, reserves its memory, and assigns the memory address to the variable name. It is also possible to give the variable the initial value as below:

```
byte exampleVariable = 123;
```

- **int** – the integer number. Its size depends on the microcontroller class. In the case of AVR (Arduino), it consists of 16 bits that can contain values from –32 767 to 32 768. In ARM-based microcontrollers (like STM32), its size is 32 bits.

```
int exampleVariable = 12300;
```

- **float** – a data type for real numbers that uses 32 bits and stores numbers approximately from –3.4 × 10^38 to 3.4 × 10^38.

```
float exampleVariable = 12300.546;
```

- **array** – a set of data of the same type that can be accessed using a serial number or index. The index of the first element is always 0. The values of an array can be initialized at the definition of it or set during the program's execution. In the following example, the array of four elements with the name “first array” and data type `int` has been created. The value of the array with an index of 0 will be 12, and the value with an index of 3 will be 15.

```
int firstArray[] = {12,-3,8,15};
```

Square brackets of the array can be used to access some value in the array by index. In the following example, the element with index 1 (that is –3) is assigned to the `secondVariable` variable.

```
int secondVariable = firstArray[1];
```

An array can be quickly processed in the loop. The following example shows how to calculate the sum of all elements from the previously defined array (`for` statement will be explained in detail in the following chapters).

```
//The loop that repeats 4 times
int sum = 0;
for(int i = 0; i < 4; i = i + 1){
     sum = sum + firstArray[i];
}
```

The loop in the example starts with index 0 (`i` = 0) and increases it by 1 while smaller than 4 (not including). That means the index value will be 3 in the last cycle because when the `i` equals 4, the inequality `i` < 4 is not true, and the loop stops working.

- **bool** – the variables of this data type can take values TRUE or FALSE. Arduino environment allows the following values to these variables: TRUE, FALSE, HIGH (logical 1 (+5 V)) and LOW (logical 0 (0 V)).

Data Type Conversion

Data type conversion can be done using multiple techniques – casting or data type conversion using specific functions.

- **Casting** – the cast operator translates one data type into another type straight forward. The desired variable type should be written in the brackets before the variable data type, which needs to be changed. In the following example, where the variable type is changed from float to int, the value is not rounded but truncated. Casting can be done to any variable type.

```
int i;
float f=4.7;

i = (int) f; //Now it is 4
```

- **Converting** – `byte()`, `char()`, `int()`, `long()`, `word()`, `float()` functions are used to convert any type of variable to the specified data type.

```
int i = int(123.45); //The result will be 123
```

- **Converting `String` to `float`** – function `toFLoat()` converts `String` type of variable to the `float`. The following example shows the use of this function. If the value cannot be converted because the String doesn't start with a digit, the returned value will be 0.

```
String string = "123fkm";
float f = string.toFLoat(); //The result will be 123.00
```

- **Converting `String` to `Int`** – function `toInt()` converts `String` type of variable to the `Int`. In the following example, the use of this function is shown.

```
String string = "123fkm";
int i = string.toInt(); //The result will be 123
```

Defining New Types

Typedef Specifier

A `typedef` specifier can give another name for existing types or declare a new one. Renaming types is possible, but software development frameworks already have several aliases. It is helpful, however, when combined with enumerations, classes and structures to give them reasonable names and re-use them later in the code to improve their readability. We present more details on structures in the chapter Structures and Classes, but here is an example presenting a reasonable use of the `typedef` specifier.

```
typedef struct {int x; int y;} tWaypoint; //Declare complex type named waypoint
```

```
...
//Declare a variable of the type of tWaypoint
tWaypoint wp1;
```

Enum Declaration

Enumerations are helpful to give meaning to the integer values and present some logic in a code instead of putting numbers into it. It can be, e.g., the device's state, error code, etc. In the case a new enumeration is needed, it is possible to declare one using the enum keyword and specifying a list:

```
enum errorcodes {ER_OK, ER_DOWNLOAD, ER_UPLOAD, ER_NOWIFI};  //define enumeration
...
errorcodes Errorcode;                                         //declare a variable
...
Errorcode = ER_DOWNLOAD;                                      //assign a value
```

The default numbering starts with 0 (ER_OK=0) and increases by 1 with every next item on the enumeration list. However, explicitly defining values represented by the item labels is possible.

```
enum errorcodes {ER_OK=0, ER_DOWNLOAD=3, ER_UPLOAD=4, ER_NOWIFI=1};
```

3.4.2. Operators, Specifiers and Pointers

Operators represent mathematical, relational, bitwise, conditional, or logical data manipulations. There are many operators in the C/C++ language. In this chapter, the most important are presented. Logical operators will be shown in the next chapter as they are used together with conditional statements.

Assignment Operator

- **Assignment operator (=)** – the operator that assigns the value on the right to the variable on the assignment operator's left. The left side should represent the variable (must be able to be modified), and the right side can be the number (constant), variable, or expression. In the case of an expression, its value is first calculated, and then the result is assigned to the variable on the left. The work of an assignment operator can be seen in any of the following operation examples.

Arithmetic Operators

Arithmetic operations are used to do mathematical calculations with numbers or numerical variables. The arithmetic operators are the following.

- **Addition (+)** – one of the four primary arithmetic operations used to add numbers. The addition operator can add only numbers, numeric variables, or a mix of both. The following example shows the use of the addition operator.

```
int result = 1 + 2; //The result of the addition operation will be 3
```

- **Subtraction (-)** – the operation that subtracts one number from another where the result is the difference between these numbers.

```
int result = 3 - 2; //The result of the subtraction operation will be 1
```

- **Multiplication (*)** – the operation that multiplies numbers and gives the result.

```
int result = 2 * 3; //The result of the multiplication operation will be 6
```

- **Division (/)** – the operation that divides one number by another. If the result variable has the *integer* type, the result will always be the whole part of the division result without the fraction behind it. If the precise division is necessary, using the *float* type of variable for this purpose is important.

```
//The result of the division operation will be 3
//(Only the whole part of the division result)
int result = 7 / 2;
//The result of the division operation will be 3.5
float result2 = 7.0 / 2.0;
```

- **Modulo (%)** – the operation that finds the remainder of the division of two numbers.

```
//The result of the modulo operation will be 1,
//Because if 7 is divided by 3, the remaining is 1
int result = 7 % 3;
```

Bitwise Operators

Bitwise operators perform operations on bits in the variable. Among them, there exist bitwise logic operations. It means the same logic function is applied to every pair of bits in two arguments. Bitwise or (|) means that if at least one bit is "1" at the chosen bit position, the resulting bit will also be "1". Bitwise and (&) means that if at least one bit is "0", the resulting bit is "0". Bitwise operators shouldn't be confused with Logic Operators (||), (&&), which operate on a single boolean logic value.

```
byte result = 5 | 8;
; //The operation in numbers gives the result of 13
; //in bits can be shown as follows
; // 00000101b
; // 00001000b
; // ---------
; // 00001101b
```

```
byte result = 5 & 1;
; //The operation in numbers gives the result of 1
; //in bits can be shown as follows
; // 00000101b
; // 00000001b
; // ---------
; // 00000001b
```

Bitwise operators also allow shifting data left («) or right (») chosen number of bit positions. Shifting is often used in embedded programming to access the bit at a specific position. Shifting data one bit left gives the result of multiplication by 2, while shifting one bit right gives the effect of dividing by 2.

```
byte result = 5 << 1;
; //The operation in numbers gives the result of 10
; //in bits can be shown as follows
; // 00000101b
; // 00001010b
```

Compound Operators

Compound operators in C/C++ are a short way of writing down the arithmetic operations with variables. All of these operations are done on integer variables. These operands are often used in the loops when it is necessary to manipulate the same variable in each cycle iteration. The compound operators are the following.

- **Increment (++)** – increases the value of integer variable by one.

```
int a = 5;
a++; //The operation a = a + 1; the result will be 6
```

- **Decrement (- -)** – decreases the value of the integer variable by one.

```
int a = 5;
a--; //The operation a = a – 1; the result will be 4
```

- **Compound addition (+=)** – adds the right operand to the left operand and assigns the result to the left operand.

```
int a = 5;
a+=2; //The operation a = a + 2; the result will be 7
```

- **Compound subtraction (-=)** – subtracts the right operand from the left operand and assigns the result to the left operand.

```
int a = 5;
a-=3; //The operation a = a – 3; the result will be 2
```

- **Compound multiplication (*=)** – multiplies the left operand by the right operand and assigns the result to the left operand.

```
int a = 5;
a*=3; //The operation a = a × 3; the result will be 15
```

- **Compound division (/=)** – divides the left operand with the right operand and assigns the result to the left operand.

```
int a = 6;
a/=3; //The operation a = a / 3; the result will be 2
```

- **Compound modulo (%=)** – takes modulus using two operands and assigns the result to the left operand.

```
int a = 5;
//The result will be the remaining
//Part of the operation a/2; it results in 1
a%=2;
```

- **Compound bitwise *OR* (|=)** – bitwise *OR* operator that assigns the value to the operand on the left.

```
int a = 5;
a|=2; //The operation a=a|2; the result will be 7
```

- **Compound bitwise *AND* (&=)** – bitwise *AND* operator that assigns the value to the operand on the left.

```
int a = 6;
a&=; //The operation a=a&2; the result will be 2
```

& and * Operators: Pointers and References

Simple and complex types can be referred to with pointer variables. A pointer is a variable that holds the address of the variable. The length of the pointer is equivalent to the length of the memory address (usually 16, 32 or 64 bits). A pointer does not contain a value but instead points to the variable (a memory) where the value is stored. A pointer variable must be initialised and dereferenced with Address-Of and Dereferencing operators.
The following example presents a simple type declaration and the use of a pointer variable.

& operator returns an address of a variable.
* operator dereferences a variable (it provides access to a value that the pointer variable points to).

```
int n = 10;     //Declare a variable of type int and initialise it with 10
int *ptr;       //Declare a pointer variable.
                //At this point, *ptr does not contain any address yet,
                //rather some random address or null.
ptr = &n;       //Assign to the pointer ptr an address of the variable n
                //ptr contains now an address of the memory where
                //variable n is located, not a value 10
int k;          //Declare another variable
k = *ptr;       //Assign k a value that is pointed by ptr
```

Simple type variables such as `int`, `double`, `float` and so on are passed to the function arguments as values, so the original value is copied, and a copy is presented to the function code (more on functions one can find in the Sub-programs, Functions). Modifications to the argument do not change the original value but just a copy. This is not the case when passing a complex type, such as an array, as an argument. The importance of pointers is not to be underestimated in this case: you declare a pointer pointing to the array's first element and pass it to the function. Then, by modifying the pointer value (an address), it is possible to refer to the following elements of the array. In this case, any modification to the referred array element modifies an original one, so the change in the value is instant. It does not need a return variable from a function.

3.4.3. Program Control Statements, Logical operators

It is essential to understand that if no statements change the normal program flow, the microcontroller executes instructions one by one in the order they appear in the source code (from the top - to the down direction). Control statements modify normal program flow by skipping or repeating parts of the code. Often, to decide if the part of the code should be executed or to choose one of the number of possible execution paths, conditional statements are used. For repeating the part of the code, loop statements can be used.

Conditional Statement

if is a statement that checks the condition and executes the following statement if the condition is *TRUE*. There are multiple ways to write down the ***if*** statement:

```
//1st example
if (condition) statement;

//2nd example
if (condition)
statement;

//3rd example
if (condition) { statement; }

//4th example
if (condition)
{
  statement;
}
```

The version with curly braces is used when there is a need to execute part of the code that consists of more than a single statement. Many statements taken together with a pair of curly braces are treated as a single statement in such cases. When both *TRUE* and *FALSE* cases of the condition should be viewed, the ***else*** part should be added to the ***if*** statement in the following ways:

```
if (condition) {
  statement1;    //Executes when the condition is true
}
else {
  statement2;    //Executes when the condition is false
}
```

If more conditions should be viewed, the ***else if*** part is added to the ***if*** statement:

```
if (condition1) {
  statement1;    //Executes when the condition1 is true
}
else if (condition2) {
  statement2;    //Executes when the condition2 is true
}
else {
  statement3;    //Executes in all other cases
}
```

For example, when the *x* variable is compared and when it is higher than 10, the *digitalWrite()* method executes.

```
if (x>10)
{
  //Statement is executed if the x > 10 expression is true
  digitalWrite(LEDpin, HIGH)
}
```

Logical Operators

To allow checking different conditions, logical operators are widely used with the condition statement ***if*** described above.

Comparison Operators

There are multiple comparison operators used for comparing variables and values. All of these operators compare the variable's value on the left to the value on the right. Comparison operators are the following:

- == (equal to) – if they are equal, the result is *TRUE*, otherwise *FALSE*,
- != (not equal to) – if they are not equal, the result is *TRUE*, otherwise *FALSE*,
- < (less than) – if the value of the variable on the left is less than the value of the variable on the right, the result is *TRUE*, otherwise *FALSE*,
- < = (less than or equal to) – if the value of the variable on the left is less than or equal to the value of the variable on the right, the result is *TRUE*, otherwise *FALSE*,
- > (greater than) – if the value of the variable on the left is greater than the value of the variable on the right, the result is *TRUE*, otherwise *FALSE*,
- > = (greater than or equal to) – if the value of the variable on the left is greater than or equal to the value of the variable on the right, the result is *TRUE*, otherwise *FALSE*.

Examples:

```
if (x==y){ //Equal
  //Statement
}

if (x!=y){ //Not equal
  //Statement
}

if (x<y){ //Less than
```

```
  //Statement
}

if (x<=y){ //Less than or equal
  //statement
}

if (x>y){ //Greater than
  //Statement
}

if (x>=y){ //Greater than or equal
  //Statement
}
```

Boolean Operators

The Boolean logical operators in C/C++ are the following:

- ! (logical *NOT*) – reverses the logical state of the operand. If a condition is *TRUE* the logical NOT operator will turn it to *FALSE* and the other way around,
- && (logical *AND*) – the result is *TRUE* when both operands on the operator's left and right are *TRUE*. If even one of them is *FALSE* the result is *FALSE*,
- || (logical *OR*) – the result is *TRUE* when at least one of the operands on the operator's left and right is *TRUE*. If both of them are *FALSE*, the result is *FALSE*.

Examples:

```
//Logical NOT
if (!a) { //The statement inside if will execute when the a is FALSE
  b = !a; //The reverse logical value of a is assigned to the variable b
}

//Logical AND
//The statement inside if will execute when the
//Values both of the a and b are TRUE
if (a && b){
  //Statement
}

//Logical OR
//The statement inside if will execute when at least one of the
//a and b values are TRUE
if (a || b){
  //Statement
}
```

Switch Case Statement

A switch statement similar to the *if* statement controls the flow of a program. The code inside *switch* is executed in various conditions. A *switch* statement compares the values of a variable to the specified values in the *case* statements. Allowed data types of the variable are *int* and *char*. The *break* keyword exits the *switch* statement.

Examples:

```
switch (x) {
   case 0:  //Executes when the value of x is 0
   // statements
   break;   //Goes out of the switch statement

   case 1:  //Executes when the value of x is 1
   // statements
   break;   //Goes out of the switch statement

   default:  //Executes when none of the cases above is true
   // statements
   break;   //Goes out of the switch statement
}
```

3.4.4. Loops

Loops are critical to control flow structures in programming. They allow executing statements or some part of the program repeatedly to process elements of data tables and texts, making iterative calculations and data analysis. In the world of microcontrollers, where sometimes there is no operating system, the whole software works in the main loop called a super loop. It means the program never ends and works until the power is off.
This is clearly visible in the Arduino programming model, with one part of the code executed once after power-on `setup()`, and another executed repeatedly `loop()`. In C/C++, there are three loop statements shown in this chapter.

for

`for` is a loop statement that specifies the number of execution times of the statements inside it. Each time all statements in the loop's body are executed is called an **iteration**. In this way, the loop is one of the basic programming techniques used for all programs and automation in general.

The construction of a `for` loop is the following:

```
for (initialization ; condition ; operation with the cycle variable) {
  //The body of the loop
}
```

Three parts of the `for` construction are the following:

- **initialisation** section usually initialises the value of the cycle variable that will be used to iterate the loop; the initialisation value is ofter 0 but can be any other value,
- **condition** allows managing the number of loop iterations; the statements in the body of the loop are executed when the condition is TRUE,
- **operation with the cycle variable** specifies how the cycle variable is modified every iteration (incremented, decremented); allows defining the number of loop iterations.

The example of the `for` loop:

```
for (int i = 0; i < 4; i = i + 1)
{
    digitalWrite(13, HIGH);
    delay(1000);
    digitalWrite(13, LOW);
    delay(1000);
}
```

On the initialisation of the `for` loop, the cycle variable `i = 0` is defined. The condition states that the `for` loop will be executed while the variable `i` value will be less than 4 (`i < 4`). In operation with the cycle variable, it is increased by 1 each time the loop is repeated.

In the example above, the Arduino function digitalWrite is used. It sets the logical state high or low at the chosen pin. If an LED is connected to pin 13 of the Arduino board, it will turn on/off four times.

while

`while` loop statement is similar to the `for` statement but does not contain the cycle variable. Because of this, the `while` loop allows executing a previously unknown number of iterations. The loop management is realised using only **condition** that needs to be TRUE for the next cycle to execute.

The construction of the `while` loop is the following:

```
while (condition is TRUE)
{
  //The body of the loop
}
```

That way, the `while` loop can be used as a good instrument for the execution of a previously unpredictable program. For example, if it is necessary to wait until the signal from pin 2 reaches the defined voltage level = 100, the following code can be used:

```
int inputVariable = analogRead(2);
while (inputVariable < 100)
{
    digitalWrite(13, HIGH);
    delay(10);
    digitalWrite(13, LOW);
    delay(10);
    inputVariable = analogRead(2);
}
```

In the loop above, the LED that is connected to pin 13 of the Arduino board will be turned on/off until the signal reaches the specified level.

do...while

The do…`while` loop works similarly to the `while` loop. The difference is that in the `while` loop, the condition is checked before entering the loop, but in the do…`while`, the condition is checked after the execution of the statements in the loop, and then if the condition is TRUE the loop repeats. As a result, the statements inside the loop will execute at least once, even if the test condition is FALSE.

The construction of a do while loop is the following:

```
do {
  //The body of the loop
} while (a condition that is TRUE);
```

If the same code is taken from the while loop example and used in the do…while loop, the difference is that the code will execute at least once, even if the inputVariable value is more than or equal to 100. The example code:

```
int inputVariable = analogRead(2);
do {
    digitalWrite(13, HIGH);
    delay(10);
    digitalWrite(13, LOW);
    delay(10);
    inputVariable = analogRead(2);
} while (inputVariable < 100);
```

3.4.5. Sub-programs, Functions

In many cases, the program grows to a size that becomes hardly manageable as a single unit. It isn't easy to navigate through the code that occupies many screens. In such a situation, subprograms can help. Subprograms are named functions in C and C++; while they are associated with an object, they are called methods (in this chapter, the name *function* will be used). The function contains a set of statements that usually form some logical part of the code that can be separately tested and verified, making the whole program easy to manage. Grouping many functions by creating a library stored in a separate file is possible. This is how external libraries are constructed.

Functions

Functions are the set of statements that are always executed when the function is called. A function can accept arguments as input data and return the resulting value. Two functions from the Arduino programming model mentioned before are already known – *setup()* and *loop()*. The programmer usually tries to make several functions containing all the statements and then calls them in the *setup()* or *loop()* functions.

The structure of the function is as follows:

```
type functionName(arguments) //A return type, name, and arguments of the function
{
  //The body of a function – statements to execute
}
```

For example, a function that periodically turns on and off the LED can look like this:

```
void exampleFunction()
{
  digitalWrite(13, HIGH); //the LED is ON
  delay(1000);
  digitalWrite(13, LOW);  //the LED is OFF
  delay(1000);
}
```

The example above shows that the return type of a*exampleFunction* function is *void*, which means the function does not return any value. This function also does not have any arguments because the brackets are empty.

This function should be called inside the *loop()* function in the following way:

```
void loop()
{
  exampleFunction(); //the call of the defined function inside loop()
}
```

The whole code in the Arduino environment looks like this:

```
void loop()
{
  exampleFunction(); //the call of the defined function inside loop()
}

void exampleFunction()
{
  digitalWrite(13, HIGH); //the LED is ON
  delay(1000);
  digitalWrite(13, LOW);  //the LED is OFF
  delay(1000);
}
```

It can be seen that the function is defined outside the *loop()* or *setup()* functions.

When some specific result must be returned as a result of a function, then the function return type should be indicated, for example:

```
//the return type is "int"
int  sumOfTwoNumbers(int x, int y)
{
    //the value next to the "return" should have the "int" type;
    //this is what will be returned as a result.
    return (x+y);
}
```

In the *loop()*, this function would be called in the following way:

```
void loop()
{
  //the call of the defined function inside the loop()
  int result = sumOfTwoNumbers(2, 3);
}
```

Built-in functions

Every programming SDK, including Arduino IDE, comes with several ready-made functions that help develop applications, significantly reducing the effort and time of writing programs. These functions are written to handle inputs and outputs, process texts, communicate using serial ports, manipulate bits and bytes, and perform mathematical calculations. Refer to Arduino or other SDK documentation for details.

Library functions

The popularity of microcontrollers and embedded programming caused the growth of communities of enthusiasts who create a vast of helpful software. This usually comes as a set of functions designed to handle specific tasks, e.g. interfacing with a family of graphical displays or communicating using the chosen protocol. Functions created for one purpose are grouped, forming the library. The number of libraries and their different version is so significant that software developers use a particular library manager to ensure that libraries are up-to-date or keep them in stable versions.

Function handlers

In the MCU world, is is common to use libraries that require a user (software developer) to implement a specific part of the code that is later automatically called by the library routines. Those functions are frequently called handlers and enable developers to inject their actions for a predefined set of activities without modifying library code. For this reason, the library contains a placeholder variable that can be assigned an executable code (a function body). This is handled with the use of pointers. A sample function handler variable is presented in the following code, along with the user function definition, assignment to the handler variable and a call to the handler:

```
int (*hUserImplementedFunction)(int); //Function handler variable
                                      //(no code is here;
                                      //it is just a pointer to the code,
                                      //currently NULL, pointing to "nowhere"
...
int fMulx2(int a) {                   //User's implementation of the function.
  return (2*a);                       //Multiply the argument 'a' value by 2
                                      //and return it to the callee.
  }                                   //Note: argument types and return types
                                      //must match with the variable above
...

hUserImplementedFunction = fMulx2;    //assign a function to the handler
                                      //starting from now,
                                      //hUserImplementedFunction
                                      //contains an address of the fMulx2 function
...
int j;
if (hUserImplementedFunction!=NULL)   //check if the handler is not null
                                      //to avoid NULL pointer exception and code hang
```

```
    j = hUserImplementedFunction(10); //call a handler, j is 20 now
```

In the example above, there is no “&” (address of) operator used when assigning a function code to the handler that is a pointer. It is because, by default, complex types as functions are referenced by reference (a pointer), not by value: `fMulx2` represents an address where the code starts.

Using a function handler is common for asynchronous actions, where user code is notified by the handler (usually low-level library code about the action to happen, e.g. data has been sent via the network interface). This method is similar to interrupts, as described later. Using function pointers (handlers) enables code to modify routines handling actions dynamically by substituting the addresses. Libraries frequently implement handler variables as lists or arrays instead of singular values, enabling adding more than one action (handler) to be called by the library.

3.4.6. Structures and Classes

Structures and classes present complex data types, definable by the developer. Not all C/C++ programming environments provide support for classes (e.g., STM32 in HAL framework mode does not), but luckily, the Arduino framework supports it. Structures, conversely, are part of the C language definition and are present in almost every implementation of software frameworks for IoT microcontrollers.

Structures

In C and C++, a structure is a user-defined data type that allows you to combine different types of variables under a single name. A structure primarily groups related variables, forming a complex data type. A custom data structure (type) that can hold multiple variables, each with its data type. These variables, called members or fields, can be of any built-in or user-defined type, including other structures. The sample named structure (equivalent to the complex type), variable declaration and use of member fields are presented below:

```
struct address {
  String city;
  String PO;
  String street;
  double longitude;
  double latitude;
};
...
address adr1;
```

Note it is also possible to declare a structure variable directly without defining a type:

```
struct {
  String city;
  String PO;
  String street;
  double longitude;
  double latitude;
} adr2, adr3;
```

Structures with type definitions are common when authoring libraries to let library users be able to declare new variables on their own, simply using a type.

Manipulating Structure's Data
Access to the fields of the structure's member variables (short: members, fields) is possible using the "." (dot) operator.

```
adr1.city = "Gliwice";
adr2.city = "Oslo";
adr3.street = "Lime Street";
```

The structure's data can be initialised member by a member or at once using the simplified syntax. Order is meaningful, and types need to fit the definition (C++ only):

```
adr3 = {"Gliwice", "44-100", "1 Lime", -2.973083901947872, 53.401615049766406 };
```

In C++, structures can also have member functions that manipulate the data (in C, they cannot). That is not so far from the Classes idea described in the following chapter. In the case of using C (or poor implementation of C++ that does not support classes nor member functions, e.g. STM32), it is common to prepare a set of data handling functions that operate on the structure referenced with a pointer. A common rule of thumb is the structure is the first argument in the function:

```
struct calcdata
{
  double x,y;
} args;

//Adds x and y of the "arguments" structure
double fCalcDataAdd(calcdata *arguments){
  return (arguments->x + arguments->y);
}
//Multilies x and y of the "arguments" structure
double fCalcDataMul(calcdata *arguments){
  return ((arguments->x)*(arguments->y));
}
//Sets x and y of the "arguments" structure
void fCalcDataSet(calcdata *arguments, double px, double py){
  arguments->x = px;
  arguments->y = py;
}
```

In the examples above, we use a “→” dereference operator to access the member fields by using the pointer to the structure rather than the structure itself.

Sample use of the functions is then:

```
args = {2,7};                //initialise structure x=2, y=7
fCalcDataSet(&args, 12,12);  //reinitialise structure x=12, y=12
int z = fCalcDataAdd(&args); //z equals to 24 now
```

Classes

Classes were introduced in C++ to extend structures encapsulating data and methods (functions) to process this data. A method presented above in the structure context brings an overhead with a need to pass a pointer to the structure for each call. Moreover, it makes access levels tricky, e.g. when you do not want to expose some functions but use them for internal data processing. Thus, classes can be considered as an extension of the structures.

Classes are an important part of IoT programming as they bring an idea of the digital twin to low-level programming: a class used to represent a single piece of hardware, e.g. a sensor and provide its current state, access to the data it grabs and gives access to the operations on the hardware. Thus, in most IoT projects, each device external to the MCU that composes an IoT device is represented by one or more classes on the software side.

Classes are legit only for C++ and not in regular C implementations.

Sample class definition is presented below:

```
class Calculator
{
  public: //you can access this part
  int x,y;
    Calculator() {  //Default constructor
      clear();
    }
    Calculator(int px, int py) { //Another constructor
      x=px;
      y=py;
    }
    ~Calculator(){} //This is dummy destructor
    int Add(){ return x+y; }
    int Mul(){ return x*y; }
    void setX(int px){ x=px; }
    void setY(int py){ y=py; }
  private: //that part is private, and you cannot access it
    void clear(){
      x=0; y=0;
    }
};
```

The code above declares a new type, `Calculator`, with member fields (members in short) `x` and `y` and methods (functions) `Calculator`, `Add`, `Mul`, `setX` and `setY`. Some are marked as `private:` and accessible only from the code of the functions (methods) within the class; some are exposed to external users when marked as `public:`.

Constructors
There are “special” functions whose name is equivalent to the class name in this example above. Those are called constructors and are executed once the object of the class type is instantiated:

```
Calculator calc1=Calculator(2,15);
```

The above code instantiates an object `calc1` of the class `Calculator` and calls the constructor explicitly `Calculator(int px, int py)`. The other constructor, `Calculator()`, is the default one, and if not explicitly called by the code developer, it is automatically called when the object is instantiated.
There can be multiple constructors, and the one executed is selected based on the arguments set.

Destructor
A destructor is called automatically when an object's lifetime is to end. It allows, e.g. to release resources, disconnect open connections, and, in general, do some cleanup before the object is gone. The destructor function in the example above does nothing and is not obligatory in the code. Destructor name starts with a ~ sign (tilde) and has the same name as a class (or constructor):

```
~Calculator(){} //This is dummy destructor
```

In the embedded world, explicitly implemented destructors releasing allocated memory are rare as for the safety of the software, dynamic allocation of the memory is rather to be avoided; thus, destructors are eventually related to the network connections more than memory management.

Members
Member fields can be of any type. When marked as `private` they are accessible only from the code of the constructors, destructor and methods within the class. When `public`, one can reference them using a “.” (dot) operator, as in the case of the structures. When using a pointer to the class instance (object) rather than an instance itself (quite common), a “→” operator works as in the case of structures.

Methods
A method can have any name other than reserved (e.g. for constructors and destructor). Methods marked as `public` are available for the object user and are referenced similarly to member fields ("." and "→" operators). `private` methods are not exposed externally; their purpose is to be called from another method internally. Sample use of methods is presented below:

```
  //continuing initialisation above: calc1.x=2, calc1.y=15
  int z = calc1.Add(); //z=17
  calc1.setX(10);      //calc1.x=10
  calc1.setY(20);      //calc1.y=20
  z = calc1.Mul();     //z=200
```

Class inheritance
Classes can be inherited. This mechanism enables the real power of C++, where existing models (classes) can be extended with new logic without a need to rewrite and fork existing source code. In the example above, the `Calculator` class misses some features, such as subtracting. A code below defines a new type `BetterCalculator` that inherits from the `Calculator` class, using ":" operator:

```
class BetterCalculator:public Calculator
{
  public:
    BetterCalculator() {
    }
    BetterCalculator(int px, int py):Calculator(px,py) {
    }
    int Sub(){return x-y;}
};
```

Members `x` and `y` are in the `Calculator` class. Inheritance before C++ release 11 requires explicit constructor definitions, as in the example above. We use public inheritance to give access to all public methods in the base `Calculator` class available from within the level of the `BetterCalculator` class. Note the `public` keyword in the class definition: `class BetterCalculator:public Calculator`.
Instantiation and use are similar to the presented ones in the previous examples:

```
  BetterCalculator calc2=BetterCalculator(10,6);
                   //BetterCalculator->Calculator->x=10, y=6
  ...
  z = calc2.Sub(); //z=4
  z = calc2.Add(); //z=16 - you can use the underlying code in the Calculator
                   //class without a need to rewrite it again
```

The description above does not deplete all features of C++ Object Oriented Programming. Please note, however, that in the case of the embedded C++, their implementation can be limited and may not contain all the features of the modern, standard C++ patterns.

A special note on the libraries with separate definitions (header) and implementation (body)
Many libraries come with a class definition in the header file (.h) and its implementation in the code file (.cpp). This is convenient for separating use patterns and implementations. A special operator, "::" (double colon), is used in the implementation to refer the code to

the definition in the header file.
The sample header file `myclass.h` with the aforementioned `Calculator` class is present below. It contains only the class definition but does not contain any implementing code.

```
#ifndef h_MYCLASS
#define h_MYCLASS
class Calculator{
  public:                       //you can access this part
  int x,y;
    Calculator();               //Default constructor
    Calculator(int px, int py); //Another constructor
    ~Calculator();              //This is dummy destructor
    int Add();
    int Mul();
    void setX(int px);
    void setY(int py);
  private:                      //that part is private,
                                //and you cannot access it
    void clear();
};
#endif
```

The implementation code refers to the class definition in the header:

```
#include "myclass.h"

Calculator::Calculator() {}
Calculator::Calculator(int px, int py) { x=px; y=py; }
Calculator::~Calculator(){}
int Calculator::Add(){ return x+y;}
int Calculator::Mul(){ return x*y; }
void Calculator::setX(int px){ x=px; }
void Calculator::setY(int py){ y=py; }
void Calculator::clear(){ x=0; y=0; }
```

3.4.7. Timing

B E

Writing code that handles interrupts from internal peripherals, for example, timers, is possible but depends strongly on the hardware.

Time-related functions

Because this chapter presents just an introduction to programming, some essential timing functions will be shown.

Delay
The simplest solution to make functions work for a particular time is to use the `delay()` [42] function. The `delay()` function halts program execution for the time specified as the argument (in milliseconds).

The blinking LED code is a simple demonstration of delay functionality:

```
  digitalWrite(LED_BUILTIN, HIGH);   //Turn the LED on
  delay(1000);                       //Stop program for a second
  digitalWrite(LED_BUILTIN, LOW);    //Turn the LED off
  delay(1000);                       //Stop program for a second
```

Using `delay()` is convenient but has a severe drawback: the algorithm is halted, and only interrupts (or tasks in the background) are executed. The main algorithm is present in the figure 16. Some tasks, e.g. receiving serial transmissions, networking, and outputting set PWM values, continue to work as background tasks, using interrupts or task management (such as FreeRTOS).

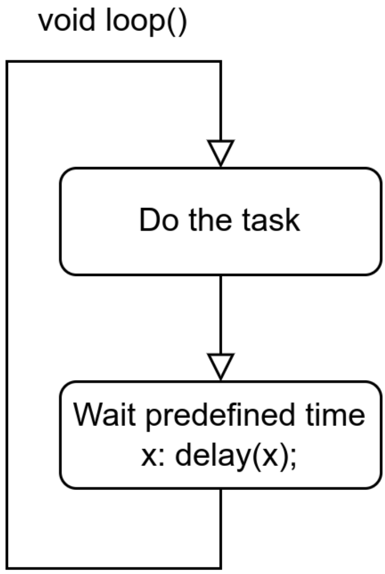


Figure 16: Blocking call: use of the delay()

The alternative to using delay is to switch to the non-blocking method, based on timing with the use of `millis()` as presented below.

Millis

The `millis()` [43] returns the number in milliseconds since MCU began running the current program. Note it has nothing to do with a real-time clock, as most microcontrollers and development boards do not have one. The readings are 32-bit and will roll over in approximately 49 days. `millis()` can be used to replace `delay()` but needs some additional coding. Instead of blocking the algorithm, one can check if the desired time has passed. Meanwhile, it is possible to handle other tasks instead of blocking execution, as presented in the algorithm in figure 17.

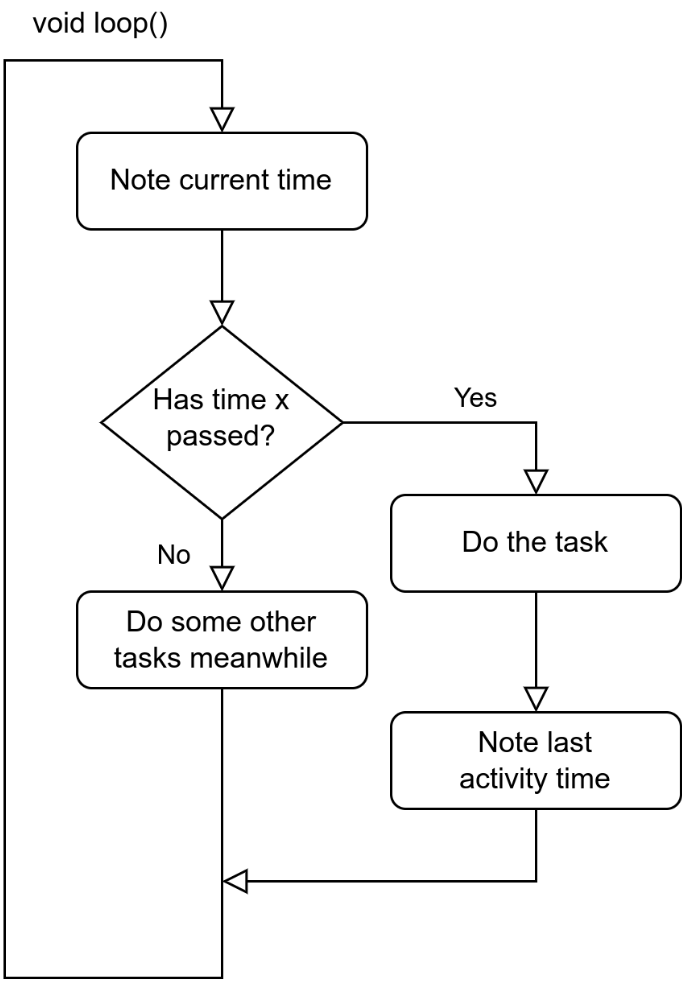


Figure 17: Non-blocking call: use of the millis()

Here is an example code of blinking LED using `millis()`. Millis is used as a timer. Every new cycle time is calculated since the last LED state change. If the time passed is equal to or greater than the threshold value, the LED is switched:

```
//Unsigned long should be used to store time values
//as the millis() returns a 32-bit unsigned number
//Store value of current millis reading

unsigned long currentTime = 0;
//Store value of time when last time the LED state was switched
```

```
unsigned long previousTime = 0;

bool ledState = LOW;               //Variable for setting LED state

const int stateChangeTime = 1000; //Time at which switch LED states

void setup() {
  pinMode (LED_BUILTIN, OUTPUT);  //LED setup
}

void loop() {
  currentTime = millis();         //Read and store current time

  //Calculate passed time since the last state change
  //If more time has passed than stateChangeTime, change the state of the LED
  if (currentTime - previousTime >= stateChangeTime) {

    previousTime = currentTime;   //Store new LED state change time
    ledState = !ledState;         //Change LED state to oposite
    digitalWrite(LED_BUILTIN, ledState); //Write current state to LED
  }
}
```

Sleep Modes

Some IoT-dedicated microcontrollers have special features such as sleep modes that hold program execution for a predefined time or unless an external trigger occurs. This can be used for periodic, time-based activities. Its side effect is energy efficiency. The model of this behaviour and its features are very vendor-specific and vary much: e.g. Espressif MCUs have the only option to restart the code. At the same time, STM32 can hold execution and then continue. Because of the variety of models, modes and features, we do not present here any specific solution but rather a general idea.

3.4.8. Digital ports, reading inputs, outputting data

Every microcontroller has many pins that can be used to connect external electronic elements. In the examples shown in previous chapters, LED was used. Such LED can be connected to a chosen General Purpose Input Output (GPIO) pin and can be controlled by setting a HIGH or LOW state. Below are some details of the functions that allow the manipulation of GPIOs using the Arduino framework. In the next chapter, analogue signals will be considered.

Digital I/O

Microcontrollers' digital inputs and outputs allow for connecting different sensors and actuators to the board. Digital signals can take two values – *HIGH*(1) or *LOW*(0). These states correspond to high voltage (usually corresponding to the power supply voltage of the microcontroller) and low voltage (around 0V). These inputs and outputs are used in applications when the signal can have only two states.

Notice that the voltage the microcontroller is powered with can be different (usually lower) than the voltage provided directly to the board. For example, the ATmega microcontroller on the Arduino Uno board is powered with 5V, while the board itself can be powered from an external source providing 7-12V. Other microcontrollers require different voltages, e.g. Espressif 3.3V. Refer to the device manual for a valid range of voltages.

pinMode()

The function *pinMode()* is essential to indicate whether the specified GPIO pin will behave like an input or an output and may also control special features. This function does not return any value. Usually, the mode of a pin is set in the *setup()* function of a program – only once, during program initialisation.

The syntax of a function is the following:

```
pinMode(pin, mode);
```

The parameter *pin* is the number of the pin.

The parameter *mode* can have three different values – *INPUT*, *OUTPUT*, *INPUT_PULLUP*, depending on whether the pin will be used as an input or an output. The *INPUT_PULLUP* mode turns on the internal pull-up resistor between the power supply and the pin itself. It ensures that if the pin remains unconnected, the logic state will be stable and equal to HIGH. More about pull-up resistors can be found on the Arduino homepage [44].

Most MCUs are pretty flexible in the configuration and use of GPIOS. There are some special GPIOs, however. Some of them cannot work as inputs or outputs or do not have internal pull-up resistors to enable them. Refer to the technical documentation of the hardware you use before setting up your configuration and external devices like sensors, buttons and applications. Some of the GPIOs are also predefined for communication protocols such as Serial, I2C, SPI, etc.

digitalWrite()

The function *digitalWrite()* writes a *HIGH* or *LOW* value to the pin. This function is used for digital pins, such as turning on/off LEDs. This function also does not return any value.

The syntax of a function is the following:

```
digitalWrite(pin, value);
```

The parameter *pin* is the number of the pin. The parameter *value* can take values *HIGH* or *LOW*. If the mode of the pin is set to the *OUTPUT*, the *HIGH* sets voltage to power supply voltage and *LOW* to 0 V.

Using this function for pins set to have the INPUT mode is also possible. In this case, *HIGH* or *LOW* values enable or disable the internal pull-up resistor.

digitalRead()

The function *digitalRead()* works in the opposite direction than the function *digitalWrite()*. It reads the pin's value that can be either *HIGH* or *LOW* and returns it.

The syntax of a function is the following:

```
digitalRead(pin);
```

The parameter *pin* is the number of the pin.

On the opposite of the functions viewed before, this one has the return type, and it can take a value of *HIGH* or *LOW*.

In the code below, the button connected to pin 3 controls the LED connected to pin 4.

```
#define BUTTON_pin 3
#define LED_pin 4

void setup() {
  pinMode(LED_pin, OUTPUT);
  pinMode(BUTTON_pin, INPUT_PULLUP);
}

bool state;

void loop() {
  state = digitalRead(BUTTON_pin); //reading digital state of the input
  digitalWrite(LED_pin, state);    //writing state back to the output
}
```

3.4.9. Manipulating analogue signals

The analogue inputs and outputs are used when the signal can take a range of values, unlike the digital signal that takes only two values (*HIGH* or *LOW*).

Analog input

For measuring the analogue signal, microcontrollers have built-in **analogue-to-digital converter** (ADC) that returns the digital value of the voltage level. Usually, the binary number corresponds to the input voltage, not the value in Volts. The number of bits of the output value depends on the accuracy and internal construction of the converter and usually varies between 8 and 12.

analogRead()

The function *analogRead()* is used for analogue pins (A0, A1, A2, etc.) and reads the value on the analogue pin.

The syntax of a function is the following:

```
analogRead(pin);
```

The parameter *pin* is the pin's name whose value is read.

The return type of the function is the integer value. On the Arduino Uno boards, it ranges between 0 and 1023. The reading of each analogue input takes around 100 ms.

Not every pin can be used as an analogue input. Read the documentation of the chosen development board for details.

Analog output

Unlike analogue input, the analogue output does not generate varying voltage directly on the pin. In general, it uses the technique known as (*Pulse Width Modulation* (*PWM*)) that generates a high/low square signal of stable frequency but varying duty cycle (ratio of active and passive periods of the signal). Details are described later in the book. Because the PMW signal can provide different average power to the external element, e.g. LED, it can be considered analogue output.

analogWrite()

The function *analogWrite()* is used to write an analogue value of the integer type as an output of the pin. An example of use is turning on/off the LED with various brightness levels or setting different speeds of the motors. The value written to the pin stays unchanged until the next value is written to the pin.

The syntax of a function is the following:

```
analogWrite(pin, value);
```

The parameter *pin* is the number of the pin.

The parameter *value* is the PWM signal value that can differ from 0 (off) to 255 (100% on).

This function does not have the return type.

Because an internal timer often generates PWM output, not every pin can be used as analogue output. Read the documentation of the chosen development board for details.

The following example shows reading an analogue value from the A0 input of an Arduino Uno board and writing the analogue value to the output that can control the intensity of the LED.

```
#define LED_pin 3           //the pin number is chosen to support PWM generation

void setup() {
  pinMode(LED_pin, OUTPUT);
}

int value;                  //variable that holds the result of analogue reading

void loop() {
  value = analogRead(A0);   //analogRead on Arduino Uno returns the value 0-1023
  value = value >> 2;       //it should be converted to the value 0-255
  analogWrite(LED_pin, value); //writing converted value to PWM output
}
```

3.4.10. Interrupts

Interrupt is a signal that stops the normal execution of a program in the processor and starts the function assigned to a specific source. This function is called *Interrupt Service Routine* (*ISR*) or interrupt handler. The ISR can be recognized as a task with higher priority than the main program. Interrupt signals can be generated by the external source, like a change of value on the pin, and by the internal source, like a timer or any other peripheral device. When the interrupt signal is received, the processor stops executing the code and starts the ISR. After completing the interrupt handler, the processor returns to the normal program execution state.

ISR should be as short as possible; good practice is avoiding delays and long code sequences. Suppose there is a need to trigger the execution of a long part of the code with an incoming interrupt signal. In that case, the good practice is to define the synchronization variable, modify this variable in the ISR with a single instruction, and handle all other steps in the main program. The interrupt handler does not have arguments and does not return any value, so its type is `void`. To ensure fast execution of the programs, some of the Arduino functions do not work or behave differently in the ISR; for example, the `delay()` function does not work inside the ISR. Variables used in the ISR must be declared as volatile.

Interrupts are used to detect critical real-time events which occur during normal code execution of the code. ISR is executed only when there is a need to do it.

Polling vs. interrupts

Interrupts can help in efficient data transmission. Using interrupts and checking if some situation occurred periodically is unnecessary. Such continuous checking is named polling. For example, a serial port interrupt is executed only when new data comes without polling the incoming buffer in a loop. This approach saves the processor time and, in many situations, creates code that is more energy efficient.

Interrupt handling example

Because interrupts need support from the hardware layer of the microcontroller, the availability of specific interrupt sources depends heavily on the microcontroller model. For example, different Arduino models have different external interrupt pin availability. In most Arduino boards, pins numbered 2 and 3 can be used for interrupts; in Arduino Uno, only these two, while in ESP32 and STM32, almost any digital pin is valid.

Very often, interrupts are used together with hardware timers to generate stable frequency signals. It ensures accurate timing independent of the main loop content and delays. Because internal peripherals are very different for different microcontrollers in this chapter, the example for the external interrupt is shown.

The function `attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)` is called to attach an interrupt to the handler. This function has 3 arguments.

1. `pin` – the pin number where the interrupt signal-generating device will be attached.
2. `ISR` – the name of an ISR function.
3. `mode` – defines when an interrupt signal is triggered. There are five basic `mode` values:
   - `LOW` – interrupt is triggered when the pin value is `LOW`,
   - `HIGH` – interrupt is triggered when the pin value is `HIGH`,
   - `RISING` – interrupt is triggered when the pin value is changed from `LOW` to `HIGH`,
   - `FALLING` – interrupt is triggered when the pin value is changed from `HIGH` to `LOW`,
   - `CHANGE` – interrupt is triggered when the pin value is changed in any direction.

The example program that uses external interrupt:

```
volatile bool button_toggle = 0; //A variable to pass the information
                                 //from ISR to the main program

void setup() {
  pinMode(13,OUTPUT);            //Define LED pin
  pinMode(2,INPUT_PULLUP);       //Define button pin
  attachInterrupt(digitalPinToInterrupt(2),ButtonIRS,FALLING);
                                 //Attach interrupt to button pin
}

void ButtonIRS() {               //IRS function
  button_toggle =!button_toggle;
}

void loop() {
  digitalWrite (13,button_toggle);
}
```

In this example, the code needed to handle the interrupt signal is just one instruction. Still, it shows how to use the synchronization variable to pass information from ISR to the main program, keeping the ISR very short.

3.4.11. Programming patterns

This chapter presents some programming templates and fragments of the code that are common in embedded systems. Some patterns, such as non-blocking algorithms, do not use `delay(x)` to hold program execution but use a timer-based approach instead. It has also been discussed in other chapters, such as in the context of timers Timing or interrupts Interrupts.

Tracing vs Debugging - Serial Ports

Almost any MCU has a hardware debugging capability. This complex technique requires an external debugger using an interface such as JTAG. Setting up hardware and software for simple projects may not be worth a penny; thus, the most frequent case is tracing over debugging. Tracing uses a technique where the Developer explicitly sends some data to the external device (usually a terminal, over a serial port, and eventually a display) that visualises it. The Developer then knows the variables' values and how the algorithm runs. The use of the serial port is common because this is the one that is most frequently used for programming. Thus, it can be used in reverse for tracing. For this reason, Arduino Framework implements a singleton object `Serial` present in every code. It is implemented by each Arduino Framework vendor at the level of the general library with Arduino Framework.
Note to use a `Serial`, it is obligatory to initialise it using the `Serial.begin(x)` method, providing the correct bps, where `x` is a transmission speed (rate) that suits the rate configured in the terminal. The most common rates are 9600 (default) and 115200, but other options are possible. On the terminal side, configuration is usually done in the menu or a configuration file, such as in the case of the `platformio.ini` file. Calling `Serial.begin(x)` is usually done as one of the first actions implemented in the `Setup()` function of the application code:

```
void setup(){
  delay(100);
  Serial.begin(115200);
  Serial.println();
  ...
}
```

A rule of thumb is that after programming and during a boot, every MCU drops some garbage to the serial buffer. That is visualised as several random characters in the terminal. To easily distinguish the tracing from the garbage, it is advised to put some `delay(100)` at the beginning of the code and drop one or two “new line” characters to scroll garbage up using dummy `println()` call (once or twice is usually enough).

The `Serial` object has several handy methods that can help represent various variable types in a textual form to be sent via a serial port to the terminal. The most common are:

- `Serial.print(x)` where `x` is any simple type available in the Arduino Framework, such as integers and floats, but also visualises arrays of characters and `String` objects.
- `Serial.println(x)` prints as above but adds the end of line/newline character by the end of the transmission. Note that the Linux style is used in Arduino, so only ASCII 13 character is sent.

Interfacing with the Device - Serial Port

The serial port and a class `Serial` handling the communication are bi-directional. It means one can send a message from the MCU to the terminal and the opposite. This can be used as a simple user interface. All configuration above steps to ensure seamless cooperation of the MCU serial interface and terminal (application) are also in charge here. As data is streamed byte by byte, it is usually necessary to buffer it. Technically, the serial port notifies the MCU every time a character comes to the serial port using the interrupts. Luckily, part of the job is done by the `Serial` class: all characters are buffered in an internal buffer, and one can check their availability using `Serial.available()`. This function returns the number of bytes received so far from the external device (here, e.g. a terminal) connected to the corresponding serial port.

Many MCUs provide hardware and software serial ports and allow multiple ports to be used. However, one serial port is usually considered the main one and is used for programming (flashing) the MCU. It is also common that other ports are implemented as software ones, so they put extra load on the MCU's processor and resources such as RAM, timers and interrupt system.

Data in the serial port are sent as bytes; thus, it is up to the developer to handle the correct data conversion. Reading a single byte of the data using `Serial.read()` gets another character from the FIFO queue behind the serial port software buffer. As most communication is done textual way, the `Serial` class has support to ease the reading of the strings: `Serial.readString()`, but use involves some extra logic such as the function may timeout. Also, it may contain the END-OF-LINE / NEXT-LINE characters that should be trimmed before usage [45].

Hardware buttons

Hardware buttons tend to vibrate when switching. This physical effect causes bouncing of the state forth and back, generating, in fact, many pulses instead of a single edge during switching. Getting rid of this is called debouncing. In most cases, switches (buttons) short to 0 (GND) and use pull-up resistors, as in the figure 18.

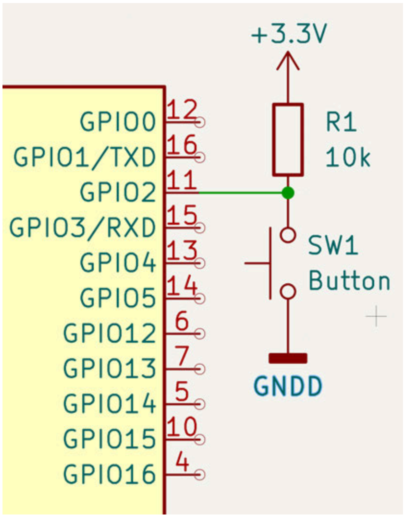


Figure 18: Sample circuit of the switch with an external pull-up resistor connected to the GPIO2 of the MCU

The switch, when open, results in VCC through R1 driving the GPIO2 (referenced as HIGH), and when short, 0 is connected to it, so it becomes LOW:

- button short → GPIO2=LOW,
- button released → GPIO2=HIGH.

Some MCUs offer internal pull-ups and pull-downs, configurable from the software level. The transition state between HIGH and LOW causes bouncing.

A dummy debouncing mechanism only checks periodically for a press/release of the button. The common period for debouncing is between 50ms and 200ms. The code below shows an example that has been provided for presentation purposes. Yet, it is not flexible nor pragmatic due to the exhausting use of the `loop()` function and extensive use of `delay()`. An internal pull-up resistor is in use in this example:

```
#define BUTTON_GPIO 2

bool bButtonPressed=false;

void setup() {
  Serial.begin(9600);
  pinMode(BUTTON_GPIO, INPUT_PULLUP);
}

void loop() {
  if (digitalRead(BUTTON_GPIO)==LOW && !bButtonPressed)
  {
    Serial.println("Button pressed");
    delay(200);
    bButtonPressed=true;
  }
  if (bButtonPressed && digitalRead(BUTTON_GPIO)==HIGH)
  {
    Serial.println("Button released");
    bButtonPressed=false;
    delay(200);
  }
}
```

A more advanced technique for complex handling of the buttons is presented below in the context of the State Machines.

Finite State Machine

A Finite State Machine (FSM) idea represents states and flow conditions between the states that reflect how the software is built for the selected system or its component. An example of button handling using the FSM is present here. The FSM reflects the physical state of the device, sensor or system on the software level, becoming a digital twin of a real device.

For the simple case (without detecting double-click or long press), 3 different button states can be distinguished: released, debouncing and pushed. An enumerator is an excellent choice to model those states (it is easily expandable):

```
typedef enum {
  RELEASED = 0,
  DEBOUNCING,
  PRESSED
} tButtonState;
```

A flow between the states can be then described in the following diagram (figure 19).

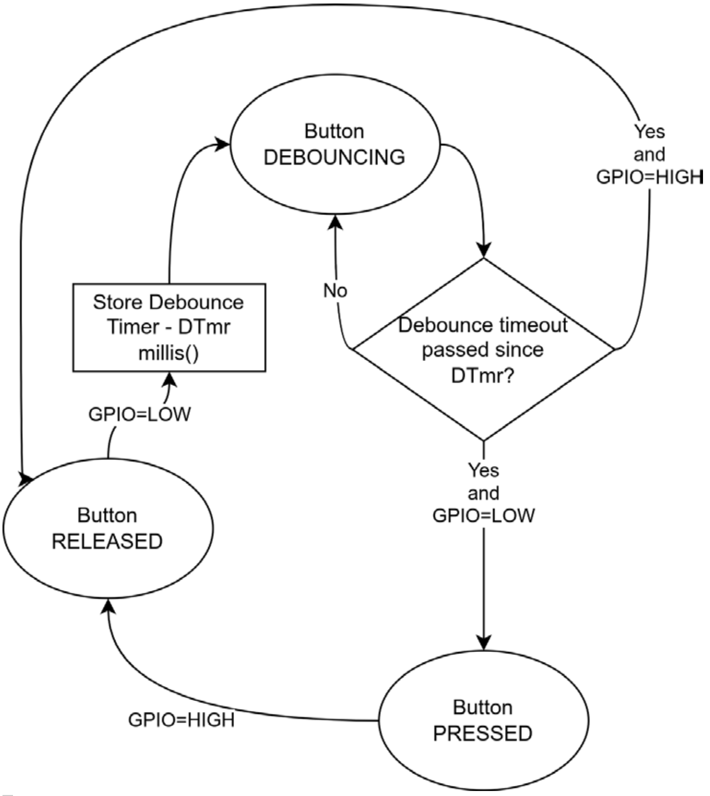


Figure 19: State machine and transitions for button handling with software debouncing

- In the RELEASED state, there is waiting until the button is pressed (LOW, for the pull-up model). The time is noted when it occurs, and the state changes to the DEBOUNCING.
- In the DEBOUNCING state, if debouncing time passes and the button is still pressed (LOW), the machine changes its state to PRESSED. If the button in DEBOUNCING becomes released (HIGH), then the machine returns to the state RELEASED.
- In the PRESSED state, it transits to the RELEASED whenever the button goes HIGH.

The state machine is implemented as a simple class and has 2 additional fields that store handlers for functions that are called when the state machine enters PRESSED or RELEASED. Those functions are called callbacks. There are 2 public functions for callback registration as callback handlers class members are private. `fButtonAction()` is intended to be called in a `loop()` as many times as possible to “catch” all pushes of the button:

```
class PullUpButtonHandler{
  private:
    tButtonState buttonState=RELEASED;
    uint8_t ButtonPin;
    unsigned long tDebounceTime;
    unsigned long DTmr;
    void(*ButtonPressed)(void);    //On button pressed callback
    void(*ButtonReleased)(void);   //On button released callback
    void btReleasedAction() {      //Action to be done
                                   //when current state is RELEASED
      if(digitalRead(ButtonPin)==LOW) {
        buttonState = DEBOUNCING;
        DTmr = millis();
      }
    }
    void btDebouncingAction() {    //Action to be done
                                   //when current state is DEBOUNCING
      if(millis()-DTmr > tDebounceTime)
        if(digitalRead(ButtonPin)==LOW) {
          buttonState = PRESSED;
          if(ButtonPressed!=NULL) ButtonPressed();
        }
        else
          buttonState=RELEASED;
    }
    void btPressedAction() {       //Action to be done
                                   //when current state is PRESSED
      if(digitalRead(ButtonPin)==HIGH) {
        buttonState=RELEASED;
        if(ButtonReleased!=NULL) ButtonReleased();
      }
    }
  public:
    PullUpButtonHandler(uint8_t pButtonPin, unsigned long pDebounceTime) {
                                   //Constructor
      ButtonPin = pButtonPin;
      tDebounceTime = pDebounceTime;
    }
    void fRegisterBtPressCalback(void (*Callback)()) {
                                   //Function registering
                                   //a button PRESSED callback
      ButtonPressed = Callback;
    }
    void fRegisterBtReleaseCalback(void (*Callback)()) {
                                   //Function registering
                                   //a button RELEASED callback
      ButtonReleased = Callback;
    }
    void fButtonAction()           //Main, non blocking loop.
                                   //Handles state machine logic
    {                              //along with private functions above
      switch(buttonState) {
        case RELEASED: btReleasedAction();
          break;
        case DEBOUNCING: btDebouncingAction();
          break;
        case PRESSED: btPressedAction();
          break;
        default:
          break;
      }
    }
};
```

Sample use looks as follows:

```
#define BUTTON_GPIO 2

PullUpButtonHandler bh = PullUpButtonHandler(BUTTON_GPIO, 200);
void onButtonPressed() {
  Serial.println("Button pressed");
}
void onButtonReleased() {
  Serial.println("Released");
}
void setup() {
  Serial.begin(9600);
  pinMode(BUTTON_GPIO, INPUT_PULLUP);
  bh.fRegisterBtPressCalback(onButtonPressed);
  bh.fRegisterBtReleaseCalback(onButtonReleased);
}

void loop() {
  bh.fButtonAction();
}
```

The `PullUpButtonHandler` is instantiated with a 200ms deboucing time. That defines a minimum press time to let the machine recognise the button press correctly. That time is quite long for most applications and can be easily shortened.

The great feature of this FSM is that it can be easily extended with new functions, such as detecting the double click or long button press.

3.4.12. Hardware-specific extensions in programming

Some generic programming techniques and patterns mentioned above require adaptation for different hardware platforms. It may occur whenever hardware-related aspects are in charge, e.g., accessing GPIOs, ADC conversion, timers, interrupts, multitasking (task scheduling and management), multicore management, power saving extensions and most of all, integrated communication capabilities (if any). It can be different for almost every single MCU or MCU family.
It is common for hardware vendors to provide rich examples, either in the form of documentation and downloadable samples (e.g. STM) or via Github (Espressif), presenting specific C/C++ code for microcontrollers.

Analog input

Some MCUs use specific setups. Analogue input may work out of the box. Still, low-level control usually brings better results and higher flexibility (e.g. instead of changing the input voltage to reflect the whole measurement range, you can regulate internal amplification and sensitivity.

A special note on analogue inputs in ESP32
Please note implementation varies even between the ESP32 chips family, and not all chips provide all of the functions, so it is essential to refer to the technical documentation [46].

ESP32 has 15 channels exposed (18 total) of the up to 12-bit resolution ADCs. Reading the raw data (12-bit resolution is the default, 8 samples per measure as default) using the `analogRead()` function is easy.
Technically, under the hood on the hardware level, there are two ADCs (ADC1 and ADC2). ADC 1 uses GPIOs 32 through 39. ADC2 GPIOs 0,2,4, 12-15 and 25-27. Note that ADC2 is used for WiFi, so you cannot use it when WiFi communication is enabled.
Just execute `analogRead(GPIO)`.
Several useful functions are here (not limited to):

- `analogReadResolution(res)` - where `res` is a value between 9 and 12 (default 12). For 9-bit resolution, you get 0..511 values; for 12-bit resolution, it is 0..4095 respectively.
- `analogSetCycles(ccl)` - where `ccl` is number of cycles per ADC sample. The default is 8: the valid number is between 1 and 255.
- `analogSetClockDiv(divider)` - sets base clock divider for the ADC. That has an impact on the speed of conversion.
- `analogSetAttenuation(a)` and `analogSetPinAttenuation(GPIO, a)` - sets input attenuation (for all channels or selected channels). The default is `ADC_11db`. This parameter reflects the dynamic scaling of the input value:
  - `ADC_0db` - no attenuation (1V on input = 1088 reading on ADC), so full scale is 0..1.1V,
  - `ADC_2_5db` - 1.34 (1V on input = 2086 reading on ADC), so full scale is 0..1.5V,
  - `ADC_6db` - 1.5 (1V on input = 2975 reading on ADC), so full scale is 0..2.2V,
  - `ADC_11db` - 3.6 (1V on input = 3959 reading on ADC), so full scale is 0..3.9V.

Do not execute consequent way `analogRead()`. As technically all channels use the same two registers (ADC1 and ADC2), you need to give it some time to sample (e.g. `delay(100)` between consecutive reads on different channels).

Analog output

PWM frequently controls analogue-style, efficient voltage on the GPIO pin. Instead of using a resistance driver, PWM uses pulses to change the adequate power delivered to the actuator. It applies to motors, LEDs, bulbs, heaters and indirectly to the servos (but that works another way).

A special note on ESP32 MCUs
The classical `analogWrite` method, known from Arduino (Uno, Mega) and ESP8266, does not work for ESP32.
ESP32 has up to sixteen (0 to 15) PWM channels (controllers) that can be freely bound to any of the regular GPIOs.

The exact number of PWM channels depends on the family member of the ESP chips, e.g. ESP32-S2 and S3 series have only 8 independent PWM channels while ESP32-C3 has only 6. In the Arduino software framework for ESP32, it is referred to as `ledc`. ESP32 can use various resolutions of the PWM, from 1 to 20 bits, while regular Arduino uses only 8-bit one. Note - there is a strict relation between resolution and frequency: e.g. with high PWM frequency, you cannot go with a resolution too high as the internal frequency of the ESP32 chip is limited.

To use PWM in ESP32, one must perform the following steps:

- configure GPIO pin as `OUTPUT`,
- initiate PWM controller by fixing PWM frequency and resolution,
- bind the controller to the GPIO pin,
- write to the controller (not to the PIN!) providing a duty cycle related to the resolution selected above - every call persistently sets the PWM duty cycle until the next call to the function setting duty cycle.

More information and detailed references can be found in the technical documentation for the ESP32 chips family [47].

Sample code controlling an LED on GPIO 26 with 5kHz frequency and 8-bit resolution is presented below:

```
#include "Arduino.h"

...

#define RGBLED_R 26
#define PWM1_Ch   5
#define PWM_Res   8
#define PWM_Freq  5000

...

ledcSetup(PWM1_Ch, PWM_Freq, PWM_Res);
    //Instantiate timer-based PWM -> PWM channel
ledcAttachPin(RGBLED_R, PWM1_Ch);
    //Bind a PWM channel to the GPIO
ledcWrite(PWM1_Ch,255);
    //Full on: control via the PWM channel, not via the GPIO
...
```

You can bind one PWM channel to many GPIOs to control them synchronously.

This technique can be easily adapted to control, e.g. standard and digital servos. PWM signal specification to control servos is presented in the chapter hardware actuators.

Interrupts

Arduino boards used to have a limited set of GPIOs to trigger interrupts. In other MCUs, it is a rule of thumb that almost all GPIOs (but those used, e.g. for external SPI flash) can trigger an interrupt; thus, there is much higher flexibility in, e.g., the use of user interface devices such as buttons.

A special note on ESP8266 and ESP32

Suppose the interrupt routine (function handler) uses any variables or access flash memory. In that case, it is necessary to use some tagging of the ISR function because of the specific, low-level memory management. A use of IRAM_ATTR is necessary (part of the code present in Interrupts:

```
void IRAM_ATTR ButtonIRS() {  //IRS function
  button_toggle =!button_toggle;
}
```

If the ISR and some other process write to the memory (variable), providing exclusive access to the variable is important. This may be achieved with so-called Muxes, Semaphores and critical sections to ensure no deadlock will occur. However, it is unnecessary if ISR writes to the variable and some other process is reading it. The use of `volatile` for the variable should be enough.

Without an advanced configuration, using the `float` type (hardware accelerated floating point) will cause the application to hang, throwing a panic error and immediate restart of the MCU. It is due to the specific construction of the MCU and FPU. Do not use the `float` type in interrupt handling. If floating point operations are needed, use `double` as this one is calculated the software way.

Timers

The number of hardware timers, their features, and specific configuration is per MCU. Even single MCU families have different numbers of timers, e.g., in the case of the STM32 chips, the ESP32, and many others. Those differences, unfortunately, also affect Arduino Framework as there is no uniform HAL (Hardware Abstraction Layer) for all MCUs so far.

A special note on ESP32 MCUs

The number of hardware timers varies between family members. Most ESP32s have 4, but ESP32-C3 has only two [48]. A timer is usually running at some high speed. The most common is 80MHz and requires a prescaller to be useful. Timers periodically call an interrupt (a handler) written by the developer and bound to the timer during the configuration. Because interrupt routines can run asynchronously to the main code and,

most of all, because ESP32s (most) are double core, it is necessary to take care of the deadlocks that can appear during the parallel access to the shared memory values, such as service flags, counters etc.
Special techniques using the critical section, muxes and semaphores are needed when more than one routine writes to the shared variable between processes (usually main code and an interrupt handler). However, It is unnecessary in the scenario where the interrupt handler writes to the variable and some other code (e.g. in the `loop()` section reads it without writing, as in the case of the example presented below.
In this example, the base clock for the timer in the ESP32 chip is 80MHz, and the timer (tHBT - short from Hear Beat Timer) runs at the 1MHz speed (PRESCALLER is 80) and counts up to 2 000 000. So, the interrupt handler is effectively called once every 2 seconds. This code runs separate from the `loop()` function, asynchronously calling the `onHBT()` interrupt handler.
`onHBT()` interrupt handler swaps the boolean value every two seconds. The value then is translated by the main `loop()` code to drive an LED on the ESP32 development board (here it is GPIO 0), switching it on and off. The `onHBT()` handler function could directly drive the GPIO to turn the LED on and off. Still, we present a more complex example with a `volatile` variable `LEDOn` just for education purposes.

```
#include "esp32-hal-timer.h"

#define LED_GPIO 0        //RED LED on GPIO 0 - vendor-specific
#define PRESCALLER 80     //80MHz->1MHz
#define COUNTER 2000000   //2 million us = 2s

volatile bool LEDOn = false;
hw_timer_t *tHBT = NULL; //Heart Beat Timer

void IRAM_ATTR onHBT(){  //Heart Beat Timer interrupt handler
  LEDOn = !LEDOn;        //Change true to false and opposite;
                         //every call
}

void setup() {
  Serial.begin(9600);
  pinMode(LED_GPIO, OUTPUT);

  tHBT = timerBegin(0, PRESCALLER, true);
    //Instantiate a timer 0 (first)
    //Most ESP32s (but ESP32-C3) have 4 timers (0-3),
    //and ESP32-C3 has only two (0-1).
  if (tHBT==NULL) //Check timer is created OK, NULL otherwise
  {
    Serial.println("Timer creation error! Rebooting...");
    delay(1000);
    ESP.restart();
  }
  timerAttachInterrupt(tHBT, &onHBT, true);
    //Attach interrupt to the timer
  timerAlarmWrite(tHBT, COUNTER, true);
    //Configure to run every 2s (2000000us) and repeat forever
  timerAlarmEnable(tHBT);

}
//Loop function only reads LEDOn value and updates GPIO accordingly
void loop() {
  digitalWrite(LED_GPIO, LEDOn);
}
```

Timers can also be used to implement a Watchdog. Regarding the example above, it is usually a “one-time” triggered action instead of a periodic one. All one needs to do is to change the last parameter of the `timerAlarmWrite` function from `true` to `false`.

3.5. Programming with the Use of Scripts

Several programming models for IoT script programming are available. Depending on the hardware model used (SoC or OS-based MCU), it may involve single script execution (e.g. Raspberry Pi Pico RP2040, Edge-class IoT) or multithreaded, parallel, multiple scripts, doing multiple tasks (e.g. Raspberry Pi 4, Fog-class IoT). The idea and model of the scripting programming for SoC class devices (edge) were presented in the chapter Script Programming with Middleware.
In the case of far more powerful, Fog-class IoT devices that are OS-based devices, a variety of programming languages and, thus, scripting interpreters are available.

Among others, the most common scripting languages for fog class devices are :

- Bash scripting (OS command scripting) usually does not provide support for the GPIO, intended to automate OS tasks,
- Python scripting, cross-platform for both Edge-class devices (Micrpython) and Fog-class (regular Python, usually run on Linux),
- C#, limited to the Windows IoT for Raspberry Pi.

Bash scripting

As Bash scripting is well covered by many manuals for Linux, in the following chapters, we focus on two others: Python and C#. Moreover, accessing the GPIO in the case of the bash requires installing external tools; thus, it does not apply to IoT programming straightforwardly but rather as a supplementary tool to automate tasks other than core programming.

Python

Python programming for IoT devices is dual:

- Regular Python interpreter can be used in Fog class devices such as Raspberry Pi and its clones. In this case, the Python interpreter is run as a separate process in the Linux OS, the same way as in regular PCs. It has full access to the GPIO, however.
- Micropython is dedicated to SoCs and is distributed as the firmware that must be flashed into the device. Commonly, Micropython exposes serial communication on dedicated pins, exposing a Python console that looks similar to the command line Python interface in the PCs

C# .NET

When writing this publication, the .NET framework with C# interpreter is available only for Raspberry Pi devices as a part of the Windows IoT operating system [49]. It is a niche, still fully functional and solid solution. Its newer versions are available only as a commercial product, however.

3.5.1. Python Fundamentals for IoT

A program in Python is stored in text files on the device's file system, as Python's source code is interpreted, not compiled, opposite to C++. A typical file extension for programs in Python is `.py`. In the context of IoT programming, both Python and Micropython share the same syntax and mostly the same libraries, so source code, in many cases, is portable. General hardware-related libraries like GPIO handling or timers are shared between those two Python worlds, and hardware-specific differences are minor compared to the Arduino framework.
Python is simple and efficient in programming the not-so-complex IoT algorithms but does not offer the level of control needed in real-time applications. It can be easily used for prototyping, testing hardware and implementing simple tasks.

Python is the language of the first choice when it comes to AI applications. Most autonomous devices (such as cars) run, in fact, Python code along with C++, hardware accelerated, on IoT fog class devices such as the NVIDIA Jetson family.

Obviously, Micropython does not contain nor allow the use of very complex libraries and frameworks that sometimes are provided to the developer with only binary backend (that is CPU or MCU specific) such as Tensorflow for AI applications.

Nowadays, Python interpreter usually comes with OS (Linux) preinstalled. The sample installation procedure for Raspbian OS is presented in the manual maintained by the Raspberry Pi manufacturer [50]. In the case of the popular Raspbian or Ubuntu for Raspberry Pi, there are usually 2 versions of Pythons preinstalled: Python 2 and Python 3, because of the historical differences between implementations. Many OS applications are written in Python.
Python version can be started from the terminal simply by calling:

```
~$ python --version
Python 3.8.10
```

In the case one needs to use a specific version, you can start the interpreter explicitly referring to Python 2 or Python 3:

```
~$ python2
Python 2.7.18 (default, Jul 29 2022, 09:29:52)
[GCC 9.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
```

```
~$ python3
Python 3.8.10 (default, May 26 2023, 14:05:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
```

Python can be executed via a desktop graphical interface (in the graphical terminal), in a text-based Linux installation via terminal, or remotely via ssh. As it is possible to write applications with visual GUI, starting them in a non-graphical installation of the Linux OS will throw an error. To execute a Python script (program), one needs to execute the following:

```
~$ python mypythoniotapp.py
```

Linux, Windows and Mac systems used to bind a .py file extension with a default Python interpreter, so it is possible to run Python script directly, either with the use of file manager or execute it from the command line:

```
~$ ./mypythoniotapp.py
```

Note: Python script must be marked as “executable” to run it directly.

The following chapters present Python coding elements specific to the microcontrollers. A complete Python course is beyond the scope of this publication, but it can be easily obtained online (links presented by the end of the chapter).

IDEs for Python

A dozen of IDEs can be used to program in Python. The most common are:

- IDLE Editor, formerly delivered with Raspbian OS in the bundle, requires GUI. It is currently obsolete but still popular among hobbyists.
- Thonny Python IDE, which comes with Raspbian OS, recently took over IDLE.
- Visual Studio Code with plugins for Python, standard with Arduino framework, that also easily integrates remote Python development - it provides two development scenarios: local on the IoT device (Raspberry Pi, requires GUI) and remote from the PC to the IoT device, that works with headless Raspberry Pi OSes installations.
- PyCharm Community Edition requires additional installation [51] and requires GUI.
- Simple code can be authored in the terminal using any text editor (e.g. Nano), as Python source files do not require compilation and are plain text ones. This is not very convenient, but it can help if no dedicated IDE and GUI are available, e.g., for rapid work remotely.

Additional Resources for Python programming for beginners

For in-depth Python courses and more, follow the links:

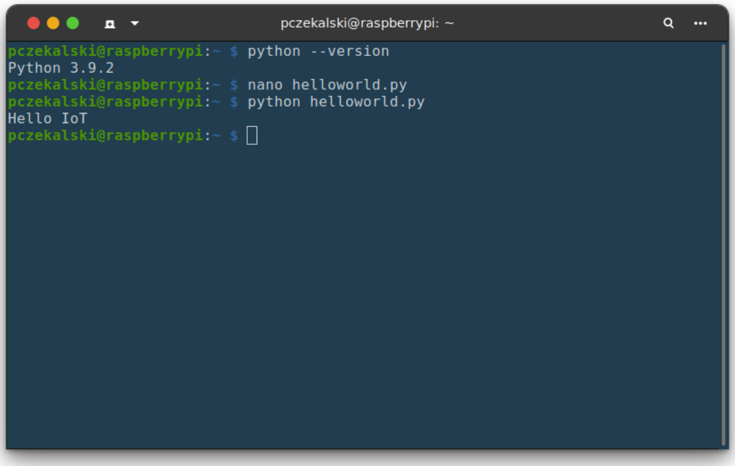
1. The Python syntax and semantics: Python Semantics.
2. The Python Package Index (PyPi): PyPi.
3. The Python Standard Library: PSL.
4. Free online Python course: learnpython.org.

First program in Python

Programming in Python is quite easy. In case you're using a Raspberry Pi with Python installed, once you access the console, you can run the following code:

```
print("Hello IoT")
```

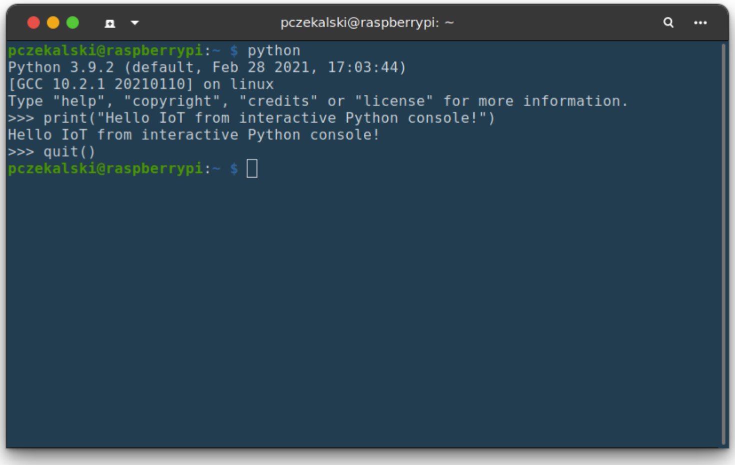
You can store a program in a text file (usually with an extension .py) or interactively type it inline in Python's console.
Assuming your file is stored in the file named `helloworld.py` that contains the code above, an execution and its result are presented on the screen as in figure 20.



```
pczekalski@raspberrypi:~ $ python --version
Python 3.9.2
pczekalski@raspberrypi:~ $ nano helloworld.py
pczekalski@raspberrypi:~ $ python helloworld.py
Hello IoT
pczekalski@raspberrypi:~ $ 
```

Figure 20: Hello World in Python

Python has an interactive console that lets one type the code in. To exit back to the command line, it is necessary to execute the `quit()` function (figure 21).



```
pczekalski@raspberrypi:~ $ python
Python 3.9.2 (default, Feb 28 2021, 17:03:44)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello IoT from interactive Python console!")
Hello IoT from interactive Python console!
>>> quit()
pczekalski@raspberrypi:~ $ 
```

Figure 21: Interactive Python console

An IDE makes programming in Python more comfortable than using a command line. There are many solutions, such as Visual Studio Code + plugins, but perhaps the easiest to use is Thonny. A sample video guide on installing and writing the first program is available on YouTube [52].

Micropython

MicroPython implements the Python programming language optimized for microcontrollers and embedded systems that are resource-constrained devices. It is simple and enables rapid prototyping.

Here are some key features and characteristics of MicroPython:

1. MicroPython is designed to run on devices with limited memory and processing power. It has a small storage and RAM footprint.
2. MicroPython is compatible with Python 3 syntax.
3. MicroPython includes a Read-Eval-Print Loop (REPL), which allows developers to run interactively and test code on a microcontroller line by line and evaluate results immediately. This feature is invaluable for debugging and tracing.
4. MicroPython provides a rich set of libraries and modules for interacting with hardware components such as GPIO pins, sensors, motors, displays and networking capabilities that make it suitable for building IoT devices.
5. MicroPython is designed to be cross-platform. including popular ones like the ESP8266, ESP32, Raspberry Pi Pico, and more. Developers should be aware of the hardware limitations that somehow reduce code portability.
6. MicroPython has a growing community, shared libraries and sample projects. A

package manager called “upip” also enables the installation of additional MicroPython libraries easily.

7. MicroPython is released under an open-source license (typically the MIT License).
8. MicroPython is designed to be power-efficient, crucial for battery-powered and energy-constrained devices.
9. While Micropython is not a real-time operating system (RTOS) itself, it can be used in conjunction with RTOSes to build real-time systems, if only needed.
10. MicroPython is a popular choice for educational purposes: thanks to REPL, the setup of the SDK is simple and quick.

Installation

Installation of Micropython usually involves flashing firmware specific to a microcontroller. This firmware contains a Python interpreter and becomes de facto a middleware between hardware and developer used with scripts. Micropython scripts can be executed inline via terminal (REPL), or a file with source code (usually named `main.py`) can be uploaded to the drive's root folder exposed via USB connection by the MCU firmware.

A website that is a starting point for Micropython is Micropython.org [53].

The installation procedure is specific to the hardware platform and sometimes differs slightly from flashing C++-based solutions or burning an OS, as in the case of the RP2040. The main steps to prepare a working environment are presented below:

- Download a Micropython binary image suitable for your hardware.
- Switch the MCU into the bootloader mode that exposes a flash drive: in the case of the RP2040, the easiest way is to hold down the *Bootsel* button and power on the device while holding.
- Move the firmware file into the flash drive; the device will flash the file to the memory and reboot.
- Connect to the serial port exposed.

Development

Once Micropython is installed in the device, it exposes a terminal (REPL) via serial port, either on the dedicated GPIOs for serial or via serial over USB. Developing directly inline is possible (samples are presented in the following chapters), but it is not convenient for complex code. Complex and multi-file s olutions c an b enefit fr om up loading fil es (even multiple) to the device. A file named `main.py` is a utomatically e xecuted w hen t he device restarts.

IDEs use those features to simplify development and enable remote code authoring and execution.

Sample Micropython development toolchain installation with Thonny IDE one can find on the web [54]. This guide presents development using RP2040.

Python Data Types and Variables

As developers use PCs to author software, but it is executed in the microcontroller, using a terminal over a serial connection or secure shell (SSH) may not be convenient for larger projects. For this reason, many IDEs can perform remote development with code authored in the IDE on the PC but executed on the MCU. It requires a stable connection between the development host and the microcontroller and sometimes installation of the remote development host. One of the most flexible IDEs, able to act in virtually any scenario of remote development, is Visual Studio Code[55].

Programming Raspberry Pi with regular Python can also be executed using development tools installed directly on the RPi, e.g. using Thonny[56]. It is not the case of programming with Micropython that one needs to connect to it remotely. However, installing an external development environment on a PC or Mac computer is usually more convenient. Visual Studio Code comes with a ready solution for both scenarios.

Python
In the case of the RPi programming, VS Code connects to the RPi via Secure Shell (SSH) and installs development tools remotely. The model is present in the figure 22.

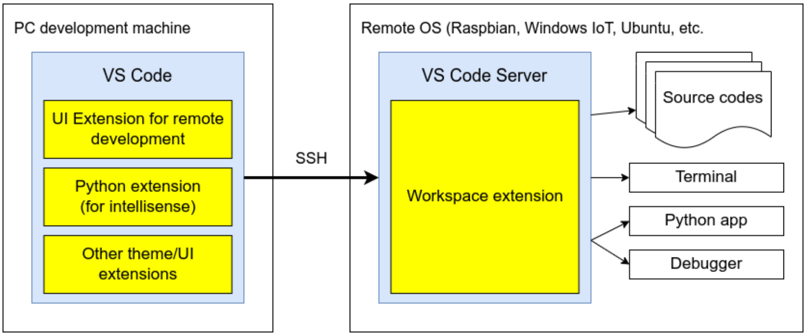
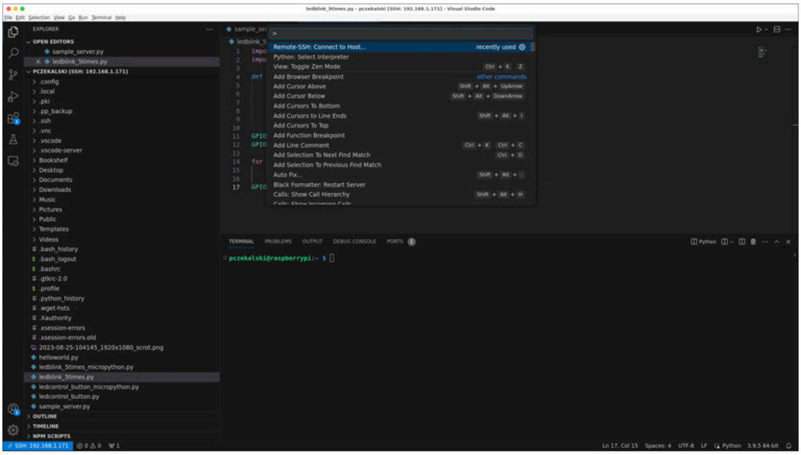

PC development machine
VS Code
UI Extension for remote development
Python extension (for intellisense)
Other theme/UI extensions
SSH
Remote OS (Raspbian, Windows IoT, Ubuntu, etc.
VS Code Server
Workspace extension
Source codes
Terminal
Python app
Debugger

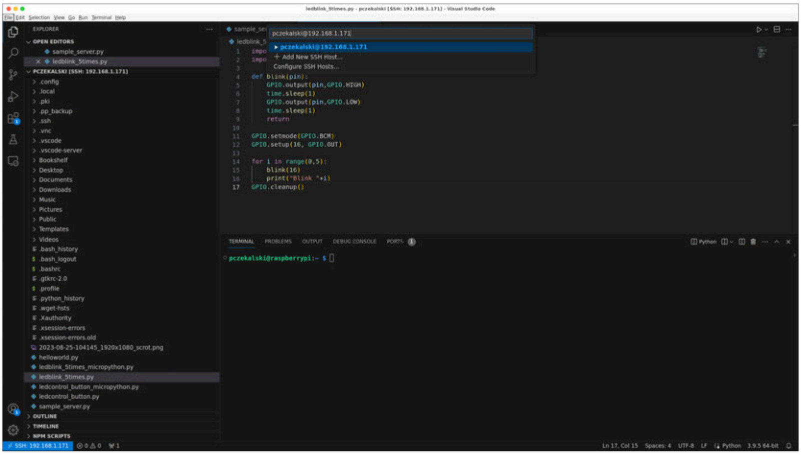
Figure 22: Remote development for RPi with VS Code and Python

Configuring the remote target requires a few simple steps.
Initiate connection to the remote device: over SSH, the target RPi board runs Linux on ARM, here Rasbian, but is also works with other Linux distributions or even remote Docker containers exposing SSH (figures 23 and 24):


ledblink_5times.py - pczekalski [SSH: 192.168.1.171] - Visual Studio Code
Remote-SSH: Connect to Host...
recently used
Python: Select Interpreter
View: Toggle Zen Mode
Add Browser Breakpoint
other commands
Add Cursor Above
Add Cursor Below
Add Cursors To Bottom
Add Cursors to Line Ends
Add Cursors To Top
Add Function Breakpoint
Add Line Comment
Add Selection To Next Find Match
Add Selection To Previous Find Match
Auto Fix...
Black Formatter: Restart Server
Calls: Show Call Hierarchy

```
pczekalski@raspberrypi:~ $
```

Figure 23: Connecting to the remote development target from under VS Code

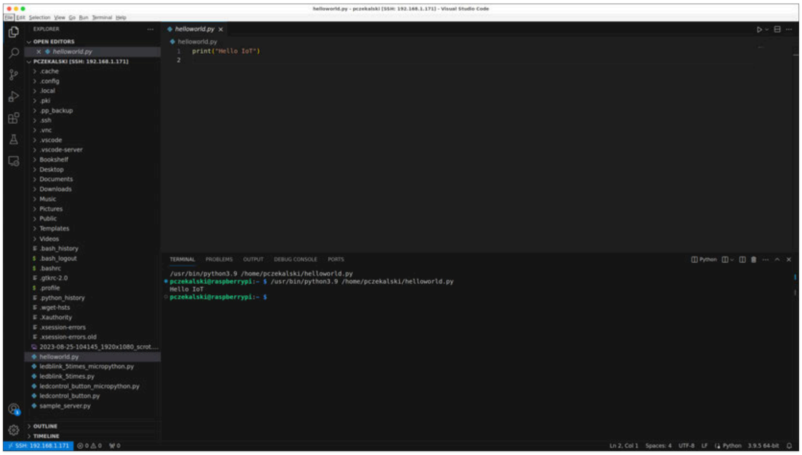


```
impo
impo

def blink(pin):
    GPIO.output(pin,GPIO.HIGH)
    time.sleep(1)
    GPIO.output(pin,GPIO.LOW)
    time.sleep(1)
    return

GPIO.setmode(GPIO.BCM)
GPIO.setup(16, GPIO.OUT)

for i in range(0,5):
    blink(16)
    print("Blink "+i)
GPIO.cleanup()
```

```
pczekalski@raspberrypi:~ $
```

Figure 24: Connecting to the remote development target from under VS Code (cont.)

Remote development is straightforward, as in the case of the local one (figure 25):



```
print("Hello IoT")
```

```
/usr/bin/python3.9 /home/pczekalski/helloworld.py
pczekalski@raspberrypi:~ $ /usr/bin/python3.9 /home/pczekalski/helloworld.py
Hello IoT
pczekalski@raspberrypi:~ $
```

Figure 25: Python remote development for RPi in action, using VS Code

Micropython

In the case of the Micropython, the connection is usually made via a serial port that is exposed either on the device's GPIOs (RX, TX) or as a Serial over USB. Micropython devices commonly expose a filesystem over the USB, alternatively to the Serial over USB. This is usually inducted with an onboard button press during the boot process. Boot system exposition enables an easy firmware (Micropython) update and source file management, such as uploading and downloading the libraries and application source code. A file named 'main.py' is executed automatically on boot if it only exists in the device's root folder and the device is not in the filesystem mode. Some devices (such as RP2040) also provide file management via Serial over USB and thus do not require enabling the filesystem mode manually but rather enable IDE to manage files along with development. For this reason, files can be stored locally and executed with REPL or stored remotely on the Micropython device. A concept of the code development for Micropython is graphically present in the figure 26.

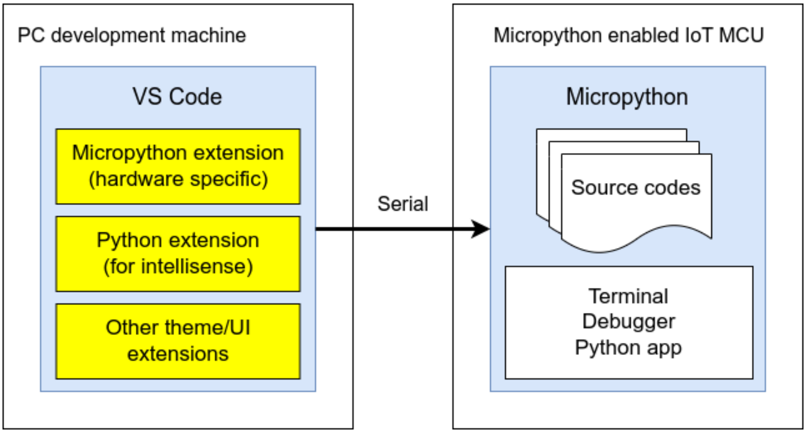
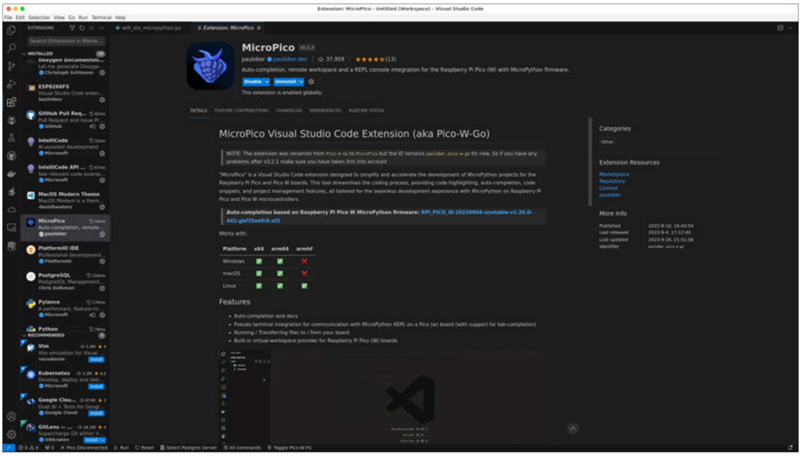

PC development machine
VS Code
Micropython extension (hardware specific)
Python extension (for intellisense)
Other theme/UI extensions
Serial
Micropython enabled IoT MCU
Micropython
Source codes
Terminal
Debugger
Python app

Figure 26: Remote development for Micropython-enabled devices with VS Code and Python (Micropython)

Remote development requires a VS Code plugin to communicate with the Micropython device. One of them is MicroPico, which is dedicated to Raspberry RP2040 (Pico and Pico W), and it can be installed and updated with VS Code extension manager (figure 27).



Auto-completion, remote workspace and a REPL console integration for the Raspberry Pi Pico (W) with MicroPython firmware.

This extension is enabled globally.

MicroPico Visual Studio Code Extension (aka Pico-W-Go)

NOTE: The extension was renamed from Pico-W-Go to MicroPico but the ID remains paulober.pico-w-go for now. So if you have any problems after v3.2.1 make sure you have taken this into account.

"MicroPico" is a Visual Studio Code extension designed to simplify and accelerate the development of MicroPython projects for the Raspberry Pi Pico and Pico W boards. This tool streamlines the coding process, providing code highlighting, auto-completion, code snippets, and project management features, all tailored for the seamless development experience with MicroPython on Raspberry Pi Pico and Pico W microcontrollers.

Auto-completion based on Raspberry Pi Pico W MicroPython firmware: RPI_PICO_W-20230904-unstable-v1.20.0-441-gbf35eefc6.uf2

Works with:

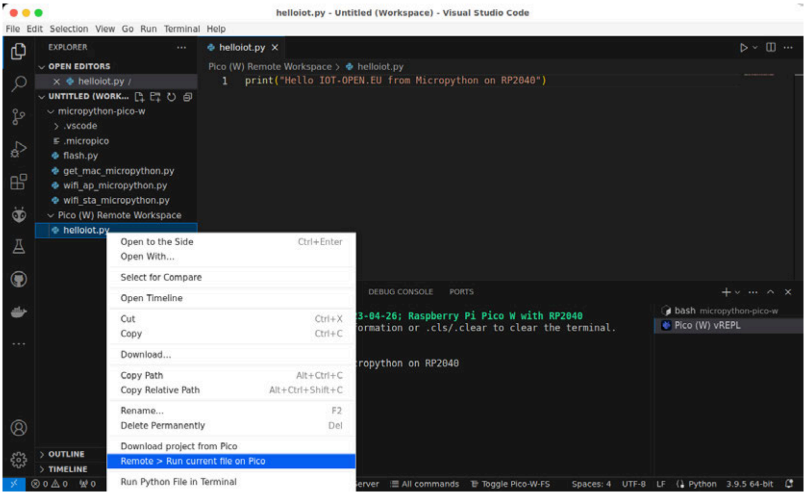
Platform	x64	arm64	armhf
Windows	✓	✓	✗
macOS	✓	✓	✗
Linux	✓	✓	✓

Features

- Auto-completion and docs
- Pseudo terminal integration for communication with MicroPython REPL on a Pico (w) board (with support for tab-completion)
- Running / Transferring files to / from your board
- Built-in virtual-workspace provider for Raspberry Pi Pico (W) boards

Figure 27: MicroPico extension for Visual Studio Code

Code then can be stored locally or remotely and easily executed via the right-click command (figure 28):



```python
print("Hello IOT-OPEN.EU from Micropython on RP2040")
```

```
3-04-26; Raspberry Pi Pico W with RP2040
ormation or .cls/.clear to clear the terminal.

ropython on RP2040
```

Figure 28: Executing Python code on Micropython device (RP2040)

Python Data Types and Variables

Python aims to be consistent and straightforward in the design of its syntax. The best advantage of this language is that it can dynamically set the variable types depending on the values' types as set for variables.

Base Types

Python has a wide range of data types, like many simple programming languages:

- `number`,
- `string`,
- `list`,
- `tuple`
- `dictionary`.

Numbers

Standard Python methods are used to create the numbers:

```
var = 1234     #Creates Integer number assignment
var = 'George' #Creates String type var
```

Python can automatically convert types of the number from one type to another. Type can also be defined explicitly.

```
int a = 10
long a = 123L
float a = 12.34
complex a = 3.23J
<code>

**String**\\
To define Strings, use eclosing characters in quotes.
Python uses single quotes ', double " and triple """ to denote strings.
<code Python>
Name = "George'
lastName = "Smith"
message = """this is the string message which is spanning across multiple lines."""
```

List

The list contains a series of values. To declare list variables, use brackets [].

```
A = [] #Blank list variable
B = [1, 2, 3] #List with 3 numbers
C = [1, 'aa', 3] #List with different types
```

The list indexing is zero-based. Data can be assigned to a specific element of the list using an index into the list.

```
mylist[0] = 'sasa'
mylist[1] = 'wawa'

print mylist[1]
```

Lists aren't limited to a single dimension.

```
myTable = [[],[]]
```

In a two-dimensional array, the first number is always the row number, while the second is the column number.

Tuple
Python Tuples are defined as a group of values like a list and can be processed similarly. When assigned, Tuples got the fixed size. In Python, the fixed size is immutable. The lists are dynamic and mutable. To define Tuples, parenthesis `()` must be used.

```
TestSet = ('Piotr', 'Jan', 'Adam')
```

Dictionary
The list of key-value pairs defines a `Dictionary` in Python. This data type holds related information that can be associated with keys. The Dictionary extracts a value based on the key name. Lists use the index numbers to access its members when dictionaries use a key. Dictionaries generally are used to sort, iterate and compare data.

To define the Dictionaries, the braces (`{}`) are used with pairs separated by a comma (`,`) and the key values associated with a colon (:). Dictionaries Keys must be unique.

```
box_nbr = {'Alan': 111, 'John': 222}
box_nbr['Alan'] = 222     #Set the associated 'Alan' key to value 222'
print (box nbr['John'])   #Print the 'John' key value
box_nbr['Dave'] = 111     #Add a new key 'Dave' with value 111
print (box_nbr.keys())    #Print the keys list in the dictionary
print ('John' in box_nbr) #Check if 'John' is in the dictionary
                          #This returns true
```

All variables in Python hold references to objects and are passed to functions. Function can't change the value of variable references in its body. The object's value may be changed in the called function with the “alias”.

```
>>> alist = ['a', 'b', 'c']
>>> def myfunc(al):
    al.append('x')
    print al

>>> myfunc(alist)
['a', 'b', 'c', 'x']
>>> alist
['a', 'b', 'c', 'x']
```

Python Program Control Structures

if Statements
If an expression returns TRUE statements are carried out; otherwise, they aren't.

```
if expression:
  statements
```

Sample:

```
no = 11
  if no >10:
   print ("Greater than 10")
   if no <=30
     printf ("Between 10 and 30")
```

Output:

```
>>>
Greater than 10
Between 10 and 30
>>>
```

else Statements
An `else` statement follows an `if` statement and contains code called when the if statement is FALSE.

```
x = 2
if x == 6
  printf ("Yes")
else:
  printf ("No")
```

elif Statements
The `elif` (shortcut of `else  if`) statement is used when changing `if` and `else` statements. A series of `if…elif` statements can have a final `else` block, which is called if none of the `if` or `elif` expressions is TRUE.

```
num = 12
if num == 5:
  printf ("Number = 5")
elif num == 4:
  printf ("Number = 4")
elif num == 3:
  printf ("Number = 3")
else:
  printf ("Number = 12")
```

Output:

```
>>>
 Number = 12
>>>
```

Boolean Logic
Python uses logic operators like `AND`, `OR` and `NOT`.

The `AND` operator uses two arguments and evaluates to TRUE if, and only if, both arguments are TRUE. Otherwise, it evaluates to FALSE.

```
>>> 1 == 1 and 2 == 2
True
>>> 1 == 1 and 2 == 3
False
>>> 1 != 1 and 2 == 2
False
>>> 4 < 2 and 2 > 6
False
>>>
```

Boolean operator **or** uses two arguments and evaluates as TRUE if either (or both) of its arguments are TRUE, and FALSE if both arguments are FALSE.

The result of `NOT TRUE` is FALSE, and `NOT FALSE` goes to TRUE.

```
>>> not 2 == 2
False
>>> not 6 > 10
True
>>>
```

Operator Precedence
Operator Precedence uses the mathematical idea of operation order, e.g. multiplication begins performed before addition.

```
>>> False == False or True
True
>>> False == (False or True)
False
>>> (False == False) or True
>>>True
>>>
```

Python Looping

while Loop
An `if` statement is run once if its condition evaluates to TRUE and never if it evaluates to FALSE.

A `while` statement is similar, except it can be run more than once. The statements inside it are repeatedly executed as long as the condition holds. Once it evaluates to FALSE, the next section of code is executed.

```
i = 1
while i<=4:
  print (i)
  i+=1
print ('End')
```

Output:

```
>>>
1
2
3
4
End
>>>
```

The **infinite loop** is a particular kind of the while loop; it never stops running. Its condition always remains TRUE.

```
while 1 == 1:
 print ('in the loop')
```

To end the while loop prematurely, the **break** statement can be used. The break statement causes the loop to finish immediately when encountered inside a loop.

```
i = 0
while 1==1:
  print (i)
  i += 1
  if i >=3:
    print('breaking')
    break;
print ('finished')
```

Output:

```
>>>
0
1
2
3
breaking
finished
>>>
```

Another statement that can be used within loops is **continue**.

Unlike break, continue jumps back to the top of the loop rather than stopping it.

```
i = 0
while True:
  i+=1
  if i == 2:
    printf ('skipping 2')
    continue
  if i == 5:
    print ('breaking')
    break
  print (i)
print ('finished')
```

Output:

```
>>>
1
skipping 2
3
4
breaking
finished
>>>
```

for Loop

```
n = 9
for i in range (1,5):
  ml = n * i
  print ("{} * {} = {}".format (n, i, ml))
```

Output:

```
>>>
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
>>>
```

Python Sub-Programs

One of the most important in mathematics concept is to use functions. The executing function produces one or more results, dependent on the parameters passed to it and provides a re-usable piece of code, still somehow flexible, depending on the parameters. In general, a function is a structuring element in the programming language which groups a set of statements so they can be called more than once in a program. Programming without functions will need to reuse code by copying it and changing its different context. Using functions enhances the comprehensibility and quality of the program. It also lowers the software's memory usage, development cost and maintenance.

Different naming is used for programming language functions, e.g., subroutines, procedures or methods.

Python language defines function by a `def` statement. The function syntax looks as follows:

```
def function-name(Parameter list):
  statements, e.g. the function body
```

Function bodies can contain one or more return statements. It can be situated anywhere in the function body. A return statement ends the function execution and returns the result, e.g. to the caller. If the return statement does not contain an expression, the value None is returned.

```
def Fahrenheit(T_in_celsius):
  """ returns the temperature in degrees Fahrenheit """
  return (T_in_celsius * 9 / 5) + 32

for t in (22.6, 25.8, 27.3, 29.8):
  print(t, ": ", fahrenheit(t))
```

Output:

```
>>>
22.6 :  72.68
25.8 :  78.44
27.3 :  81.14
29.8 :  85.64
>>>
```

Optional Parameters
Functions can be called with optional parameters, also named default parameters. If the function is called without parameters, the default values are used. The following code greets a person. If no person's name is defined, it greets everybody:

```
def Hello(name="everybody"):
  """ Say hello to the person """
  print("Hello " + name + "!")

Hello("George")
Hello()
```

Output:

```
>>>
Hello George!
Hello everybody!
>>>
```

Docstrings
The string is usually the first statement in the function body, which can be accessed with function_name.doc. This is a Docstring statement.

```
def Hello(name="everybody"):
  """ Say hello """
  print("Hello " + name + "!")
print("The docstring of the function Hello: " + Hello.__doc__)
```

Output:

```
>>>
The function Hello docstring:  Say hello
>>>
```

Keyword Parameters
The alternative way to make function calls is to use keyword parameters. The function definition remains unchanged.

```
def sumsub(a, b, c=0, d=0):
  return a - b + c - d
print(sumsub(12,4))
print(sumsub(42,15,d=10))
```

Only keyword parameters are valid, which are not used as positional arguments. If keyword parameters don't exist, the next call to the function will need all four arguments, even if the c needs just the default value:

```
print(sumsub(42,15,0,10))
```

Return Values
In the above examples, the return statement exists in sumsub but not in the Hello function. The return statement is not mandatory. If an explicit return statement doesn't exist in the sample code, it will not show any result:

```
def no_return(x,y):
  c = x + y
res = no_return(4,5)
print(res)
```

Any result will not be displayed in:

```
>>>
```

Executing this script, the None will be printed. If a function doesn't contain an expression, the None will also be returned:

```
def empty_return(x,y):
    c = x + y
    return
res = empty_return(4,5)
print(res)
```

Otherwise, the expression value following return will be returned. In this example, 11 will be printed:

```
def return_sum(x,y):
  c = x + y
  return c
res = return_sum(6,5)
print(res)
```

Output:

```
>>>
9
>>>
```

Multiple Values Returning
Any function can return only one object. An object can be a numerical value – integer, float, list or a dictionary. To return, e.g. three integer values, one can return a list or a

tuple with these three integer values. It means that the function can indirectly return multiple values. The following example calculates the Fibonacci boundary for a positive number returns a 2-tuple. The Largest Fibonacci Number smaller than x is the first, and the Smallest Fibonacci Number larger than x is next. The return value is stored via unpacking into the variables lub and sup:

```python
def fib_intervall(x):
  """ returns the largest Fibonacci number, smaller than x and the lowest
  Fibonacci number, higher than x"""
  if x < 0:
    return -1
  (old, new, lub) = (0,1,0)
  while True:
    if new < x:
      lub = new
      (old,new) = (new,old+new)
    else:
      return (lub, new)

while True:
  x = int(input("Your number: "))
  if x <= 0:
    break
  (lub, sup) = fib_intervall(x)
  print("Largest Fibonacci Number < than x: " + str(lub))
  print("Smallest Fibonacci Number > than x: " + str(sup))
```

Python for Hardware

Using hardware interfaces with Python requires specific binary libraries. Thus, it is not as easily exchangeable among platforms as the hardware-aware part of the code. Below are some hardware-related samples for Python and Micropython.

Controlling GPIO

The following code presents a sample Python application that flashes an LED connected to Raspberry Pi's GPIO pin 16. One must build a circuit (LED + resistor of a proper value) and connect it to the GPIO before running the code.
This example uses a dedicated GPIO handling library (specific for hardware): `RPi.GPIO`. For other IoT platforms, this may vary. For example, Micropython uses a Machine library instead—it covers all the microcontroller's hardware.

```python
import RPi.GPIO as GPIO
import time

def blink(pin):
    GPIO.output(pin,GPIO.HIGH)
    time.sleep(1)
    GPIO.output(pin,GPIO.LOW)
    time.sleep(1)
    return

GPIO.setmode(GPIO.BCM)
GPIO.setup(16, GPIO.OUT)
```

```
for i in range(0,5):
    blink(16)
GPIO.cleanup()
```

A code equivalent for the above algorithm to run in Micropython (here for RP2040) looks quite similar:

```
import machine
import time

led=machine.Pin(16, machine.Pin.OUT)

def blink():
    led.toggle()
    time.sleep(1)
    led.toggle()
    time.sleep(1)

led.value(0)

for i in range(5):
    blink()
```

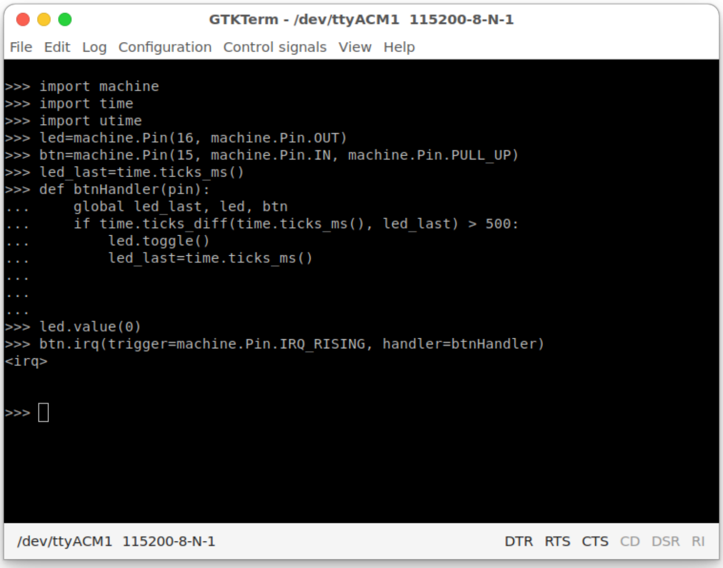
Interrupts Handling

Similarly to the GPIO, interrupts are hardware-specific; thus, libraries may differ among platforms, hence Python syntax. The sample present below is for Raspberry Pi (regular Python), and the following code is for RPi Pico (RP2040, Micropython).

```
import RPi.GPIO as GPIO
import time

led_last = time.time_ns()
led_state = GPIO.LOW

GPIO.setmode(GPIO.BCM)
GPIO.setup(16, GPIO.OUT, initial=led_state)
GPIO.setup(17, GPIO.IN, pull_up_down=GPIO.PUD_UP)

def btnHandler(pin):
    global led_test, led_state
    if (led_state==GPIO.LOW):
        led_state=GPIO.HIGH
    else:
        led_state=GPIO.LOW
    GPIO.output(16, led_state)

GPIO.add_event_detect(17, GPIO.FALLING, callback=btnHandler, bouncetime=200)

while(True):
    time.sleep(0)
```

The sample code for RP2040 Micropython is present below, and its implementation live, via serial port, directly on the MCU is present in the figure 29:

```
import machine
import time
import utime
led=machine.Pin(16, machine.Pin.OUT)
btn=machine.Pin(15, machine.Pin.IN, machine.Pin.PULL_UP)
led_last = time.ticks_ms()

def btnHandler(pin):
    global led, led_last, btn
    if (time.ticks_diff(time.ticks_ms(), led_last)) > 500:
        led.toggle()
        led_last=time.ticks_ms()

led.value(0)
btn.irq(trigger=machine.Pin.IRQ_RISING, handler=btnHandler)
```


GTKTerm - /dev/ttyACM1 115200-8-N-1
File Edit Log Configuration Control signals View Help
/dev/ttyACM1 115200-8-N-1
DTR RTS CTS CD DSR RI

```
>>> import machine
>>> import time
>>> import utime
>>> led=machine.Pin(16, machine.Pin.OUT)
>>> btn=machine.Pin(15, machine.Pin.IN, machine.Pin.PULL_UP)
>>> led_last=time.ticks_ms()
>>> def btnHandler(pin):
...     global led_last, led, btn
...     if time.ticks_diff(time.ticks_ms(), led_last) > 500:
...         led.toggle()
...         led_last=time.ticks_ms()
...
...
...
>>> led.value(0)
>>> btn.irq(trigger=machine.Pin.IRQ_RISING, handler=btnHandler)
<irq>


>>> 
```

Figure 29: Micropython (on RP2040 Pico) hardware interrupt sample, implemented via serial port, inline

Note: there is no need to do a blind, infinite loop by the end of the application in the case of the Micropython inline coding, as code runs infinitely in this example. This is because interrupts run asynchronously once defined. However, this is not the case in the regular RPI Python scripts, where the program is executed as a task within an operating system (Raspbian, Armbian, Ubuntu, etc.), and the script quits if not blocked explicitly with an infinite loop.

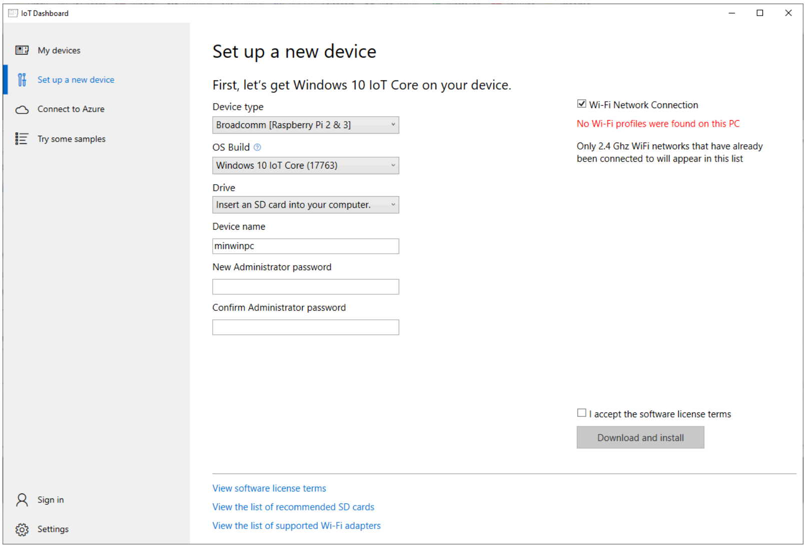
3.5.2. Windows IoT and C# Fundamentals

B M E

Installing the Windows 10 IoT Core

Microsoft Windows 10 IoT OS system is available for download from Windows 10 IoT Core Developers tools [57].

Step 1
Download the Windows 10 IoT Core Dashboard Setup. Run the Setup.exe file and choose the Setup a new device tab as present in figure 30.



Set up a new device

First, let's get Windows 10 IoT Core on your device.

Device type: Broadcomm [Raspberry Pi 2 & 3]

OS Build: Windows 10 IoT Core (17763)

Drive: Insert an SD card into your computer.

Device name: minwinpc

New Administrator password

Confirm Administrator password

☑ Wi-Fi Network Connection

No Wi-Fi profiles were found on this PC

Only 2.4 Ghz WiFi networks that have already been connected to will appear in this list

☐ I accept the software license terms

Download and install

View software license terms

View the list of recommended SD cards

View the list of supported Wi-Fi adapters

Figure 30: Windows Insider Program Win10 IoT Core Setup

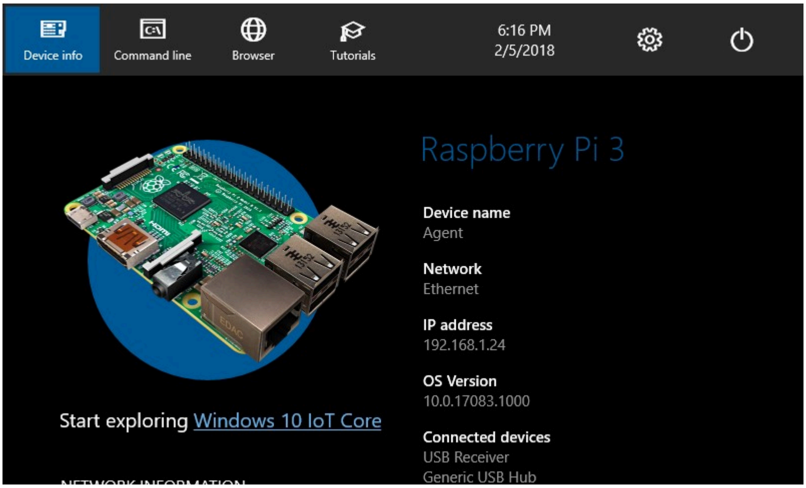
Step 2

On the IoT Dashboard window, the User must choose the device type to install the OS system and the OS version to install (latest developer version 17763 or custom). The device's name and Administrator password can be set during SD card with OS preparation. The default password for the Windows 10 IoT Core is **passw0rd** If the Wi-Fi network Connection is available on the computer running the Setup, it is possible to configure the generated OS image to set this connection on the selected device board automatically.
Accepting the software license terms enables the Download and Install button to format the SD Card and prepare the OS image for the selected device board.
Choosing the Connect to Azure tab and following the Azure User registration opens the ability of the OS image to automatically connect to the Azure IoT hub after the device powers on.

Step 3
Start formatting the SD card and install the FFU image on it.

Step 4
Gently remove an SD card from the reader and push it into the Raspberry Pi SD card slot.

Step 5
Power on the Raspberry Pi board and follow the Windows 10 Core setup commands configuring the Windows 10 Core features.
After the board reboot, the Main Windows 10 Core screen displays (figure 31):



6:16 PM
2/5/2018

Raspberry Pi 3

Device name
Agent

Network
Ethernet

IP address
192.168.1.24

OS Version
10.0.17083.1000

Connected devices
USB Receiver
Generic USB Hub

Start exploring Windows 10 IoT Core

NETWORK INFORMATION

Figure 31: Windows 10 Core system view

Configuring Windows IoT Development Environment

This chapter describes the typical programming technics used in Raspberry Pi board development projects.

Raspberry Pi Under Windows 10 IoT Core

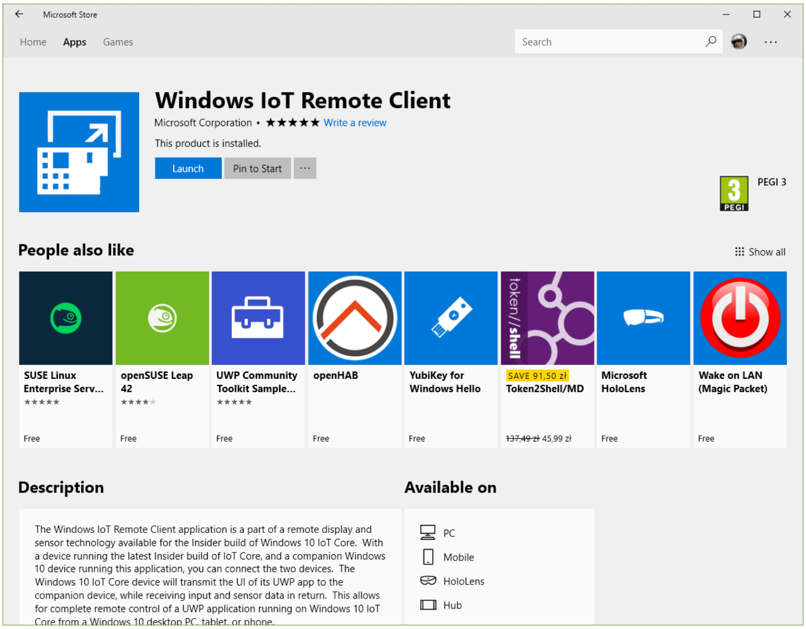
To create and develop control applications on the Raspberry Pi boards, one needs the following development components:

- PC with Windows 10 System installed,
- Visual Studio 2015 or higher,
- Raspberry PI 2 or 3 board with Windows 10 IoT Core installed,
- configured TCP/IP network for Raspberry Pi and Windows 10 Desktop computer (Local LAN or WiFi subnet),

A list of programming skills necessary to seamlessly develop for Windows IoT is listed below:

- C# language knowledge,
- XML/XAML language knowledge,
- Windows API understanding.

The Windows IoT Remote Client is welcome for a better development experience of Raspberry applications. This application is available for download from the Microsoft Store. This application captures the keyboard, mouse and screen from the Raspberry Pi board running the Windows 10 IoT Core system on the desktop PC. It allows developers to use a standalone Raspberry board without a connected mouse, keyboard or monitor (figure 32).


Microsoft Store
Home
Apps
Games
Search
Windows IoT Remote Client
Microsoft Corporation • ★★★★★ Write a review
This product is installed.
Launch
Pin to Start
PEGI 3
People also like
Show all
SUSE Linux Enterprise Serv... Free
openSUSE Leap 42 Free
UWP Community Toolkit Sample... Free
openHAB Free
YubiKey for Windows Hello Free
SAVE 91,50 zł Token2Shell/MD ~~137,49 zł~~ 45,99 zł
Microsoft HoloLens Free
Wake on LAN (Magic Packet) Free
Description
Available on
PC
Mobile
HoloLens
Hub

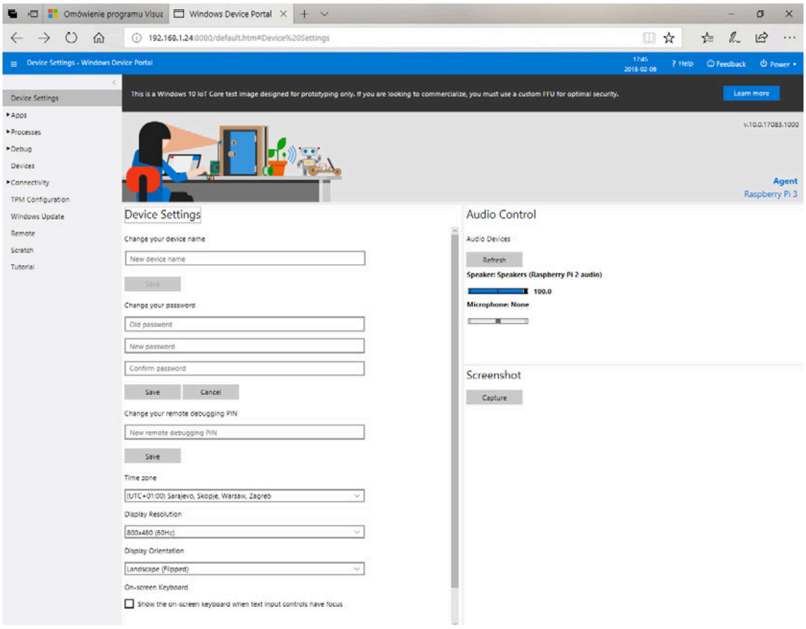
The Windows IoT Remote Client application is a part of a remote display and sensor technology available for the Insider build of Windows 10 IoT Core. With a device running the latest Insider build of IoT Core, and a companion Windows 10 device running this application, you can connect the two devices. The Windows 10 IoT Core device will transmit the UI of its UWP app to the companion device, while receiving input and sensor data in return. This allows for complete remote control of a UWP application running on Windows 10 IoT Core from a Windows 10 desktop PC, tablet, or phone.

Figure 32: Microsoft Store – Windows IoT Remote Client

To write and develop applications under Windows 10 IoT Core, developers must know how the Windows operating system interacts with User applications. The advantage of using Windows 10 IoT Core is that Microsoft's concept uses the same Kernel API available on different hardware platforms – desktop PCs, IoT boards suitable to run Windows Core, tablets, etc. It reduces development costs due to the unifying system environment, and the only difference is in the display view of the same application code written in C#/C++. Windows 10 Core is specially designed to handle applications working as standalone on the IoT platforms in a 24/7 time model.

Configuring the Windows 10 IoT Core Platform

After installing the Windows 10 IoT Core, the developer must configure the IoT platform using Windows Device Portal (WDP, figure 33).



192.168.1.24:8080/default.htm#Device%20Settings

17:45 2018-02-06

This is a Windows 10 IoT Core test image designed for prototyping only. If you are looking to commercialize, you must use a custom FFU for optimal security.

v.10.0.17083.1000

Device Settings

Change your device name: New device name

Change your password: Old password / New password / Confirm password

Change your remote debugging PIN: New remote debugging PIN

Time zone: (UTC+01:00) Sarajevo, Skopje, Warsaw, Zagreb

Display Resolution: 800x480 (60Hz)

Display Orientation: Landscape (Flipped)

On-screen Keyboard: ☐ Show the on-screen keyboard when text input controls have focus

Speaker: Speakers (Raspberry Pi 2 audio) 100.0

Microphone: None

Figure 33: Windows Device Portal view

IoT board can be managed using Chrome, Microsoft Edge, Firefox, and any Internet browser. To open the WDB portal on the IoT board, the user must enter the board IP address – IPaddress:8080/default.htm. The site is protected with a username/password. Default account credentials are: `administrator/p@ssw0rd`. Following tabs in the WDP, it is possible to configure all necessary IoT platform settings, check the current board status, download development crash/debug information, configure network/Bluetooth settings, download drivers, and configure security TPM modules. If all tasks are ready, the developer can start to write his own IoT application under Windows 10 IoT Core.
The following steps are recommended before the IoT board will be used for application:

Step 1

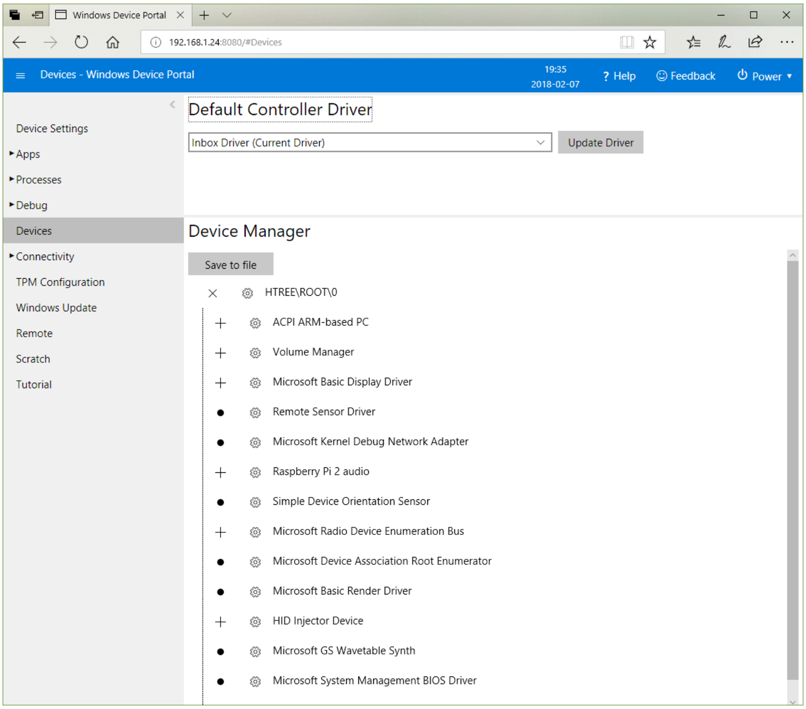
In the `Device Settings`, the user is recommended to `Change your device name`. The default name is `minwinpc`. The aim to change it is that if a user uses many IoT devices connected to the same network segment, it is challenging to recognise which role each device is set for. Enumerating IoT devices will show boards with the same name but different IP addresses. Proper naming will make it easy to know what role each device plays.

Step 2
Because RPI boards don't have their own RTC clock modules, Windows 10 IoT Core sets the time using the NTP services during its work. So, very important in industrial implementations, and a case when time is essential in developed applications, is to set the proper time zone for the board. In the `Device settings`, the user is recommended to select the appropriate `Time zone`

Step 3
Security reasons – the default password for the newly flashed device is `p@ssw0rd`. It is strongly recommended right after the first board boot to change it to make it unique! It will prevent the IoT device from remote hacking. The password can be changed in the `Device Settings` tab.

Step 4
The Windows 10 IoT Core comes with `Cortana` service ready. If the board has a microphone and speakers, it is always possible to turn the Cortana service on for voice commands communication with the board.

Step 5
If the IoT board needs special hardware connected, then in the `Devices/Device Manager`, the user can upload and install an appropriate driver for it in case it is not preinstalled in the IoT Core (figure 34).



Default Controller Driver

Inbox Driver (Current Driver)

Device Manager

```
HTREE\ROOT\0
  ACPI ARM-based PC
  Volume Manager
  Microsoft Basic Display Driver
  Remote Sensor Driver
  Microsoft Kernel Debug Network Adapter
  Raspberry Pi 2 audio
  Simple Device Orientation Sensor
  Microsoft Radio Device Enumeration Bus
  Microsoft Device Association Root Enumerator
  Microsoft Basic Render Driver
  HID Injector Device
  Microsoft GS Wavetable Synth
  Microsoft System Management BIOS Driver
```

Figure 34: Device Manager view

Step 6
Raspberry Pi boards 1/2/3 are equipped with network connection modules. If the board under Windows 10 IoT Core is connected to a LAN RJ45 connector, the IP number can be set via the DHCP server. If the user wants to use a WiFi connection or activate Bluetooth, he can do it directly on the board main display or manage it via the Windows Device Portal as present in figures 35 and 36.

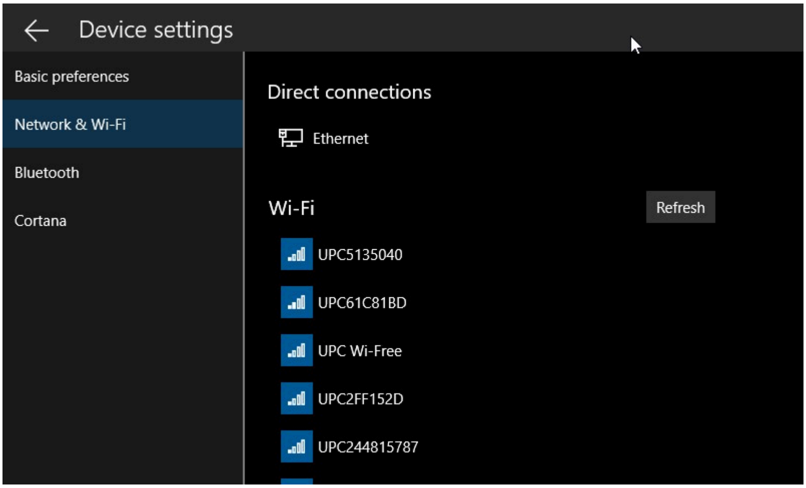
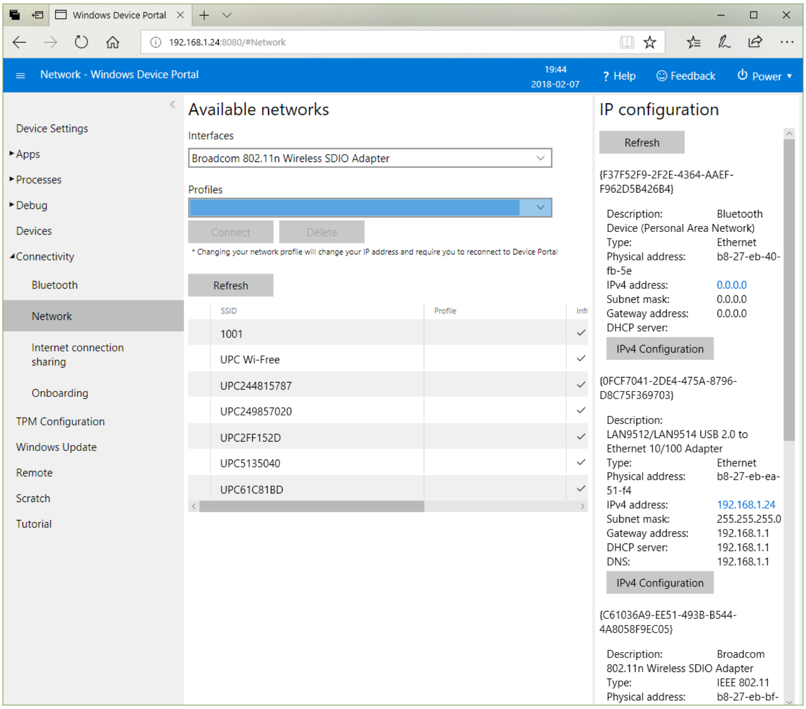


Figure 35: Network & WiFi view



Available networks

Interfaces: Broadcom 802.11n Wireless SDIO Adapter

Profiles

* Changing your network profile will change your IP address and require you to reconnect to Device Portal

SSID	Profile	Infr
1001		✓
UPC Wi-Free		✓
UPC244815787		✓
UPC249857020		✓
UPC2FF152D		✓
UPC5135040		✓
UPC61C81BD		✓

IP configuration

{F37F52F9-2F2E-4364-AAEF-F962D5B426B4}

```
Description:        Bluetooth
Device (Personal Area Network)
Type:               Ethernet
Physical address:   b8-27-eb-40-
fb-5e
IPv4 address:       0.0.0.0
Subnet mask:        0.0.0.0
Gateway address:    0.0.0.0
DHCP server:
```

{0FCF7041-2DE4-475A-8796-D8C75F369703}

```
Description:
LAN9512/LAN9514 USB 2.0 to
Ethernet 10/100 Adapter
Type:               Ethernet
Physical address:   b8-27-eb-ea-
51-f4
IPv4 address:       192.168.1.24
Subnet mask:        255.255.255.0
Gateway address:    192.168.1.1
DHCP server:        192.168.1.1
DNS:                192.168.1.1
```

{C61036A9-EE51-493B-B544-4A8058F9EC05}

```
Description:        Broadcom
802.11n Wireless SDIO Adapter
Type:               IEEE 802.11
Physical address:   b8-27-eb-bf-
```

Figure 36: Network view

Step 7
Security. In a case when an IoT device must be protected from remote hacking, one of

the solutions is to use a Trusted Platform Modules (TPM) module following ISO/IEC 11889 standards for a secure cryptoprocessor, a dedicated microcontroller designed to secure hardware through integrated cryptographic keys. The chip contains physical security mechanisms to protect it from tampering, and malicious software cannot hack the TPM security functions. Some of the TPM key advantages are:

- generate and store the cryptographic keys,
- use the TPM unique RSA key technology for platform device authentication, which is burned into the chip,
- help ensure platform integrity.

The most common TPM functions are used for system integrity measurements, key creation and use. The boot code (including firmware and OS components) is loaded during the system boot process and can be measured and recorded in the TPM module. The integrity measurements are used to show how the OS started and to be sure when the correct boot software was used with the TPM-based key. Windows 10 IoT Core supports a few TPM module standards, which can be connected to the 40-pin GPIO connector (figure 37).

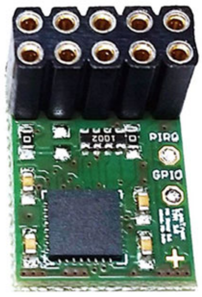
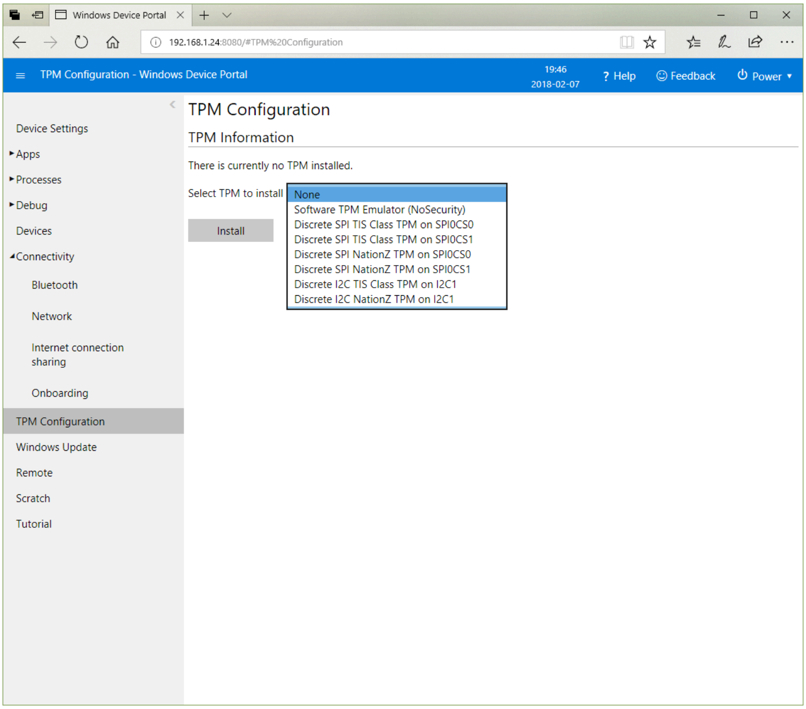


Figure 37: TPM module view

Under the TPM Configuration tab in the Windows Device, the Portal user can select the proper communication protocol for the TPM module installed in the Raspberry Pi board. Then, the appropriate driver for the TPM module can be installed in the Device Manager tab (figure 38).


Windows Device Portal

192.168.1.24:8080/#TPM%20Configuration

TPM Configuration - Windows Device Portal

19:46
2018-02-07

? Help

Feedback

Power

Device Settings
Apps
Processes
Debug
Devices
Connectivity
Bluetooth
Network
Internet connection sharing
Onboarding
TPM Configuration
Windows Update
Remote
Scratch
Tutorial

TPM Configuration

TPM Information

There is currently no TPM installed.

Select TPM to install: None

Install

None
Software TPM Emulator (NoSecurity)
Discrete SPI TIS Class TPM on SPI0CS0
Discrete SPI TIS Class TPM on SPI0CS1
Discrete SPI NationZ TPM on SPI0CS0
Discrete SPI NationZ TPM on SPI0CS1
Discrete I2C TIS Class TPM on I2C1
Discrete I2C NationZ TPM on I2C1

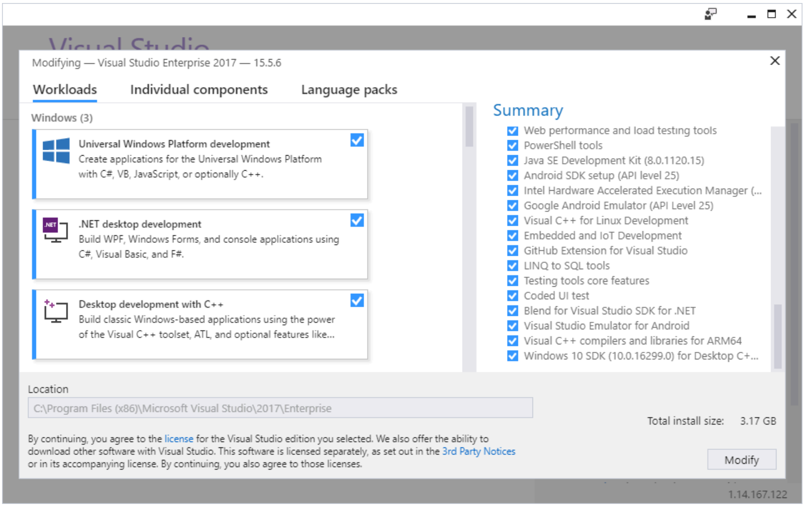
Figure 38: TPM Configuration view

RPI Windows 10 IoT Sample Project

B M E

Create Simple Hello World Application for Raspberry Pi Board

To create a simple Hello Word application under Windows 10 IoT Core, Visual Studio 2022 or newer version is needed. Visual Studio must be installed with the Universal Windows Platform development extension (figure 39).


Modifying — Visual Studio Enterprise 2017 — 15.5.6

Windows (3)

- ✓ Universal Windows Platform development — Create applications for the Universal Windows Platform with C#, VB, JavaScript, or optionally C++.
- ✓ .NET desktop development — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F#.
- ✓ Desktop development with C++ — Build classic Windows-based applications using the power of the Visual C++ toolset, ATL, and optional features like...

Summary

- ✓ Web performance and load testing tools
- ✓ PowerShell tools
- ✓ Java SE Development Kit (8.0.1120.15)
- ✓ Android SDK setup (API level 25)
- ✓ Intel Hardware Accelerated Execution Manager (...
- ✓ Google Android Emulator (API Level 25)
- ✓ Visual C++ for Linux Development
- ✓ Embedded and IoT Development
- ✓ GitHub Extension for Visual Studio
- ✓ LINQ to SQL tools
- ✓ Testing tools core features
- ✓ Coded UI test
- ✓ Blend for Visual Studio SDK for .NET
- ✓ Visual Studio Emulator for Android
- ✓ Visual C++ compilers and libraries for ARM64
- ✓ Windows 10 SDK (10.0.16299.0) for Desktop C+...

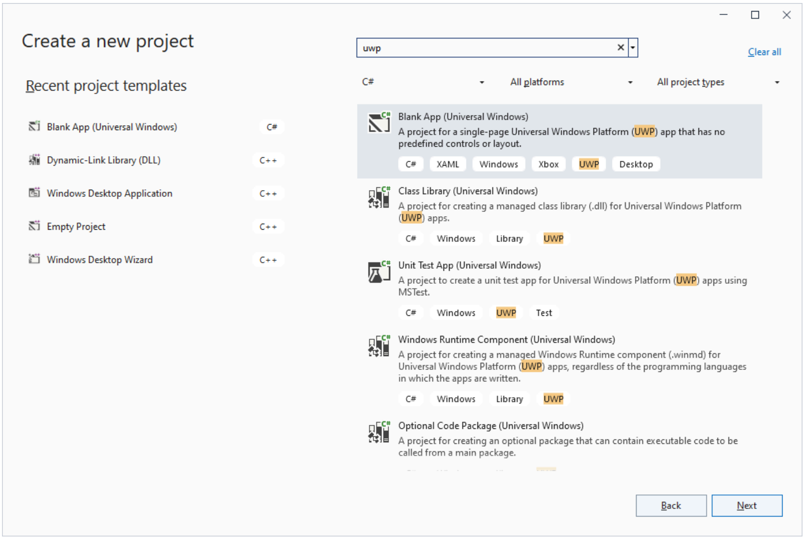
Location
C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise

Total install size: 3.17 GB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Figure 39: Visual Studio UWP development view

Step 1
Create a new UWP project by choosing the Windows Universal/Blank App Project (figure 40).


Create a new project

Recent project templates

Blank App (Universal Windows)	C#
Dynamic-Link Library (DLL)	C++
Windows Desktop Application	C++
Empty Project	C++
Windows Desktop Wizard	C++

uwp — Clear all

C# — All platforms — All project types

Blank App (Universal Windows)
A project for a single-page Universal Windows Platform (UWP) app that has no predefined controls or layout.
C# XAML Windows Xbox UWP Desktop

Class Library (Universal Windows)
A project for creating a managed class library (.dll) for Universal Windows Platform (UWP) apps.
C# Windows Library UWP

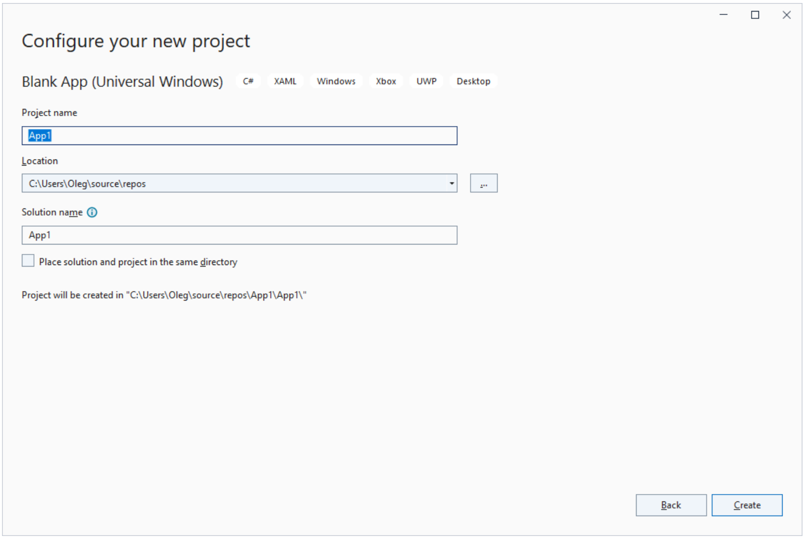
Unit Test App (Universal Windows)
A project to create a unit test app for Universal Windows Platform (UWP) apps using MSTest.
C# Windows UWP Test

Windows Runtime Component (Universal Windows)
A project for creating a managed Windows Runtime component (.winmd) for Universal Windows Platform (UWP) apps, regardless of the programming languages in which the apps are written.
C# Windows Library UWP

Optional Code Package (Universal Windows)
A project for creating an optional package that can contain executable code to be called from a main package.

Figure 40: Visual Studio Create New project view

Step 2
Configure your new project (according to Raspberry Pi Windows 10 IoT Core build version) is presented in figure 41.


Configure your new project

Blank App (Universal Windows)

Project name: App1

Location: C:\Users\Oleg\source\repos

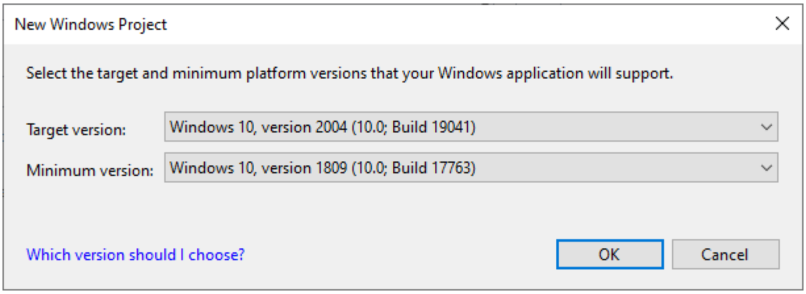
Solution name: App1

Place solution and project in the same directory

Project will be created in "C:\Users\Oleg\source\repos\App1\App1\"

Figure 41: Configure new project

Step 3
Choose the Target version of your device platform which Windows IoT Core will support, as in figure 42.


New Windows Project

Select the target and minimum platform versions that your Windows application will support.

Target version: Windows 10, version 2004 (10.0; Build 19041)

Minimum version: Windows 10, version 1809 (10.0; Build 17763)

Figure 42: Target platform selection tab

Step 4
Create App1 solution (figure 43).

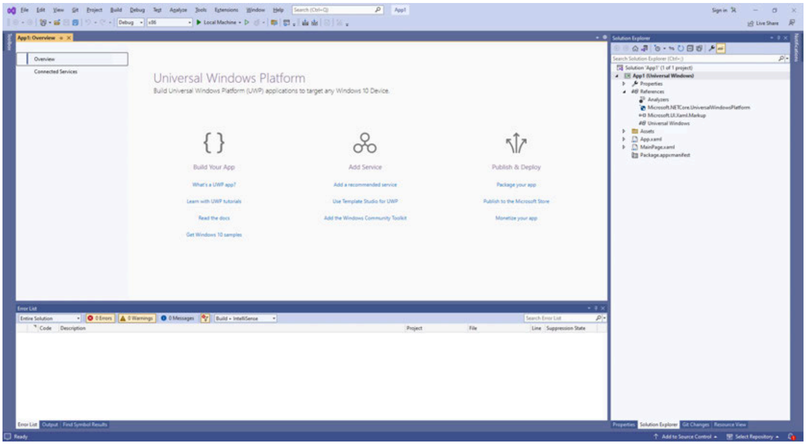
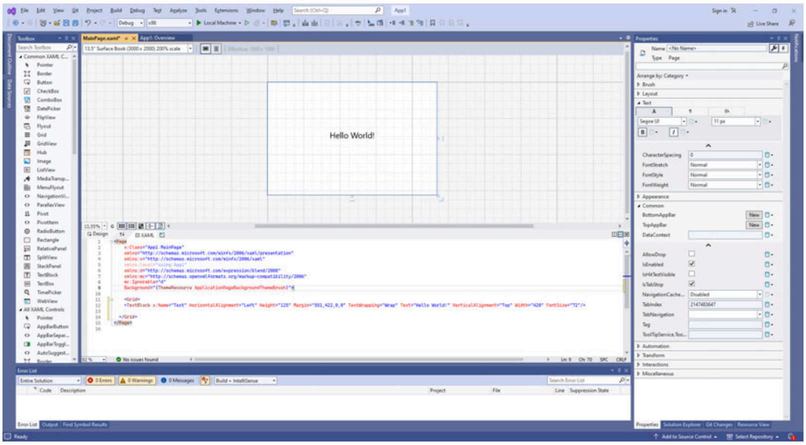

Universal Windows Platform
Build Universal Windows Platform (UWP) applications to target any Windows 10 Device.
Build Your App
Add Service
Publish & Deploy
Solution Explorer
Error List

Figure 43: VS 2022 project environment view

Step 5

Design the application screen by modifying the `MainPage.xaml` file. To add different screen features, use the Toolbox/All XAML controls (figure 44).

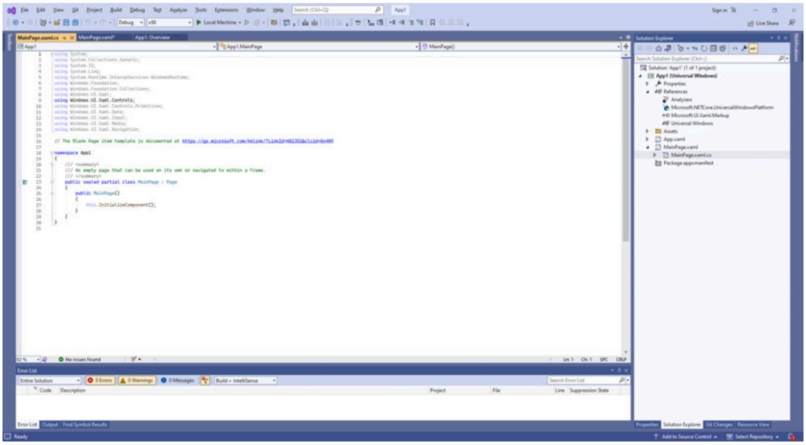


```
<Page
    x:Class="App1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:App1"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    <Grid>
    <TextBlock x:Name="Text" HorizontalAlignment="Left" Height="125" Margin="551,422,0,0" TextWrapping="Wrap" Text="Hello World!" VerticalAlignment="Top" Width="420" FontSize="72"/>

   </Grid>
</Page>
```

Figure 44: Add TextBlock control in the MainPage.xaml

Step 6

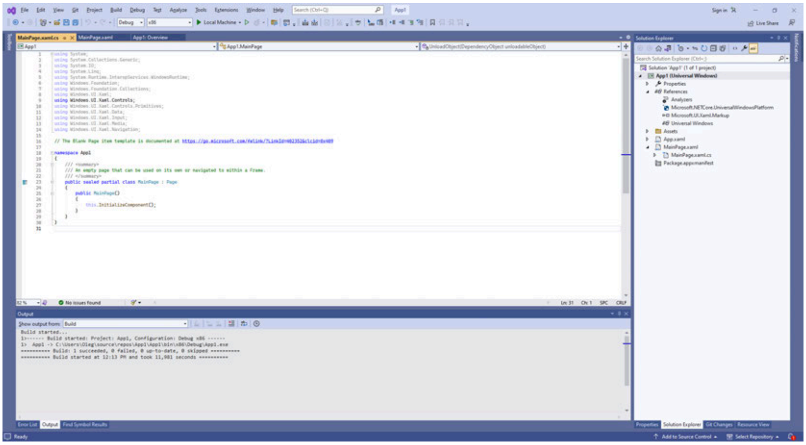
Modify the `MainPage.cs` file content if you need control events programming (figure 45).



```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at https://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

namespace App1
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

Figure 45: C# module design MainPage.xaml.cs

Step 7
Compile and run Hello solution. Choosing the Solution Platform for the x86 user will be able to debug and run the Hello application on the computer's desktop emulator. This step is useful for program touchscreen design but cannot test the sensors and controls programming due to software emulator restrictions (figure 46).

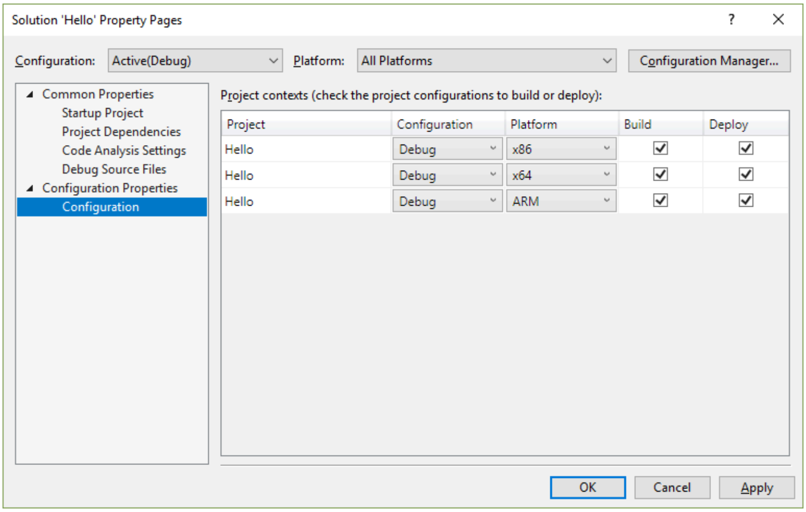


```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at https://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

namespace App1
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

```
Build started...
1>------ Build started: Project: App1, Configuration: Debug x86 ------
1>  App1 -> C:\Users\Oleg\source\repos\App1\App1\bin\x86\Debug\App1.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Build started at 12:13 PM and took 11,981 seconds ==========
```

Figure 46: Solution x86 platform build

Software emulators aren't capable of simulating their behaviour. Instead, the Solution Platform must be changed to the ARM platform in the VC Solution Configuration property to use sensors and controls. The application package must be transferred to the real IoT device to debug. Correct configuration is present in the figure 47.


Solution 'Hello' Property Pages

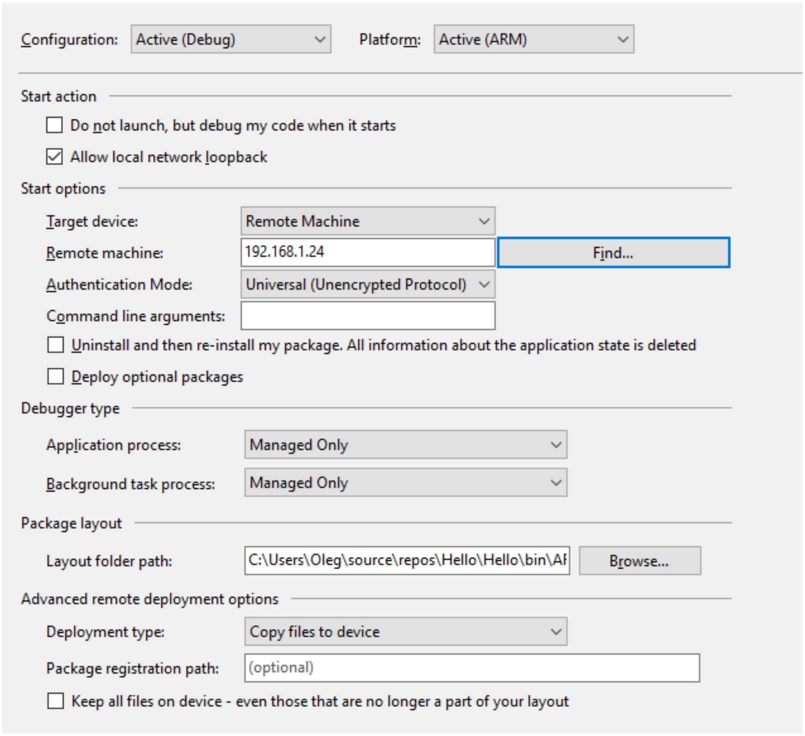
Configuration: Active(Debug) Platform: All Platforms Configuration Manager...

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
Hello	Debug	x86	✓	✓
Hello	Debug	x64	✓	✓
Hello	Debug	ARM	✓	✓

Figure 47: Select a hardware platform for the solution

Step 8
The user must configure the debug application settings to deploy and debug the application package on the real IoT device. In the Debug property page, the user must enter the proper Remote IoT device IP number (figure 48).



Configuration: Active (Debug)   Platform: Active (ARM)

Start action

- [ ] Do not launch, but debug my code when it starts
- [x] Allow local network loopback

Start options

Target device:	Remote Machine
Remote machine:	192.168.1.24
Authentication Mode:	Universal (Unencrypted Protocol)
Command line arguments:	

- [ ] Uninstall and then re-install my package. All information about the application state is deleted
- [ ] Deploy optional packages

Debugger type

Application process:	Managed Only
Background task process:	Managed Only

Package layout

Layout folder path:	C:\Users\Oleg\source\repos\Hello\Hello\bin\AI

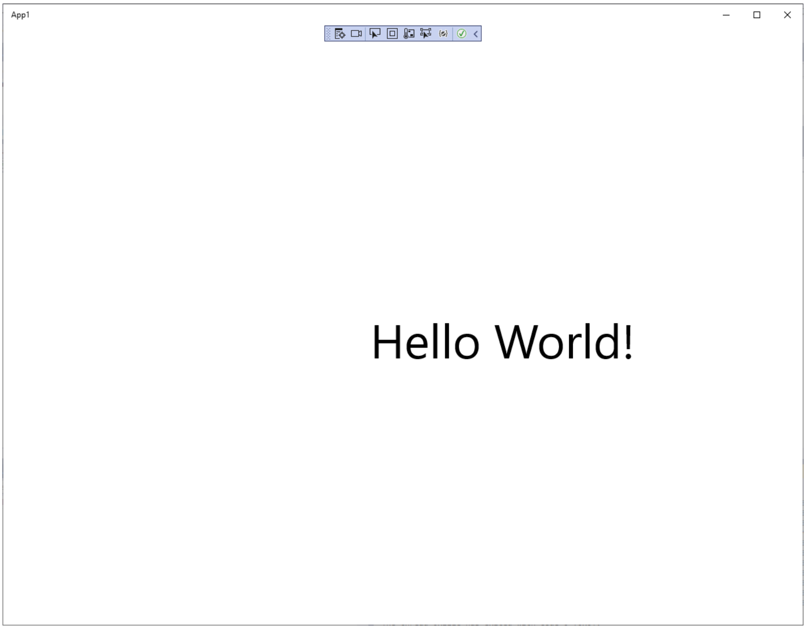
Advanced remote deployment options

Deployment type:	Copy files to device
Package registration path:	(optional)

- [ ] Keep all files on device - even those that are no longer a part of your layout

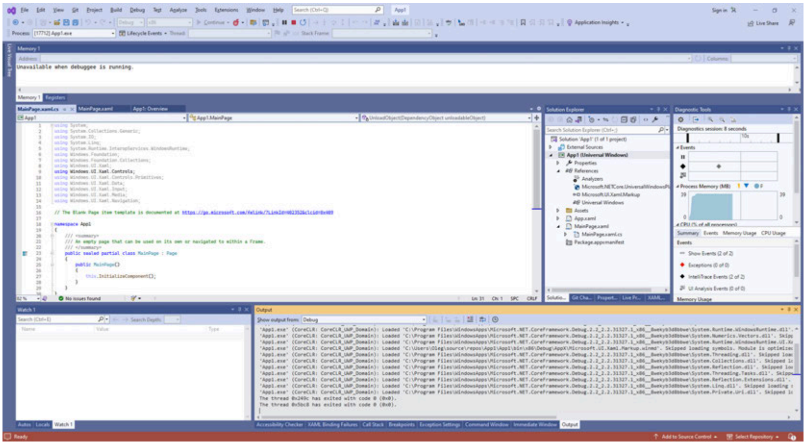
Figure 48: Set the Remote machine IP number

Step 9
Start debugging the application, and after deploying the application package to the board SD card, it will be displayed on the monitor: Hello application is present in figure 49 and debugging view in figure 50.



Hello World!

Figure 49: Hello Application on the Raspberry Pi display



```
Unavailable when debuggee is running.
```

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at https://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

namespace App1
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

```
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Runtime.WindowsRuntime.dll'.
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Numerics.Vectors.dll'. Skipp
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Runtime.WindowsRuntime.UI.Xa
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Users\Oleg\source\repos\App1\App1\bin\x86\Debug\AppX\Microsoft.UI.Xaml.Markup.winmd'. Skipped loading symbols. Module is optimizec
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Threading.dll'. Skipped loac
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Collections.dll'. Skipped lo
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Reflection.dll'. Skipped loa
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Threading.Tasks.dll'. Skipp
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Reflection.Extensions.dll'.
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Linq.dll'. Skipped loading s
'App1.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Program Files\WindowsApps\Microsoft.NET.CoreFramework.Debug.2.2_2.2.31327.1_x86__8wekyb3d8bbwe\System.Private.Uri.dll'. Skipped lo
The thread 0x249c has exited with code 0 (0x0).
The thread 0x5bc8 has exited with code 0 (0x0).
```

Figure 50: Hello Application debugging view

C# Variables and Data Types

The C# [58] variables are categorized into the following types:

- value types,
- reference types,
- pointer types.

Value Type Variables

Value-type variables can assign a value directly. The class `System.ValueType` defines them.

The value types directly contain data. Value types may be: `int`, `char` and `float`, storing numbers, strings or floating point values. When an int type is declared, the system allocates memory to store its value.

The available value types listed in C# are presented as follows (table 5):

Table 5: C# 2010 Value Definitions

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	(–7.9 × 10E28 to 7.9 × 10E28) / 10E0 to 28	0.0M
double	64-bit double-precision floating-point type	(+/–)5.0 × 10E–324 to (+/–)1.7 × 10E308	0.0D
float	32-bit single-precision floating-point type	–3.4 × 10E38 to + 3.4 × 10E38	0.0F
int	32-bit signed integer type	–2 147 483 648 to 2 147 483 647	0
long	64-bit signed integer type	–9 223 372 036 854 775 808 to 9 223 372 036 854 775 807	0L
sbyte	8-bit signed integer type	–128 to 127	0
short	16-bit signed integer type	–32 768 to 32 767	0
uint	32-bit unsigned integer type	0 to 4 294 967 295	0
ulong	64-bit unsigned integer type	0 to 18 446 744 073 709 551 615	0
ushort	16-bit unsigned integer type	0 to 65 535	0

Object Type

The Object Type is an alias for the `System.Object` class. It is the ultimate base class for all data types in the C# Common Type System (CTS). The object types can be assigned with values of any other types, value types, reference types, predefined or user-defined types. Before assigning values, the type conversion is needed.

When a value type is converted to an object type, it is called **boxing**; when it is converted to a value type, it is called **unboxing**.

```
object obj;
obj = 100; //This is boxing
```

Dynamic Type

The data type variable can store any value. But this type of checking takes place at run-time.

The syntax for declaring a dynamic type is:

```
dynamic <variable_name> = value;
```

For example,

```
dynamic d = 20;
```

Dynamic types are similar to object types. That type checking for object type variables takes place at compile time. For the dynamic type variables, checking takes place at run time.

String Type

The string type allows assigning any string values to a variable. The string type is an alias for the `System.String` class derived from the object type. The string type value can be assigned using string literals in two forms: **quoted** and **@quoted**.

For example,

```
string str = "Tutorials Point";
```

A @quoted string literal looks as follows:

```
@"Tutorials Point";
```

The user-defined reference types are `class`, `interface`, or `delegate`.

Reference Type

The reference types don't contain the actual data stored in a variable. They contain a reference to the variables.

Using multiple variables, the reference types can refer to a memory location. If the variable changes the data in the memory location, the other variable automatically reflects this change in value. Built-in reference example types are `object`, `dynamic`, and `string`.

Pointer Type

Pointer-type variables store the memory address, which is another type. Pointers in C# are similar to pointers in C or C++.

The syntax for declaring a pointer type is:

```
type* identifier;
```

For example,

```
char* cptr;
int* iptr;
```

C# Variables

Each variable in C# has a specific type, which determines the size and layout of the variable's memory.

The basic value types in C# can be categorised as follows:

Table 6: C# 2010 Variables

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong, and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

Variable Definitions

Variable syntax definition in C# is:

```
<data_type> <variable_list>;
```

`data_type` must be a valid C# data type like `char`, `int`, `float`, `double`, or any user-defined data type. `variable_list` may consist of one or more identifier names separated by commas.

Examples of valid variable definitions are shown below:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

The variable can be initialized immediately during definition time:

```
int i = 100;
```

Variables Initialization

Variables are initialized with an equal sign followed by a constant expression. The general initialization form looks:

```
variable_name = value;
```

Variables can be initialized during their declaration. The initializer consists of an equal sign followed by a constant expression as:

```
<data_type> <variable_name> = value;
```

Some examples are:

```
int d = 3, f = 5;    /* Initializing d and f */
byte z = 22;         /* Initializes z */
double pi = 3.14159; /* Declares an approximation of PI */
char x = 'x';        /* The variable x has the value 'x' */
```

It is essential to initialize variables properly; otherwise, sometimes, it may produce unexpected results.

The following example uses various types of variables:

```
using System;

namespace VariableDefinition {
 class Program {
   static void Main(string[] args) {
     short a;
     int b ;
     double c;
     /* Actual initialization */
     a = 10;
     b = 20;
     c = a + b;
     Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
     Console.ReadLine();
   }
 }
}
```

Output:

```
a = 10, b = 20, c = 30
```

C# Program Control Structures

C# [59] Specifying one or more conditions to be evaluated or tested by the program requires decision-making structures. The proper statements must be performed if the condition is true or false.

C# provides the following types of decision-making statements (table 7):

Table 7: C# 2010 Loops Definitions

Sr.No.	Loop Type & Description
1	An **if** statement consists of a boolean expression followed by one or more statements
2	**if...else** statement – an **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false
3	Nested **if** statements – you can use one **if** or **else if** statement inside another **if** or **else if** statement(s)
4	**switch** statement – a switch statement allows a variable to be tested for equality against a list of values
5	Nested **switch** statements – you can use one switch statement inside another switch statement(s)
6	The ? Operator

if Statement

An **if** statement consists of a boolean expression followed by one or more statements. The syntax of an **if** statement in C# is:

```
if(boolean_expression) {
 /* Statement(s) will execute if the boolean expression is true */
}
```

If the boolean expression evaluates to true, the code block inside the **if** statement is executed. If the boolean expression evaluates to false, the first code set after the end of the **if** statement (after the closing curly brace) is executed.

Example:

```
using System;

namespace DecisionMaking {
  class Program {
    static void Main(string[] args) {
      /* Local variable definition */
      int a = 10;

      /* Check the boolean condition using if statement */
      if (a < 20) {
        /* If condition is true then print the following */
        Console.WriteLine("a is less than 20");
      }
      Console.WriteLine("value of a is : {0}", a);
      Console.ReadLine();
    }
  }
}
```

Output:

```
a is less than 20;
value of a is : 10
```

if…else Statement
An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false. The syntax of an **if…else** statement in C# is:

```
if(boolean_expression) {
  /* Statement(s) will execute if the boolean expression is true */
} else {
  /* Statement(s) will execute if the boolean expression is false */
}
```

If the boolean expression evaluates to true, then the if block of code is executed; otherwise, **else** block of code is executed.

Example:

```
using System;

namespace DecisionMaking {
  class Program {
    static void Main(string[] args) {
      /* Local variable definition */
      int a = 100;

      /* Check the boolean condition */
      if (a < 20) {
        /* If condition is true then print the following */
        Console.WriteLine("a is less than 20");
      } else {
        /* If condition is false then print the following */
        Console.WriteLine("a is not less than 20");
      }
      Console.WriteLine("value of a is : {0}", a);
      Console.ReadLine();
    }
  }
}
```

Output:

```
a is not less than 20;
value of a is : 100
```

Nested if Statement
It is always legal in C# to nest **if...else** statements, which means you can use one **if** or **else if** statement inside another **if** or **else if** statement(s). The syntax for a nested **if** statement is as follows:

```
if( boolean_expression 1) {
  /* Executes when the boolean expression 1 is true */
  if(boolean_expression 2) {
    /* Executes when the boolean expression 2 is true */
  }
}
```

Example:

```
using System;

namespace DecisionMaking {
  class Program {
    static void Main(string[] args) {
      //* Local variable definition */
      int a = 100;
      int b = 200;

      /* Check the boolean condition */
      if (a == 100) {

        /* If condition is true then check the following */
        if (b == 200) {
          /* If condition is true then print the following */
          Console.WriteLine("Value of a is 100 and b is 200");
        }
      }
      Console.WriteLine("Exact value of a is : {0}", a);
      Console.WriteLine("Exact value of b is : {0}", b);
      Console.ReadLine();
    }
  }
}
```

Output:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

switch Statement
A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable switched on is checked for each switch case.

The syntax for a **switch** statement in C# is as follows:

```
switch(expression) {
  case constant-expression  :
    statement(s);
    break; /* Optional */
  case constant-expression  :
    statement(s);
    break; /* Optional */

  /* You can have any number of case statements */
  default : /* Optional */
  statement(s);
}
```

The following rules apply to a **switch** statement.

1. The expression in a **switch** statement must have an integral or enumerated type or a class type in which the class has a single conversion function to an integral or enumerated type.
2. You can have any number of case statements within a **switch**. Each case is followed by the value to be compared to and a colon.
3. The constant expression for a case must be the same data type as the variable in the **switch**, and it must be a constant or a literal.
4. When the variable switched on is equal to a case, the statements following that case will execute until a break statement is reached.
5. When a break statement is reached, the **switch** terminates, and the control flow jumps to the next line following the **switch** statement.
6. Not every case needs to contain a break. If no break appears, the control flow will fall through to subsequent cases until a break is reached.
7. A **switch** statement can have an optional default case, which must appear at the end of the **switch**. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
using System;

namespace DecisionMaking {
  class Program {
    static void Main(string[] args) {
      /* Local variable definition */
      char grade = 'B';

      switch (grade) {
        case 'A':
          Console.WriteLine("Excellent!");
          break;
        case 'B':
        case 'C':
          Console.WriteLine("Well done");
          break;
        case 'D':
          Console.WriteLine("You passed");
          break;
        case 'F':
          Console.WriteLine("Better try again");
          break;
        default:
          Console.WriteLine("Invalid grade");
          break;
      }
      Console.WriteLine("Your grade is  {0}", grade);
      Console.ReadLine();
    }
  }
}
```

Output:

```
Well done
Your grade is B
```

Nested switch Statement
It is possible to have a **switch** as part of an outer **switch**statement sequence. No conflicts will arise even if the case constants of the inner and outer **switch** contain common values.

The syntax for a nested **switch** statement is as follows:

```
switch(ch1) {
  case 'A':
    Console.WriteLine("This A is part of outer switch" );

  switch(ch2) {
    case 'A':
      Console.WriteLine("This A is part of inner switch" );
      break;
    case 'B': /* Inner B case code */
  }
  break;
  case 'B': /* Outer B case code */
}

Example:
<code C>
using System;

namespace DecisionMaking {
  class Program {
    static void Main(string[] args) {
      int a = 100;
```

```
   int b = 200;

   switch (a) {
    case 100:
     Console.WriteLine("This is part of outer switch ");

     switch (b) {
      case 200:
       Console.WriteLine("This is part of inner switch ");
       break;
     }
    break;
   }
   Console.WriteLine("Exact value of a is : {0}", a);
   Console.WriteLine("Exact value of b is : {0}", b);
   Console.ReadLine();
  }
 }
}
```

Output:

```
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```

C# Looping

C# [60] provides the following types of loops to handle looping requirements, listed in table 8.

Table 8: C# 2010 Loops Definitions

Sr.No.	Loop Type & Description
1	`while` loop – it repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body
2	`for` loop – it executes a sequence of statements multiple times and abbreviates the code that manages the loop variable
3	do…`while` loop – it is similar to a while statement, except that it tests the condition at the end of the loop body
4	Nested loop – you can use one or more loop inside any another `while`, `for` or do…`while` loop

while Loop

A `while` loop statement in C# repeatedly executes a target statement if a given condition is true.

```
while(condition) {
   statement(s);
}
```

Example:

```
using System;
```

```
namespace Loops {
  class Program {
    static void Main(string[] args) {
      /* Local variable definition */
      int a = 10;

      /* while loop execution */
      while (a < 20) {
        Console.WriteLine("value of a: {0}", a);
        a++;
      }
      Console.ReadLine();
    }
  }
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

for Loop
A `for` loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times. The syntax of a for loop in C# is:

```
for ( init; condition; increment ) {
  statement(s);
}
```

Here is the flow of control in a `for` loop:

1. The `init` step is executed first, and only once. This step allows you to declare and initialise any loop control variables. You are not required to put a statement here as long as a semicolon appears.
2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If false, the loop's body does not run, and the control flow jumps to the next statement just after the `for` loop.
3. After the body of the `for` loop executes, the control flow returns to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank if a semicolon appears after the condition.
4. The condition is now re-evaluated. If it is true, the loop executes, and the process repeats itself (body of the loop, then increment step, and then again testing for a condition). After the condition becomes false, the `for` loop terminates.

Example:

```
using System;
namespace Loops {
  class Program {
    static void Main(string[] args) {

      /* for loop execution */
      for (int a = 10; a < 20; a = a + 1) {
        Console.WriteLine("value of a: {0}", a);
      }
      Console.ReadLine();
    }
  }
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

do…while Loop
The syntax of a do…while loop in C# is:

```
do {
  statement(s);
} while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) execute once before the condition is tested.

If the condition is true, the control flow returns to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false. Example:

```
using System;

namespace Loops {
  class Program {
    static void Main(string[] args) {
      /* Local variable definition */
      int a = 10;

      /* do loop execution */
      do {
        Console.WriteLine("value of a: {0}", a);
        a = a + 1;
      }
      while (a < 20);
      Console.ReadLine();
    }
  }
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Nested for Loop
C# allows using one loop inside another loop (loop nesting). The following section shows a few examples to illustrate the concept. The syntax for a nested `for` loop statement in C# is as follows:

```
for ( init; condition; increment ) {
  for ( init; condition; increment ) {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested `while` loop** statement in C# is as follows:

```
while(condition) {
  while(condition) {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested `do…while` loop** statement in C# is as follows:

```
do {
  statement(s);
  do {
    statement(s);
  }
  while( condition );
}
while( condition );
```

A final note on loop nesting is that you can put any loop inside any other type. For example, a for loop can be inside a while loop or vice versa. Example:

```
using System;

namespace Loops {
  class Program {
    static void Main(string[] args) {
    /* local variable definition */
    int i, j;

    for (i = 2; i < 100; i++) {
      for (j = 2; j <= (i / j); j++)
        if ((i % j) ** 0) break; // if factor found, not prime
          if (j > (i / j)) Console.WriteLine("{0} is prime", i);
      }
      Console.ReadLine();
    }
  }
}
```

Output:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop is required, you can make an endless loop by leaving the conditional expression empty.

Example:

```
using System;

namespace Loops {
  class Program {
    static void Main(string[] args) {
      for (; ; ) {
        Console.WriteLine("Hey! I am Trapped");
      }
    }
  }
}
```

When the conditional expression is absent, it is assumed to be true.

C# Object-Oriented Programming

C# object-oriented programming does not differ much from the C++ model. Below there are major C# class components along with samples.

C# Classes

Defining a Class

A C# [61] class definition starts with the keyword class followed by the class name and the class body enclosed by a pair of curly braces. Following is the general form of a class definition:

```
<access specifier> class  class_name {
  //Member variables
  <access specifier> <data type> variable1;
  <access specifier> <data type> variable2;
  ...
  <access specifier> <data type> variableN;
  //Member methods
  <access specifier> <return type> method1(parameter_list) {
    //Method body
  }
  <access specifier> <return type> method2(parameter_list) {
    //Method body
  }
  ...
  <access specifier> <return type> methodN(parameter_list) {
    //Method body
  }
}
```

Note:

- access specifiers specify the access rules for the members and the class itself. If not mentioned, then the default access specifier for a class type is internal. Default access for the members is private;
- data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any;
- to access the class members, you use the dot (.) operator;
- the dot operator links an object's name with a member's name.

Example:

```
using System;

namespace BoxApplication {

  class Box {
    public double length;   //Length of a box
    public double breadth;  //Breadth of a box
    public double height;   //Height of a box
  }
  class Boxtester {
    static void Main(string[] args) {
      Box Box1 = new Box();   //Declare Box1 of type Box
      Box Box2 = new Box();   //Declare Box2 of type Box
      double volume = 0.0;    //Store the volume of a box here

      //Box1 specification
      Box1.height = 5.0;
      Box1.length = 6.0;
      Box1.breadth = 7.0;

      //Box2 specification
      Box2.height = 10.0;
      Box2.length = 12.0;
      Box2.breadth = 13.0;

      //Volume of Box1
      volume = Box1.height * Box1.length * Box1.breadth;
      Console.WriteLine("Volume of Box1 : {0}",  volume);

      //Volume of Box2
      volume = Box2.height * Box2.length * Box2.breadth;
      Console.WriteLine("Volume of Box2 : {0}", volume);
      Console.ReadKey();
    }
  }
}
```

Output:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Member Functions and Encapsulation
A member function of a class is a function that has its definition or its prototype within the class definition similar to any other variable. It operates on an object of the class of which it is a member and has access to all the members of a class for that object.

Member variables are the attributes of an object (from the design perspective), and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Sample:

```
using System;

namespace BoxApplication {
   class Box {
      private double length;   //Length of a box
      private double breadth;  //Breadth of a box
      private double height;   //Height of a box

      public void setLength( double len ) {
         length = len;
      }
      public void setBreadth( double bre ) {
         breadth = bre;
      }
      public void setHeight( double hei ) {
         height = hei;
      }
      public double getVolume() {
         return length * breadth * height;
      }
   }
   class Boxtester {
      static void Main(string[] args) {
         Box Box1 = new Box();   //Declare Box1 of type Box
         Box Box2 = new Box();
         double volume;

         //Declare Box2 of type Box
         //Box1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         //Box2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);

         //Volume of Box1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}" ,volume);

         //Volume of Box2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         Console.ReadKey();
      }
   }
}
```

Output:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

C# Constructors
A class constructor is a special member function of a class that is executed whenever we create new objects of that class.

A constructor has the same name as that of a class and does not have any return type.

Example:

```
using System;

namespace LineApplication {
   class Line {
      private double length;   //Length of a line

      public Line() {
         Console.WriteLine("Object is being created");
      }
      public void setLength( double len ) {
         length = len;
      }
      public double getLength() {
         return length;
      }

      static void Main(string[] args) {
         Line line = new Line();

         //Set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

Output:

```
Object is being created
Length of line : 6
```

A default constructor has no parameter, but you can make one if you need to pass some setup values on the initialisation - such constructors are called parameterised constructors. This technique helps you to assign an initial value to an object at the time of its creation.

Example:

```
using System;

namespace LineApplication {
   class Line {
      private double length;   //Length of a line

      public Line(double len) {  //Parameterized constructor
         Console.WriteLine("Object is being created, length = {0}", len);
         length = len;
      }
      public void setLength( double len ) {
         length = len;
      }
      public double getLength() {
         return length;
      }
      static void Main(string[] args) {
         Line line = new Line(10.0);
         Console.WriteLine("Length of line : {0}", line.getLength());

         //Set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

Output:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

C# Destructors
A destructor is a special member function of a class executed whenever an object of its class goes out of scope. A destructor has the same name as the class with a prefixed tilde (~), and it can neither return a value nor take any parameters. C# (.NET environment) has a built-in memory management system that tracks unused objects and releases memory automatically. Still, in constrained memory systems like RPI, it is sometimes essential to manually notify this mechanism about the possibility of releasing memory once the object is no longer used. Here, destructor helps much. Moreover, the destructor can handle hardware-related issues, e.g. close connection, sending a farewell message to the external device, etc. Destructors cannot be inherited or overloaded.

Example:

```
using System;

namespace LineApplication {
   class Line {
      private double length;   //Length of a line

      public Line() {   //Constructor
         Console.WriteLine("Object is being created");
      }
      ~Line() {   //destructor
         Console.WriteLine("Object is being deleted");
      }
      public void setLength( double len ) {
         length = len;
      }
      public double getLength() {
         return length;
      }
      static void Main(string[] args) {
         Line line = new Line();

         //Set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
      }
   }
}
```

Output:

```
Object is being created
Length of line : 6
Object is being deleted
```

Static Members of a C# Class
We can define class members as static using the static keyword. When we declare a class member as static, no matter how many class objects are created, there is only one copy of the static member.

The keyword static implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance. Static variables can be initialised outside the member function or class definition. You can also initialise static variables inside the class definition.

Example:

```
using System;

namespace StaticVarApplication {
   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }
      public int getNum() {
         return num;
      }
   }
   class StaticTester {
      static void Main(string[] args) {
         StaticVar s1 = new StaticVar();
         StaticVar s2 = new StaticVar();

         s1.count();
         s1.count();
         s1.count();

         s2.count();
         s2.count();
         s2.count();

         Console.WriteLine("Variable num for s1: {0}", s1.getNum());
         Console.WriteLine("Variable num for s2: {0}", s2.getNum());
         Console.ReadKey();
      }
   }
}
```

Output:

```
Variable num for s1: 6
Variable num for s2: 6
```

You can also declare a member function as static. Such functions can access only static variables. The static functions exist even before the object is created. Example:

```
using System;

namespace StaticVarApplication {
   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }
      public static int getNum() {
         return num;
      }
   }
   class StaticTester {
      static void Main(string[] args) {
         StaticVar s = new StaticVar();

         s.count();
         s.count();
         s.count();

         Console.WriteLine("Variable num: {0}", StaticVar.getNum());
         Console.ReadKey();
      }
   }
}
```

Output:

```
Variable num: 3
```

C# Events

Events occur when a user makes actions like a key press, clicks, mouse movements, etc., or some other occurrence such as system-generated notifications. Applications must respond to events if they occur, e.g. handle interrupts. Events are used during inter-process communication.

Using Delegates With Events

The events are declared and raised in a class. They are associated with the event handlers using delegates within the same or some other class. To publish the event, the class containing it must be defined. It is called the **publisher** class. Another class that accepts this event is called the **subscriber** class. Events use the **publisher-subscriber** model.

The object containing a definition of the event and the delegate is named **publisher**. The event-delegate association is also defined in this object. A publisher class object invokes the event and is notified to other objects.

A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

Declaring Events
First, a delegate type for the event must be declared to declare an event inside a class. For example,

```
public delegate string MyDel(string str);
```

Next, the event itself is declared using the event keyword:

```
event MyDel MyEvent;
```

The preceding code defines a delegate named `MyDel` and an event named `MyDel`, which invokes the delegate when it is raised.

Example:

```
using System;

namespace SampleApp {
   public delegate string MyDel(string str);

   class EventProgram {
      event MyDel MyEvent;

      public EventProgram() {
         this.MyEvent += new MyDel(this.WelcomeUser);
      }
      public string WelcomeUser(string username) {
         return "Welcome " + username;
      }
      static void Main(string[] args) {
         EventProgram obj1 = new EventProgram();
         string result = obj1.MyEvent("Tutorials Point");
         Console.WriteLine(result);
      }
   }
}
```

Output:

```
Welcome Tutorials Point
```

4. Embedded Communication

IoT systems and related data flows are typically structured into three primary layers 51, eventually into five 52, which is less popular and mainly used in advanced research [62] [63].

The lowest layer is the Perception (physical, acquisition) Layer, the intermediate is the Network Layer, and the higher is the Application Layer. The function of the Perception layer is to keep in contact with the physical environment. Devices working in this layer are designed as embedded systems with a network module. The modern embedded device includes a microcontroller, sensors, and actuators. External memories and typical microcomputer peripherals are usually built into the microcontroller, so they do not require a special connection. Sensors are elements that convert a value of some physical parameter into an electrical signal, while actuators are elements that control environmental parameters. Sensors and actuators are interfaced with the microcontroller using different connection types, including simple digital or analogue connections or much more complex communication links and protocols. IoT nodes in the Perception layer communicate with higher layers using more complex data transmission methods. The wireless transmission protocols between the Perception layer and other layers are described in communications_and_communicating_sut.

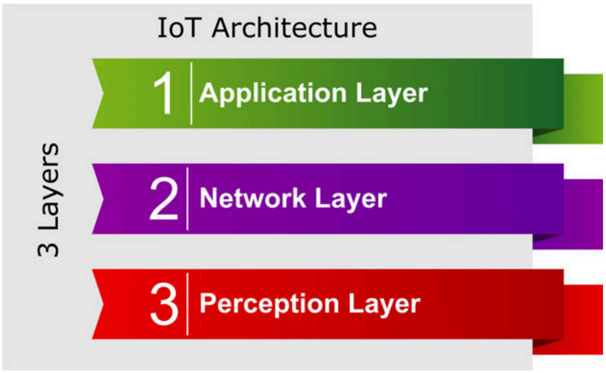


Figure 51: IoT architecture, 3-layered

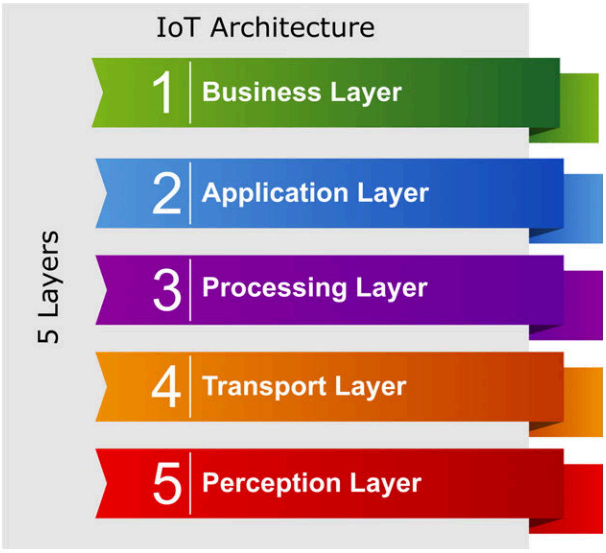


Figure 52: IoT architecture, 5-layered

This chapter describes some popular internal protocols used to communicate between microcontrollers and other electronic elements called “embedded protocols”.

The embedded protocol that can be used in specific implementation depends mainly on the type of peripheral element. The method of connection and data exchange strictly depends on the kind of element. Some parts are analogue sensors that should be connected to an analogue-digital converter; some can be connected to digital pins working as inputs (for sensors) or outputs (for actuators).

Analog

Simple sensors do not implement the conversion and communication logic, and the output is just the analogue signal – voltage level, depending on the value of the measured parameter. It needs to be further converted into a digital representation; this process can be made by the Analogue to Digital Converters (ADC), implemented as the internal part of a microcontroller or separate integrated circuit. Examples of sensors with analogue output are a photoresistor, thermistor, potentiometer, and resistive touchscreen. ADC conversion is a process of conversion of the continuous-time signal into a discrete one. It has 2 crucial parameters to consider:

- Sampling rate: usually measured in Hz (kHz, MHz) is a sampling frequency, or in other words, defines a time period between two consecutive reads. A Nyquist-Shannon theorem defines minimum sampling frequency. Oversampling (using higher than Nyquist-Shannon) is common because many ADC converters built into the MCUs tend to be noisy due to the electromagnetic inference of other components, such as e.g. built-in radio. Oversampling brings the capability to average consecutive reads and obtain more reliable and less noisy ADC conversion.
- Sampling resolution: measured in bits, defines the minimum change in the input voltage the device can measure, e.g. 12-bit resolution brings 4096 values mapped

to the input range. The ideal ADC converter linearly maps the discrete values to the voltage input range. Still, in real-life applications, input characteristics of the ADC used to be non-linear, and software correction may be required once input characteristics are evaluated.

It is worth noting that each ADC has its useable input range (voltage), and the input and analogue signal should be altered accordingly. In real applications, input signal adaptation requires external electronics; thus, many ADC converters provide the ability to amplify the input signal, and it can be programmed.

Digital

Simple, true/false information can be processed via digital I/O. Most devices use positive logic, where, e.g. +5 V (TTL) or +3.3 V (the most popular, yet other voltage standards exist) presents a logical one, also referenced as *HIGH*. In contrast, 0V gives a logical zero, referenced as *LOW*. In real systems, this bounding is fuzzy. It brings some tolerance, simplifying, e.g. communication from 3.3 V output to 5 V input, without a need for the conversion (note, the reverse conversion is usually not so straightforward, as 3.3 V inputs driven by the 5V output may burn quickly). A sample sensor providing binary data is a button (On/Off).
Alternating *HIGH* and *LOW* constitutes a square wave signal, usually used as a clock signal (when symmetrical) or used to control the power delivered to the external devices with means of so-called PWM.

Communication Protocols

Elements that need more data to be transferred (e.g. displays) usually use some digital data transmission protocol. It is often a serial protocol, meaning that data is transmitted bit by bit. Serial communication can be done in three modes.

- In simplex mode, only one of the two devices on a link can transmit; the other can only receive. The simplex mode can use the entire capacity of the channel to send data.
- In half-duplex mode, each station can transmit and receive, but not simultaneously. When one device sends, the other can only receive, and vice versa.
- In full-duplex mode, both stations can transmit and receive simultaneously. The link must contain two physically separate transmission paths, one for sending and the other for receiving.

Serial data transmission can be done synchronously or asynchronously. In synchronous data transmission, bits are synchronized with a clock signal common to the transmitter and receiver. Examples of synchronous protocols are TWI (Two Wire Interface) and SPI (Serial Peripheral Interface). Asynchronous data transmission does not need any separate synchronization signal, but the transmitter and receiver must use the exact timings, and synchronization information must be included in the information transmitted. Examples of asynchronous interfaces implemented in microcontrollers are 1-Wire and UART (Universal Asynchronous Receiver Transmitter).

4.1. PWM

The PWM signal controls the energy delivered to the device, usually a DC motor, LED light, bulb, etc. To control voltage, instead of using inefficient resistance-based voltage dividers (where the remaining part of the voltage is distracted as heat), PWM is based on approximating the energy delivered to the device with periodical switching on and off (HIGH and LOW). Only two voltages are delivered to the device: low (0V) and HIGH (Vcc, e.g. +5V). One can easily observe how PWM works, e.g. when dimming the LED, if recorded with a high fps camera: the LED light flashes with the PWM signal frequency.
PWM controls, in fact, the ratio between HIGH and LOW signals in one period: the higher the ratio, the more energy is being delivered to the device. It is called a duty cycle. A perfect square wave signal, usually referenced as a clock signal, has a duty cycle of 50% (or 0.5); thus, its energy is half of the energy that can be carried when the signal is HIGH all the time. An LED light with a duty cycle of 100% will be fully bright, and with a duty cycle of 0 will be off.

A 50% duty cycle does not necessarily transfer straightforwardly to 50% of brightness or 50% of maximum rpm of the DC motor rotation, as characteristics of the devices regarding the voltage and energy provided to their input may be non-linear.

Some devices are fragile to the changes and cannot accept instant on and off. For this reason, we can use a capacitor that acts as an intermediate energy accumulator and thus flattens the characteristics to be more linear.

PWM signal is then characterised by the following:

- voltage (values when HIGH and LOW),
- frequency,
- duty cycle.

Generating PWM

In microcontrollers, PWM used to be generated with timers and interrupts to ensure asynchronous operation and stability of the operation. Due to the digital nature of the signal generation, a duty cycle generation precision is given by the PWM timer resolution. A n 8-bit resolution splits a period into 256 chunks, and a single chunk defines the minimum time one can increment or decrement the duty cycle. Modern MCUs provide developers with much higher resolution, even up to 14-bit.

A frequency of 5kHz is equivalent to 0.2ms period that can be controlled in steps of

0.2/256 `ms ~= 781 ns.`
Sample visualisation of the 5kHz PWM signal (3.3V) is presented in the following figures, with a duty cycle of, respectively:

- 50/256→~39us (19.5%) in image 53,
- 100/256→~78us (28%) in image 54,
- 150/256→~117us (58.6%) in image 55,
- 200/256→~156us (78.1%) in image 56,
- 250/255→~195us (98%) in image 57.

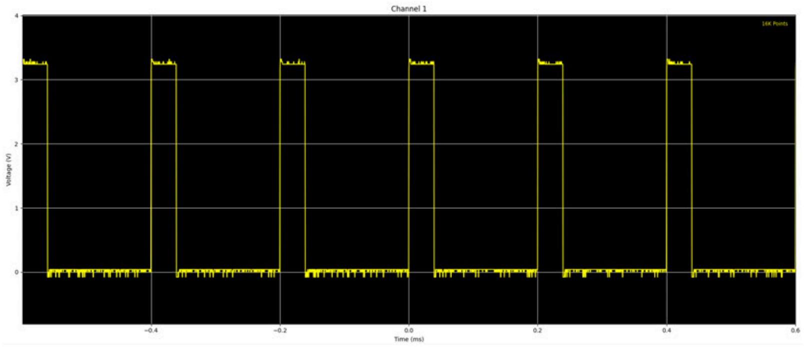


Figure 53: Visualisation of the 5kHz PWM signal with a duty cycle of 19.5%

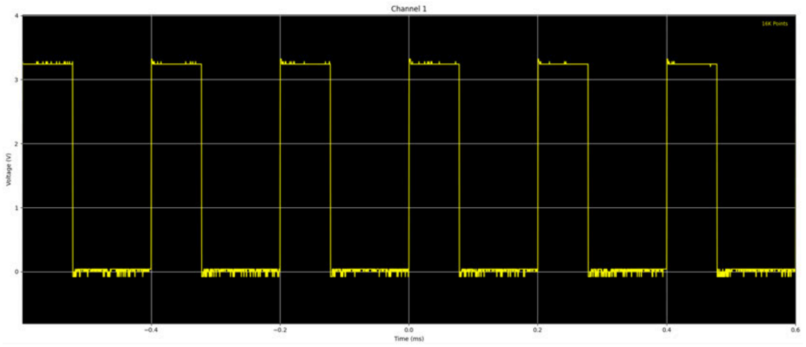


Figure 54: Visualisation of the 5kHz PWM signal with a duty cycle of 28%

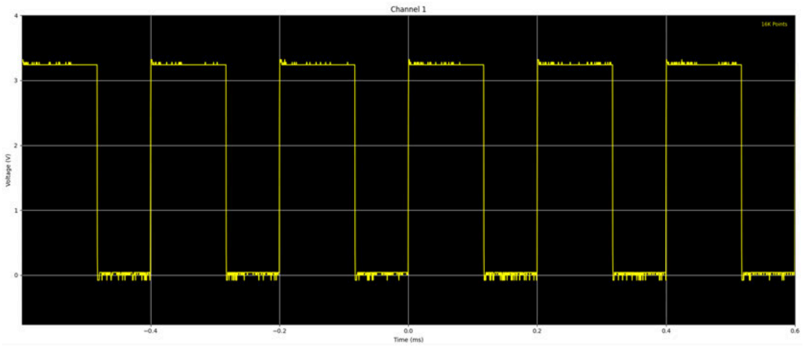


Figure 55: Visualisation of the 5kHz PWM signal with a duty cycle of 58.6%

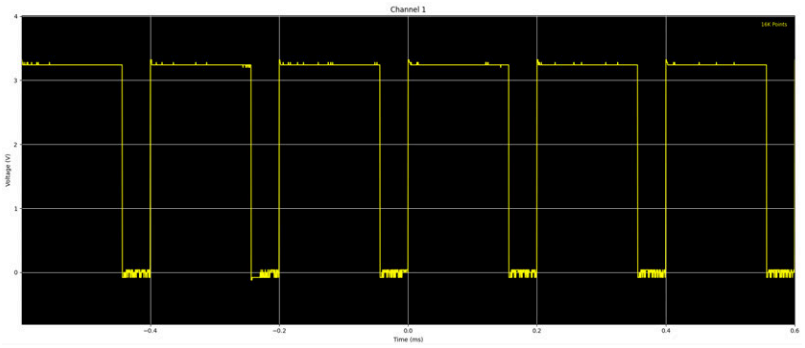


Figure 56: Visualisation of the 5kHz PWM signal with a duty cycle of 78.1%

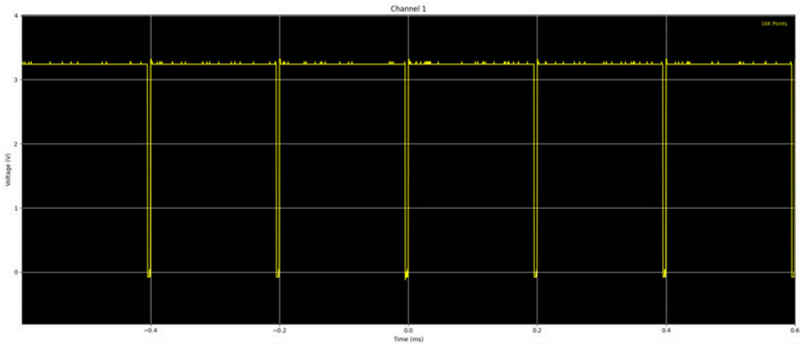


Figure 57: Visualisation of the 5kHz PWM signal with a duty cycle of 98%

Because of the limited hardware resources, an increase in PWM generation resolution or PWM signal frequency may cause an inability to generate a signal or instability of the PWM generation process. Always refer to the MCU's hardware specification for details on the PWM signal limits.

A voltage delivered to the device powered with a PWM signal can be calculated as an integral of the PWM signal over time: e.g., a 50% duty cycle of the 5V signal is equivalent to the delivery of the constant 2.5V.

4.2. SPI

One of the most popular interfaces to connect different external devices is SPI (Serial Peripheral Interface). It is a synchronous serial interface and protocol that can transmit data with speeds up to 20 Mbps. SPI is used to communicate microcontrollers with one or more peripheral devices over short distances – usually internally in the device. High transmission speed enables SPI to be suitable even for sending animated video data to colourful displays. In SPI connection, there is always one master device, in most cases the microcontroller (µC) that controls the transmission, and one or more slave devices – peripherals. To communicate, SPI uses three lines common to all connected devices and one enabling line for every slave element (table 9).

Table 9: SPI Lines

Line	Description	Direction
MISO	Master In Slave Out	peripheral → µC
MOSI	Master Out Slave In	µC → peripheral
SCK	Serial Clock	µC → peripheral
SS	Slave Select	µC → peripheral

The MISO line is intended to send bits from slave to master, the MOSI wire transmits data from master to slave, and the SCK line sends clock pulses that synchronise data transmission. The master device always generates the clock signal. Every SPI-compatible device has the SS (Slave Select) input that enables communication in this specific device. Master is responsible for generating this enable signal – separately for every slave in the system, as present in figure 58.

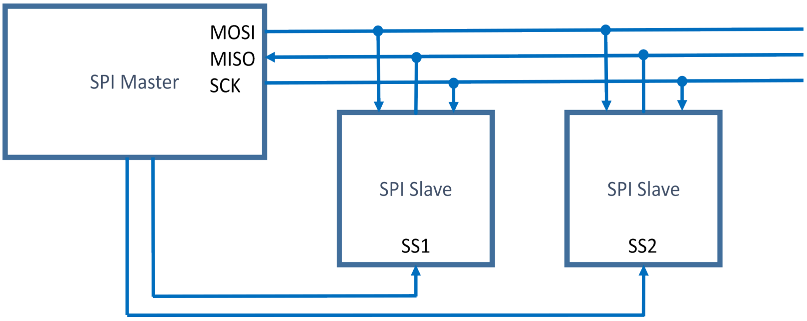


Figure 58: Sample SPI connection

SPI is used in many electronic elements:

- analogue to digital converters (ADC),
- real-time clocks (RTC),
- EEPROMs,
- LCD, OLED, and e-paper displays,
- communication interfaces (e.g. Ethernet, WiFi),
- and many others.

Due to different hardware implementations, there are four SPI protocol operation modes (table 10). The mode used in the master must fit the mode implemented in the slave device.

Table 10: SPI Modes

Mode	Clock polarity	Clock phase	Idle state	Active state	Output edge	Data capture
mode 0	0	0	0	1	falling	rising
mode 1	0	1	0	1	rising	falling
mode 2	1	0	1	0	rising	falling
mode 3	1	1	1	0	falling	rising

It results in different timings of the clock signal concerning the data sent (figure 59). Clock polarity = 0 means that the idle state of the SCK is 0, so every data bit is synchronised with the pulse of logic 1. Clock polarity = 1 reverses these states. Output edge (rising/ falling) says at which edge of active SCK signal sender puts a bit on the data line. The data capture edge says at what edge of SCK signal data should be captured by the receiver.

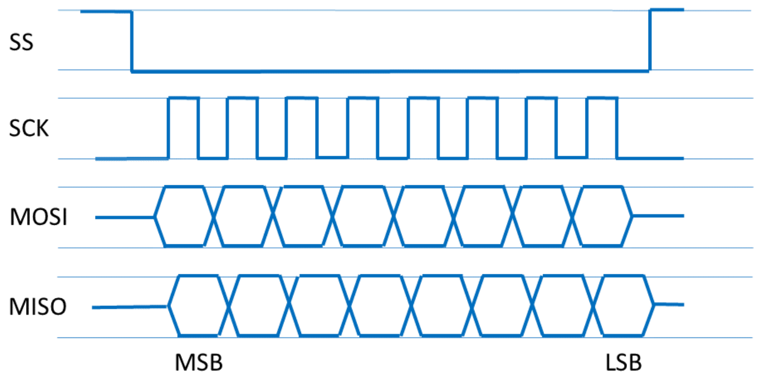


Figure 59: Sample SPI timing

4.2.1. QSPI

Even if a 20MHz frequency ensures good transmission speed, it can be too slow for some use. Some modern microcontrollers use external flash memory for program storage and execute programs from internal SRAM memory, downloading executable code in chunks as required. This requires a higher data rate to avoid stalls in program execution. A QSPI (Quad-SPI) link was developed to achieve higher transmission speed. It has four bidirectional data lines instead of two unidirectional to increase speed four times (figure 60). Additionally, it supports higher clock frequency, increasing speed even higher, currently more than 100MBps. Operation of QSPI requires a special protocol with a set of commands, so hardware implementation is much more complex than the original SPI.

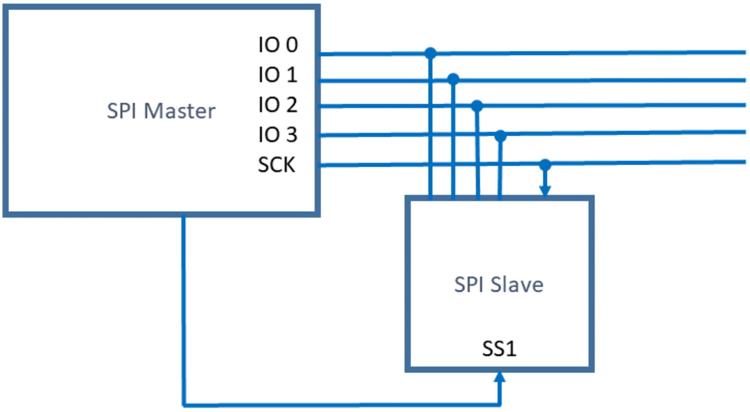


Figure 60: Sample QSPI connection

4.3. TWI (I2C)

TWI (Two Wire Interface) is one of embedded systems' most popular communication links and protocols. Philips has designed it as an I2C (Inter-Integrated Circuit) for audio-video appliances controlled by the microprocessor. Many chips can be connected to the processor with this interface, including:

- EEPROM memory chips,
- RAM memory chips,
- AD/DA converters,
- real-time clocks,
- sensors (temperature, pressure, gas, air pollution),
- port extenders,
- displays,
- specialised AV circuits.

TWI, as the name says, uses two wires for communication. One is the data line (SDA); the second is the clock line (SCL). Both lines are common to all circuits connected to the one TWI bus. The method of communication of TWI is the master-slave synchronous serial transmission. It means that data is sent bit after bit synchronised with the clock signal. The SCL line is always controlled by the master unit (usually the microcontroller); the signal on the SDA line is generated by the master or one of the slaves – depending on the direction of communication. Sample connection is present in figure 61. The frequency rate of the transmission is up to 100 kHz for most of the chips; for some, it can be higher – up to 400 kHz. The new implementation allows an even higher frequency rate, reaching 5 MHz. At the output side of units, the lines have the open-collector or open-drain circuit. This means that external pullup resistors are needed to ensure the proper operation of the TWI bus. The value of these resistors depends on the number of connected elements, the speed of transmission, and the power supply voltage. It can be calculated with the formulas presented, e.g. in the Texas Instrument Application Report [64]. Usually, it is assumed between 1 kΩ and 4.7 kΩ.

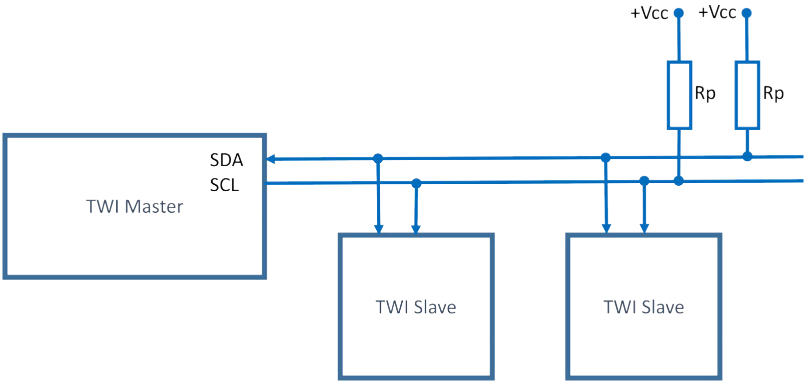


Figure 61: Sample TWI connection

The data is sent using frames of bytes. Every frame begins with a sequence of signals called the start condition. Slaves detect this sequence, which causes them to collect the next eight bits that form the address byte – unique for every circuit on the bus. If one of the slaves recognises its address remains active until the end of the communication frame, others become inactive. To inform the master that some unit has been appropriately addressed slave responses with the acknowledge bit – it generates one bit of low level on the SDA line (the master generates clock pulse). After sending the proper address, data bytes are sent. The direction of the data bytes is controlled by the last bit of the address; for 0, data is transmitted by the master (Write), and for 1, data is sent by the slave (Read). The receiving unit must acknowledge every full byte (eight bits). There is no limitation on the number of data bytes in the frame; for example, samples from the AD converter can be read continuously byte after byte. At the end of the frame, another special sequence is sent by the master–stop condition. It is also possible to generate another start condition without the stop condition. It is called a repeated start condition. Sample TWI fame is present in figure 62.

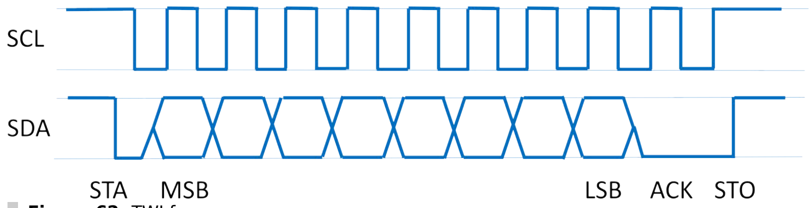


Figure 62: TWI frame

Address byte only activates one chip on the bus, so every unit must have a unique physical address. This byte usually consists of three elements: a 4-bit field fixed by the producer. This 3-bit field can be set by connecting three pins of the chip to 0 (ground) or 1 (power supply line), a 1-bit field for setting the direction of communication (R/#W). Some elements (e.g. EEPROM memory chips) use the 3-bit field for internal addressing, so only one such circuit can be connected to one bus. There are no special rules for the data bytes. The first data byte sent by the master can be used to configure the slave chip. In memory units, it is used for setting the internal address of the memory for writing or reading in multi-channel AD converters to choose the analogue input. Detailed information on the meaning of every bit of the transmission is present in the documentation of the specific integrated circuit. The I2C standard also defines the multi-master mode, but in most small projects, there is one master device only.

4.4. 1-Wire

1-Wire is a master-slave communication asynchronous bus interface designed formerly by Dallas Semiconductor Corp[65]. It can transmit data at long distances at the cost of transmission speed. The typical data speed of the 1-Wire interface is about 16.3 kbit/s, and the maximum length is approx. 300m. Name 1-Wire comes from the feature that the data line can directly power elements connected to the bus. A network chain of 1-Wire devices consists of one master device and many slave devices (figure 63). Such a chain is called a MicroLAN. 1-Wire devices may be a part of a product's circuit board, a single component device such as a temperature probe, or a remote device for monitoring purposes.

Each 1-Wire device must contain a logic unit to operate on the bus. A dedicated bus converter is needed to connect a 1-wire bus to a PC. The most popular PC/1-Wire converters use a USB plug to connect to the PC and the RJ11 connectors (telephones 6P2C/6P4C modular plugs) for MicroLAN. 1-Wire devices can also be connected directly to the microcontroller boards.

Protocol Description

Within the MicroLAN, there is always one master device, typically a PC or a microcontroller unit. The master always initiates activity on the bus to avoid collisions on the network chain. If a collision occurs, the master device retries the communication. In the 1-Wire network, many devices can share the same bus line. To identify devices in the MicroLAN, each connected device has a unique 64-bit ID number. The ID number's least significant byte defines the type of the device (temperature, voltage, etc.). The most significant byte represents a standard 8-bit CRC. The 1-Wire protocol description contains several broadcast commands and commands used to address the selected device. The master sends a selection command, then the address of the selected slave device. This way, the following command is executed only by the addressed device. The 1-Wire bus implements an enumeration procedure that allows the master to get information about the ID numbers of all connected slave devices to the MicroLAN network. The device address includes the device type, identifying what type of slaves are connected to the network chain. The 64-bit address space is searched as a binary tree. It makes it possible to find up to 75 devices per second.

The physical implementation of the 1-Wire network is based on an open drain master device connected to one or more open drain slaves. One single pull-up resistor for all devices pulls the bus up to 3/5 V and can be used to power the slave devices. 1-Wire communication starts when a master or slave sets the bus to low voltage (connects the pull-up resistor to ground through its output MOSFET).

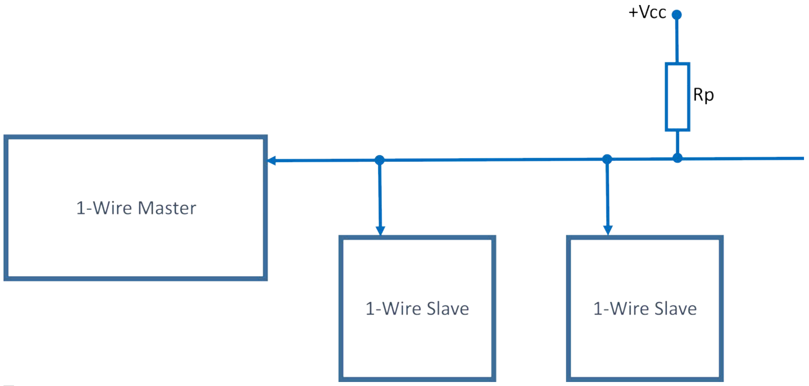


Figure 63: 1-Wire bus connection

The 1-Wire protocol allows for bursting the communication speed up by 10 factors. In this case, the master starts a transmission with a reset pulse, pulling down the data line to 0 volts for at least 480 µs. It resets all slave devices in the network chain bus. Then, any slave device shows it exists, generating the “presence” pulse. It holds the data line low for at least 60 µs after the master releases the bus. To send a “1”, the bus master sends a 1–15 µs low pulse. To send a “0”, the master sends a 60 µs low pulse. The negative edge of the pulse is used to start a slave's monostable multivibrator. The slave's multivibrator clocks to read the data bus about 30 µs after the falling edge. The slave's multivibrator has analogue tolerances that affect its timing accuracy, for the “0” pulses are 60 µs long, and “1” pulses are limited to a max of 15 µs. When the designed solution doesn't contain a dedicated 1-Wire interface peripheral, a UART can be used as a 1-Wire master. Dallas also offers Serial or USB “bridge” chips, which are very useful when the distance between devices is long (greater than 100 m). For longer, up to 300 m buses, the simple twisted pair telephone cable can be used. It will require adjustment of pull-up resistances from 5 kΩ to 1 kΩ. The basic sequence is a reset pulse followed by an 8-bit command, and after it, data can be sent/received in groups of 8-bits. In the case of transmission errors, the weak data protection 8-bit CRC checking procedure can be used.

To find the devices, the enumeration broadcast command must be sent by a master. The slave device responds with all ID bits to the master, and at the end, it returns a 0.

Sample 1-Wire timings are present in figures 64, 65 and 66.

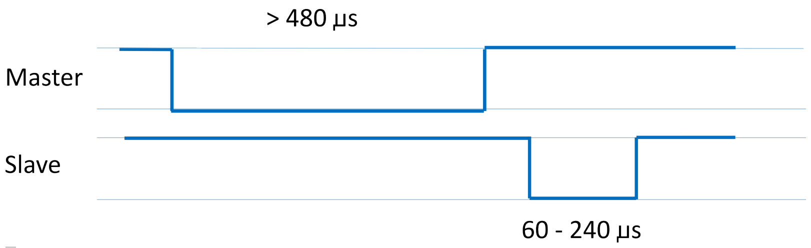


Figure 64: 1-Wire reset timings

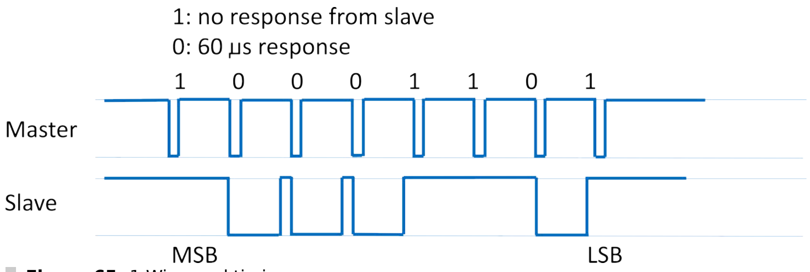


Figure 65: 1-Wire read timings

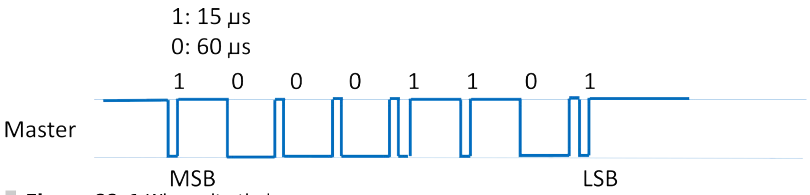


Figure 66: 1-Wire write timings

1-Wire Products

The Dallas/Maxim integrated 1-Wire devices list contains many implementations. The 1-Wire protocol can be quickly implemented into the current IoT boards; most manufacturers share the software libraries, allowing developers to include them in their projects in C, C++, and assembly languages. The 1-Wire sensors (temperature, humidity, pressure, etc.) are factory-calibrated and read the physical measurements following the International System of Units (SI). 1-Wire products can be grouped as follows:

- secure authenticators,
- memory EPROM, EEPROM ROM,
- temperature sensors and temperature switches,
- data loggers,
- 1-Wire interface products,
- battery monitors, protectors, and selectors,
- battery ID and authentication,
- timekeeping and real-time clocks.

4.5. UART

UART name is an abbreviation of Universal Asynchronous Receiver Transmitter. It is one of the most often used communication methods, traditionally named serial interface or serial port. In contrast to previously presented interfaces, UART uses direct point-to-point communication. UART is the communication unit implemented in microcontrollers rather than the communication protocol. It sends the series of bits via the TxD pin and receives a stream of bits with the RxD pin (figure 67). It is important to remember that pin TxD from one device should be connected to pin RxD in another device. This is a general rule, but please always check the documentation for some non-standard markings.

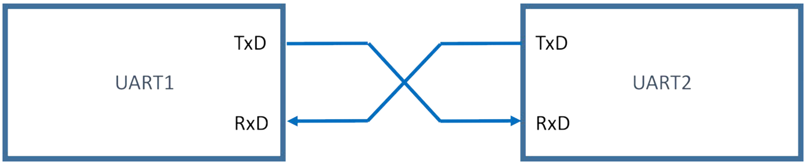


Figure 67: UART connection

The transmission speed and bit duration must be the same at the transmitter and receiver to properly transmit data. Although the transmission speed can be freely chosen, some standard, commonly used baud rates exist. They differ from 300 to 115200 bits per second. Higher baud rates are also available in modern microcontrollers, like 230400, 250000, 500000, 1M, 2M or 3Mbps. In UART, data is sent in frames. The frame begins with the start bit of value “zero”. Next, from five to eight data bits are transmitted. Next, an optional parity bit can appear. The frame is finished with the stop bit of value “one”. Stop bit can be prolonged to 1.5 or 2 times the standard bit duration. After at least one stop bit, the next frame can be sent, beginning with a start bit. Start and stop bits are used to synchronise the receiver and transmitter. Sample transmission flow is present in figure 68.

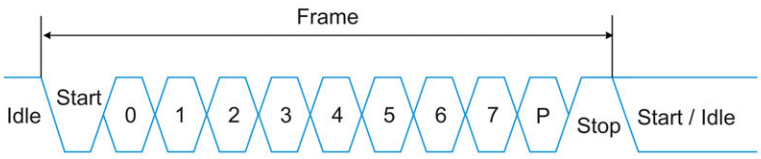


Figure 68: UART frame

UART, namely Serial Port, is used in many modern microcontrollers to upload the executable program, debug, and as the standard input/output for the user interface. For example, in Arduino, functions that operate on the serial port are included in a common set of built-in functions.

Many modern PC computers (except industrial ones) do not have a serial port exposed, so USB to serial converters must be used. Some development boards have a USB-serial converter on board (e.g. Arduino Uno, NodeMCU, STM Nucleo, etc.)

Even if a PC computer has a serial port, it is usually compatible with the RS-232 standard. It uses the same frame structure but different voltage levels (with opposite zero-one encoding, known as reverse logic).

5. IoT Hardware Overview

IoT hardware infrastructure is mainly inherited from the embedded systems of the SoC type for Edge class IoT devices and from PCs for Fog class. As IoT devices are by their nature network-enabled, many of the existing embedded platforms evolved towards network-enabled solutions, sometimes indirectly through delivering network communication module (wired or wireless) as an external device yet integrated on the development board (e.g. Arduino Uno with Ethernet Networking shield, GSM shield, etc.), sometimes a new system, integrating networking capabilities in one SoC (e.g. Espressif SoCs). More advanced devices that require OS to operate preliminarily benefited from externally connected peripheral network interfaces via standard wired ports like USB (e.g. early versions of the Raspberry Pi, where WiFi card was delivered as USB stick), currently, usually integrate most of the network interfaces in a single board (e.g. RPi 4, including Ethernet, WiFi and Bluetooth). Still, in the case of the Fog class devices, those are separate chips from the CPU, and they communicate over, e.g., PCI or USB protocol.

A microcontroller with network capabilities is the key, but it is not the only element forming an IoT node device. Additional elements, including sensors and actuators, are needed to stay in touch with the environment. It is important to emphasise that only hardware components carrying CE marking are appropriate in Europe. This is important in providing human safety for HVAC, DC modules and other electrical devices, while EMC/ETSI regulatory compliance applies to radio devices.

5.1. Most Noticeable Platforms

The IoT market is an emerging one. New hardware solutions appear almost daily, while others disappear quickly. At the moment of writing the first version of this book (2016-2019), some hardware solutions that seemed prominent for at least a couple of years existed. After a few years, while the 2nd edition of the publication is being prepared (2023–2025), most of the hardware solutions described previously are still present on the market, even strengthening their position and having modernized and improved versions (e.g. ESP32 as the successor of ESP8266). However, some other platforms increased their popularity, mainly because of their appearance in the VSCode programming environment with PlatformIO, and what is even more important, the possibility of writing programs in the Arduino model. In the following sections, a short review of these platforms is provided.

- AVR: Arduino – a development board using the Atmel microcontroller, undoubtedly the most popular development platform for enthusiasts and professionals. Arduino itself barely offers networking capabilities yet; there are many extension boards, including wired and wireless network interfaces.
- ESP: Espressif (Espressif Systems) – the great SoC solutions with wireless network interfaces built-in; the family of Espressif chips includes ESP8266 (WiFi) and ESP32 (802.11: WiFi, Bluetooth and 802.15.4: Matter, BLE, Thread and Zigbee).
- nRF52: Nordic Semiconductor SoC based on ARM architecture offers 802.15.4 protocols: Bluetooth, ZigBee, Matter, and Thread.
- STM32: Another ARM-based family of SoCs; some have Bluetooth wireless module built-in and 802.15.4 protocols.
- ARM: Raspberry Pi (and its clones) – advanced boards, including Linux operating system with GUI interface, even able to replace desktop computers. There are also, however, low-powered, constrained devices with ARM cores, such as Cortex-M0+ (Raspberry Pi Pico/Pico W: RP2040).

5.1.1. Arduino General Overview

No doubt, Arduino became the most widespread name in the development boards world, particularly among enthusiasts, educators, amateurs, and hobbyists, driving de-facto the embedded systems market for years.

Using cheap Atmel AVR microcontrollers, delivered along with development board and peripherals of almost any kind, including sensors and actuators, where you do not need to develop your PCB nor solder to obtain the fully functional device, all that triggered a new era where almost anyone can afford to have a development set and start playing the way only professionals used to do. Moreover, Arduino was not only the hardware but also the programming idea, delivering a simple development environment that is easy to use for beginners. Perhaps the most crucial impact of the Arduino on daily use was to spread the idea of taking automation control from the industry and bringing it on a massive scale to regular life, homes, cars, and toys to automate daily life.

The beginnings of the Arduino are dated to the year 2003 in Italy. Their most popular development board was delivered to the market in the fall of 2010. While AVRs microcontrollers are considered to be embedded systems more than IoT, and most of the early Arduino boards didn't offer any network interface, even then, it is essential to understand the idea of how to work with SoCs, so we start our guide here. However, many extension boards are suitable for the standard development boards (so-called shields) that offer wired and wireless networking for Arduino. Also, their clones, made mainly by Chinese manufacturers, evolved into more sophisticated products, integrating, e.g. Arduino Mega 2560 and ESP8266 SoC into one development board.

Initially, all Arduino development boards were using ATMEL's MCUs. It is no longer the case due to the demand for integrated radio communication that ATMEL's MCUs lack.

At the moment of writing this book, the Arduino family contains 4 main branches:

- Nano: the tiniest yet powerful boards, newer models containing integrated radio modules such as Bluetooth and WiFi. Many 3rd party clones are available worldwide. A wide choice of shields provides the capability to extend the system with sensors and actuators. Depending on the board, you can find ATMEL's ATmegas, RP2040 or ARM Cortex-based ones.
- MKR: much bigger than Nano, providing broader wireless connectivity capabilities, including LoRa, Sigfox and NB-IoT. All of those boards use ARM Cortex M0, 32-bit MCU.
- Mega: the biggest development boards with many GPIO pins, efficiently allocating, e.g. dot matric displays that need to be connected with parallel interface. There are currently 3 family members, each using a different MCU: the original ATmega2560, ARM Cortex M3 and STM32.
- Classic: the most recognisable shape of the development boards still driving the look of the embedded systems and IoT: Arduino Uno's development board shape. The family uses ATmegas, Reneas and ARM Cortex M0+ MCUs.

There are also a dozen retired products that are still present on the market, such as the LilyPad series, which was intended to become intelligent jewellery and smart clothing, or the Yun series - the first of real IoT devices made by Arduino, that were designed to **run Linux distribution.**

Hardware

The Arduino boards work by reacting on signals at **inputs** that are received from various sensors, and after executing a **set of instructions**, an **output** is generated to respond to the environment. The input signal can be generated by pressing a button, receiving the radio or light signal, hearing the sound, perceiving an image of the situation using a camera resulting from the environmental sensor measurement, and many others. The output actions in the environment use output elements like actuators, blinking LEDs, audio devices, and others. The set of instructions executed to handle both sensors and actuators is created using the **Arduino programming language** based on an open-source programming framework called **Wiring** and the **Arduino Software** (IDE) based on **Processing**. The microcontroller or System on Chip is the most crucial element in the IoT and embedded devices built nowadays. It is not common to add peripheral elements external to the microcontroller, so the choice of this element influences almost all hardware parameters and the set of peripherals of the board. Because many versions of Arduino boards are available, only their selection based on the AVR family of microcontrollers is presented in the following chapters.

AVR microcontrollers

The initial, still very popular version of the Arduino board - Arduino Uno, is based on the ATmega328P microcontroller. The same chip is used in, e.g. Arduino Nano and Pro Mini. Arduino Leonardo or Micro is based on ATmega32u4, which has a built-in USB interface. The Arduino Mega board is created with an extended microcontroller ATmega2560, which has many more interface pins.

Memory

There are three different types of memory on the Arduino board: flash memory, SRAM and EEPROM. They are usually built into the main microcontroller, so their type determines the amount of memory available. A list of memory sizes regarding the microcontroller type is presented in table 11.

The **flash memory** stores the Arduino code, a non-volatile type of memory. That means the information in the memory is not deleted when the power is turned off.

The **SRAM** (static random access memory) is used for storing variables' values when the Arduino program is running. This volatile memory keeps information only until the power is turned off or the board is reset.

The **EEPROM** (electrically erasable programmable read-only memory) is a non-volatile type of memory that can be used as long-term memory storage.

Table 11: The Comparison of Basic Arduino Boards by Microcontroller Type and Memory Size

	Uno	**Leonardo**	**Micro**	**Mega**	**Nano**	**Pro Mini**
Microcontroller	ATmega328p	ATmega32u4	ATmega32u4	ATmega2650	ATmega328p	ATmega328p
Flash (kB)	32	32	32	256	32	32
SRAM (kB)	2	2	2.5	8	2	2
EEPROM (kB)	1	1	1	4	1	1

Peripherals

Peripherals are all functional units which play the roles of external elements of the CPU. Arduino boards are mainly implemented internally in the microcontroller, so the number and type of peripherals depend on the microcontroller version. Peripherals include Timers, Communication and networking interfaces, GPIOs, Analog comparators and converters, and supervisory units.

Networking

The basic Arduino boards do not implement any networking connectivity. This capability to use Ethernet, WiFi, Bluetooth, ZigBee, and other wireless protocols can be added with an external module or shield. Example shields are Arduino Ethernet Shield, WiFly Shield, Arduino WiFi Shield, Electric Imp Shield, XBee Shield, Cellular Shield SM5100B and GPS Shield. In the simplest version, the WiFi module like Espressif ESP01S can be connected to Arduino's serial port and programmed with AT commands.

Communication Interfaces

Communication interfaces for Arduino are used to send and receive information to and from other external devices. Standard interfaces for Arduino are UART, I2C (also called TWI - Two-Wire Interface), SPI, and USB.

Timers

Timers are implemented as the essential elements of almost every microcontroller. These units can operate in timer mode or counter mode. In the first mode, they count pulses generated internally in the microcontroller. This makes it possible to generate square signals of specified frequency, signal periodic interrupts, or generate pulse width modulated signals at PWM outputs. In counter mode, counting the number of external pulses is possible. In selected Arduino boards, there are 8-bit and 16-bit timers, an additional real-time clock with a separate generator, and a watchdog timer that can work as a supervisory unit which resets the microcontroller in case of software hang-up. The list of interfaces and timers is presented in table 12.

Table 12: The Comparison of Arduino Boards by Interfaces and Timers Available

	Uno	Leonardo	Micro	Mega	Nano	Pro Mini
USB	1 USB B	1 Micro	1 Micro	1 USB B	1 Mini	–
UART	1	1	1	4	1	1
I2C	1	1	1	1	1	1
SPI	1	1	1	1	1	1
8-bit Timer	1	1	1	2	1	1
16-bit Timer	2	2	2	4	2	2
Watchdog Timer	1	1	1	1	1	1
Real-time clock	1	-	-	1	1	1

Video subsystem

Arduino boards do not contain specialised video chips. Their memory size does not allow them to generate, capture, or even store complex high-resolution images. The most common approach to display images is connecting the LCD, OLED or TFT display with an SPI port. Connecting the camera is even more complicated. None of the microcontrollers used in basic Arduino boards have an adequate camera port to convey high-speed video signals. An answer to this challenge is the Arducam, which implements the camera and the hardware to capture the image to the RAM. It can be connected to an Arduino board with an SPI interface, allowing it to read and process the image data at the main processor speed.

Hardware connectors

Digital Input/Output Pins

Digital input/output (I/O) pins are contacts on the Arduino board that can receive or transmit a digital signal. The status of the pin can be set either to 0, which represents *LOW* signal or to 1 – *HIGH* signal. The maximum current of the pin output is 40 mA.

Pulse Width Modulation

Pulse Width Modulation (PWM) is a function of a pin to generate a square wave signal with a variable length of the HIGH level of the output signal. The PWM is used for digital pins to simulate the analogue output.

Analog Pins

Analog pins convert the analogue input value to a 10-bit number using Analog Digital Converter (ADC). This function maps the input voltage between 0 and the reference voltage to numbers between 0 and 1023. By default, the reference voltage is set to a microcontroller operating voltage. Usually, it is 5 V or 3.3 V. Also, other internal or external reference sources, for example, AREF pin, can be used.

A list of pins and hardware interfaces for popular Arduino boards is present in table 13.

Table 13: The Comparison of Basic Arduino Boards by the Number of Pins in Hardware Interfaces

	Uno	Leonardo	Micro	Mega	Nano	Pro Mini
Digital I/O	14	20	20	54	22	14
PWM	6	7	7	12	6	6
Analog pins	6	12	12	16	8	6

Power and Other Pins

- Power pins on the Arduino board connect the power source to the microcontroller and/or voltage regulators. They can also be a power source for external components and devices.
- The VIN pin connects the external power source to the internal regulator to provide the regulated 5 V output. The input voltage of the board must be within the specific range, mainly between 7 V and 12 V.
- The 5V pin is used to supply a microcontroller with the regulated 5 V from the external source or is used as a power source for the external components in the case when the board is already powered using the USB interface or the VIN pin.
- The 3V3 pin provides the regulated 3.3 V output for the board components and external devices. The GND (ground pin) is where the negative terminal of the power supply is applied.
- The reset pin and button reset the Arduino board and the program. Resetting using the reset pin is done by connecting it to the GND.

5.1.2. Espressif Family

Arduino and a vast amount of peripheral boards lack integration of the networking capabilities in one SoC. Espressif ESP series was the natural answer for this disadvantage as their ESP 8266 with integrated WiFi, introduced in 2014, is widely recognised as a turning point for the IoT market, delivering de-facto fully functional IoT chip, providing high performance and low power to the end users and developers. ESP32, launched in 2016, brought even more disruptive effects to the IoT ecosystems, introducing an additional Bluetooth interface to the above. The most popular series of these microcontrollers are ESP8266EX, ESP32, ESP32S, C and H families.

The significant difference is that ESP SoCs (both 8266 and 32) use 3.3 V logic, while most (but not all!) Arduinos use 5 V logic. This can be easily handled using one or bi-directional voltage converters/adapters. Additionally, many ESP boards and development kits offer double power source, including 5 V, even if the device itself still operates on 3.3 V

ESP 8xxx Family

At the moment, the ESP 8xxx family includes the following chips:

- ESP8266EX (figure 69),
- ESP8285 (figure 70).

Figure 69: ESP8266EX chip

Figure 70: ESP8285 chip

The ESP8285 module continues the ESP8266 line with 1 MB of built-in flash, higher integration, and reduced dimensions.

ESP 8266

ESP 8266 General Information

The ESP8266 is a low-cost system-on-chip (SoC) microcontroller with WiFi and full TCP/IP stack capability [66]. The main advantages of that family are:

- Low power consumption,
- Availability of WiFi and Bluetooth connections,
- Wide availability of low-cost modules from various suppliers,
- Wide availability in a variety of form factors, including SoC.

The low price and the fact that the module had very few external components, which suggested that it could eventually be very inexpensive in volume, attracted many users to explore it.

Esp8266 Architecture Overview

The main standard features of the ESP8266EX are:
Processor

- **Main processor:** L106 32-bit RISC microprocessor core based on the Tensilica Xtensa Diamond Standard 106Micro running at 80 MHz. Both the CPU and flash clock speeds can be doubled by overclocking on some devices. CPU can be run at 160 MHz, and flash can be sped up from 40 MHz to 80 MHz. Success varies from chip to chip.

Memory

- **External QSPI flash**: up to 16 MB is supported (512 kB to 4 MB typically included),
- 32 kB instruction RAM,
- 32 kB instruction cache RAM,
- 80 kB user data RAM,
- 16 kB ETS system data RAM.

Interfaces

- IEEE 802.11 b/g/n WiFi ,
- Integrated TR switch, balun, LNA, power amplifier and matching network WEP or WPA/WPA2 authentication, or open networks,
- 17 GPIO pins,
- SPI,
- I²C (software implementation),
- I²S interfaces with DMA (sharing pins with GPIO),
- UART on dedicated pins, plus a transmit-only UART can be enabled on GPIO2,
- 10-bit ADC (successive approximation ADC).

Figure 71 shows functional block diagram of ESP8266 chip [67].

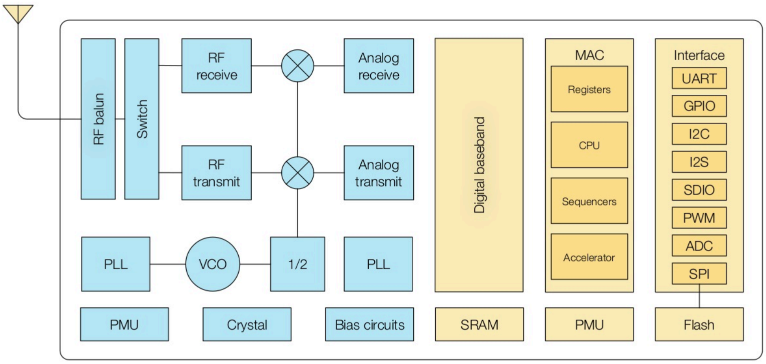


Figure 71: ESP8266&ESP8285 functional block diagram

ESP8266 Modules

Many still popular ESP8266-based modules are on the market [68]. These modules combine the ESP8266EX microcontroller and additional components mounted on the PCB.

The most popular are these produced by AI-Thinker and remain the most widely available [69].

- ESP-01 (512 kB Flash),
- ESP-01S (1 MB Flash),
- ESP-12 (FCC and CE approved),
- ESP-12E,
- ESP-12F (4 MB Flash, FCC and CE approved).

Popular modules from other manufacturers:

- Sparkfun ESP8266 Thing,
- Wemos D1 mini, D1 mini Pro [70].

The Espressif company also produces ready-made modules using the aforementioned chip. This is the series of ESP8266-based modules made by Espressif (table 14).

Table 14: Espressif ESP8266 modules

Name	Active pins	LEDs	Antenna	Shielded	Dimensions (mm)	Notes
ESP-WROOM-02[71]	18	No	PCB trace	Yes	18 × 20	FCC ID 2AC7Z-ESPWROOM02
ESP-WROOM-02D[72]	18	No	PCB trace	Yes	18 × 20	FCC ID 2AC7Z-ESPWROOM02D. Revision of ESP-WROOM-02 is compatible with both 150-mil and 208-mil flash memory chips
ESP-WROOM-02U[73]	18	No	U.FL socket	Yes	18 × 20	Differs from ESP-WROOM-02D in that includes an U.FL compatible antenna socket connector
ESP-WROOM-S2[74]	20	No	PCB trace	Yes	16 × 23	FCC ID 2AC7Z-ESPWROOMS2

The most widely used chipset ESP-01 is presented in (figure 72) and its pinout on (figure 73).

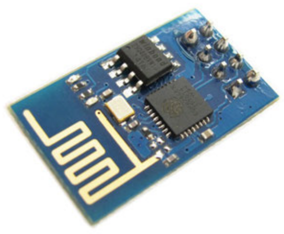


Figure 72: ESP-01

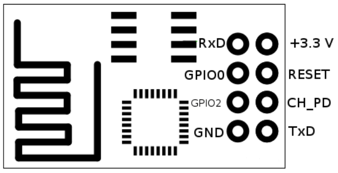


Figure 73: ESP-01 pinout

Module ESP12F with pinout is presented on (figure 74) and its pinout on (figure 75).

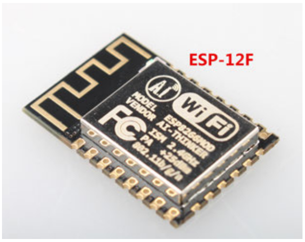


Figure 74: ESP-12F

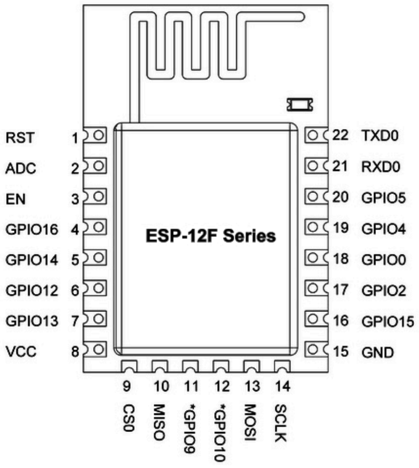
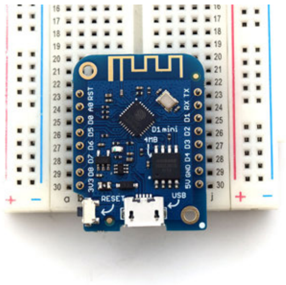


Figure 75: ESP-12F pinout

Among the other modules, it is worth being interested in WEMOS modules [75] (figure 76, figure 77). The WEMOS company offers dedicated sensor modules and inputs/outputs compatible with the processor modules. They are called WEMOS shields (figure 78).



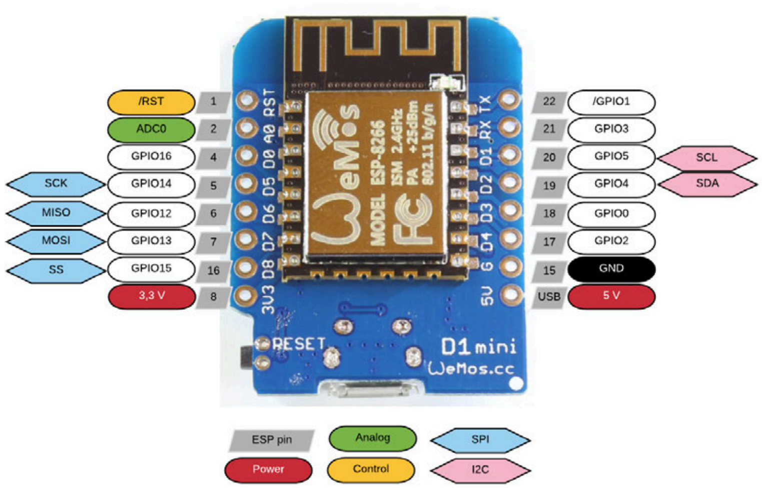


Figure 76: Wemos D1 mini with pinout

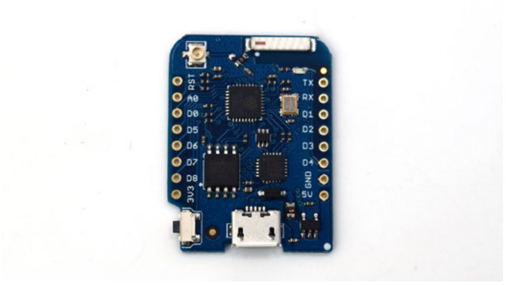


Figure 77: Wemos D1 Pro

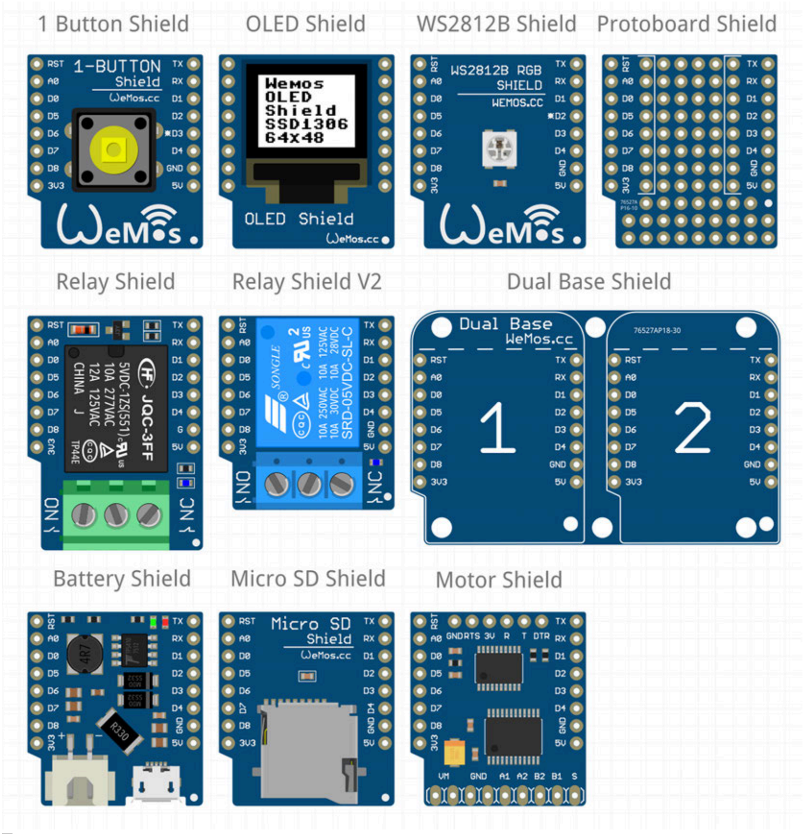


Figure 78: Wemos I/O shields

ESP 8285

ESP8285 Architecture Overview

Main differences between ESP8285 and ESP8266 are:
Processor

- L106 32-bit RISC microprocessor core running at 160MHz

Memory

- **Internal 1MB or 2MB** program memory,

Power consumption

- ESP8285 has a lower power consumption than the ESP8266. The ESP8285 consumes 2.7 mA in deep-sleep mode, vs 10 mA ESP8266,
- Wake up within 2 ms.

Security

- Supports secure boot and flash encryption.

ESP32 General Information

ESP32 is a low-cost, low-power system on a chip (SoC) series microcontroller with WiFi & dual-mode Bluetooth capabilities [76]. ESP32 SoC is highly integrated with built-in antenna switches, power amplifiers, low-noise receive amplifiers, filters, and power management modules. Inside all families of ESP32, there is a single-core or dual-core Tensilica Xtensa LX6 microprocessor with a clock rate of up to 240 MHz. ESP32 is designed for mobile, wearable electronics, and Internet-of-Things (IoT) applications. It features all the state-of-the-art characteristics of low-power chips, including fine-grained clock gating, multiple power modes, and dynamic power scaling. For now, the ESP32 family includes the following chips in mass production:

- ESP32-D0WD-V3 (figure 79) ,
- ESP32-U4WDH (figure 80),
- ESP32-PICO-D4 - SiP (system in package) (figure 81) – additionally contains crystal oscillator, 4MB flash memory, filter capacitors and RF matching links,
- ESP32-PICO-V3 - SiP (system in package) – new core (ECO V3)
- ESP32-PICO-V3-02 - SiP (figure 82 – package size is slightly thicker - 7 × 7 × 1.11 (mm), the chip integrates 8 MB flash and 2 MB PSRAM with different pin layout,

and older chips, not for new designs:

- ESP32-D0WDQ6 (figure 83),
- ESP32-D0WDQ6-V3,
- ESP32-D0WD (figure 84),
- ESP32-S0WD (figure 86).

Figure 79: ESP32-D0WD-V3

Figure 80: ESP32-U4WDH

Figure 81: ESP32-PICO-D4

Figure 82: ESP32-PICO-V3-02

Figure 83: ESP32-D0WDQ6

Figure 84: ESP32-D0WD

Figure 85: ESP32-D2WD

Figure 86: ESP32-S0WD

ESP32 Architecture Overview

The functional block diagram of the ESP32 chip is shown in figure 87. Main common features of the ESP32 are: [77] [78].

Processors

- **Main processor:** Tensilica Xtensa 32-bit LX6 microprocessor.
  - **Cores**: 2 or 1 (depending on variation). (All chips in the ESP32 series are dual-core except for ESP32-S0WD, which is single-core.)
  - Internal 8 Mhz oscillator with calibration.
  - External 2 MHz to 60 MHz crystal oscillator (40 MHz only for WiFi/BT functionality).
  - External 32 kHz crystal oscillator for RTC with calibration.
  - **Clock frequency:** up to 240 MHz.
  - **Performance:** up to 600 DMIPS.

- **Ultra low power co-processor:** allows you to do ADC conversions, I2C connecting, computation, and level thresholds while in a deep sleep.

Wireless connectivity

- **WiFi:** 802.11 b/g/n/e/i (802.11n @ 2.4 GHz up to 150 Mbit/s) with simultaneous Infrastructure BSS Station mode/SoftApp mode/Promiscuous mode.
- **Bluetooth:** v4.2 BR/EDR and Bluetooth Low Energy (BLE) with multi-connections Bt and BLE and simultaneous advertising and scanning capability.

Memory: Internal memory

- **ROM:** 448 kB (booting and core functions).

- **SRAM:** 520 kB (for data and instruction).
- **RTC fast SRAM:** 8 kB (for data storage and main CPU during RTC Boot from the deep-sleep mode).
- **RTC slow SRAM:** 8 kB (for co-processor accessing during deep-sleep mode).
- **eFuse:** 1 Kibit (of which 256 bits are used for the system (MAC address and chip configuration), and the remaining 768 bits are reserved for customer applications, including Flash-Encryption and Chip-ID).
- **Embedded flash** (flash connected internally via IO16, IO17, SD_CMD, SD_CLK, SD_DATA_0 and SD_DATA_1 on ESP32-D2WD and ESP32-PICO-D4):
    - 0 MB (ESP32-D0WDQ6, ESP32-D0WD, and ESP32-S0WD chips),
    - 2 MB (ESP32-D2WD chip).

External Flash & SRAM

- ESP32 supports up to four 16 MB external QSPI flashes and SRAMs with hardware encryption based on AES to protect developers' programs and data. ESP32 can access the external QSPI flash and SRAM through high-speed caches.
- Up to 16 MB of external flash are memory-mapped onto the CPU code space, supporting 8-bit, 16-bit and 32-bit access. Code execution is supported.
- Up to 8 MB of external flash/SRAM memory is mapped onto the CPU data space, supporting 8-bit, 16-bit and 32-bit access. Data-read is supported on the flash and SRAM. Data write is supported on the SRAM.

ESP32 chips with embedded flash do not support the address mapping between external flash and peripherals.

Peripheral Input/Output

- Rich peripheral interface with DMA includes capacitive touch (10× touch sensors).
- 12-bit ADCs (analog-to-digital converter) up to 18 channels.
- 2 × 8 bit DACs (digital-to-analog converter).
- 2 × I^2C (Inter-Integrated Circuit.
- 3x UART (universal asynchronous receiver/transmitter).
- CAN 2.0 (Controller Area Network).
- 4 × SPI (Serial Peripheral Interface).
- 2 × I^2S (Integrated Inter-IC Sound).
- RMII (Reduced Media-Independent Interface).
- Motor PWM (pulse width modulation).
- LED PWM up to 16 channels.
- Hall sensor.
- Internal temperature sensor.

Security

- Secure boot.

- Flash encryption.
- IEEE 802.11 standard security features are all supported, including WFA, WPA/WPA2 and WAPI.
- 1024-bit OTP, up to 768-bit for customers.
- Cryptographic hardware acceleration:
  - AES,
  - SHA-2,
  - RSA,
  - elliptic curve cryptography (ECC),
  - random number generator (RNG).

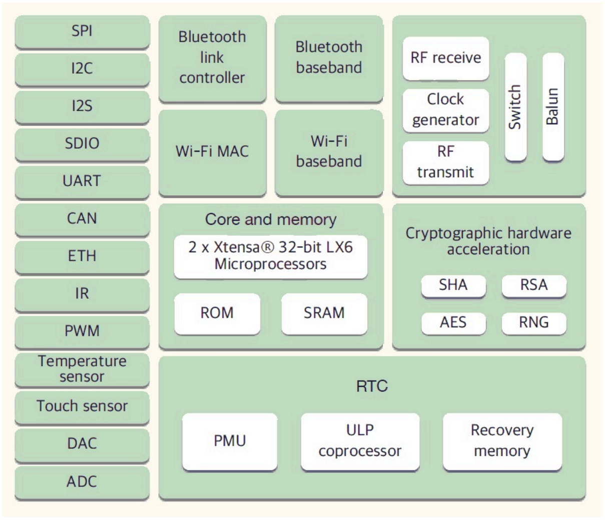


Figure 87: ESP32 Functional block diagram

ESP32 Modules

The company also produces ready-made modules using the processors above [79]. These modules combine ESP32 microcontroller and additional components mounted on PCB with EM shield (table 15):

Table 15: Espressif ESP32 modules

Module	Chip	Number of cores	Flash, MB	PSRAM, MB	Ant.	Dimensions, mm
ESP32-WROOM-32(figure 88)	ESP32-D0WDQ6	2	4	–	PCB	18 × 25.5 × 3.1
ESP32-WROOM-32D	ESP32-D0WD	2	4, 8, or 16	–	PCB	18 × 25.5 × 3.1
ESP32-WROOM-32U(figure 89)	ESP32-D0WD	2	4, 8, or 16	–	U.FL	18 × 19.2 × 3.1
ESP32-SOLO-1	ESP32-S0WD	1	4	–	PCB	18 × 25.5 × 3.1
ESP32-WROVER (PCB)(figure 90)	ESP32-D0WDQ6	2	4	8	PCB	18 × 31.4 × 3.3
ESP32-WROVER (IPEX)	ESP32-D0WDQ6	2	4	8	U.FL	18 × 31.4 × 3.3
ESP32-WROVER-B	ESP32-D0WD	2	4, 8, or 16	8	PCB	18 × 31.4 × 3.3

Module	Chip	Number of cores	Flash, MB	PSRAM, MB	Ant.	Dimensions, mm
ESP32-WROVER-IB(figure 91)	ESP32-D0WD	2	4, 8, or 16	8	U.FL	18 × 31.4 × 3.3

- U.FL - U.FL / IPEX antenna connector

Figure 88: ESP32-WROOM-32

Figure 89: ESP32-WROOM-U

Figure 90: ESP32-WROVER

Figure 91: ESP32-WROVER-I

ESP32 Pico Architecture Overview

ESP32-PICO-D4

The ESP32-PICO-D4[80] is a System-in-Package (SiP) module that is based on ESP32. ESP32-PICO-D4 integrates all peripheral components in one package, including a crystal oscillator, flash, filter capacitors and RF matching links. The module is as small as 7.0 mm × 7.0 mm × 0.94 mm, thus requiring minimal PCB area. The main characteristics that distinguish it from the ESP32 family are:

- Integrated crystal oscillator, filter capacitors and RF matching circuit,
- Internal built-in memory 4 MB SPI flash,
- chip size 7.0 mm × 7.0 mm × 0.94 mm.

ESP32-PICO-V3

The ESP32-PICO-V3[81] is a System-in-Package (SiP) module that is based on ESP32 but with a new ECO V3 wafer. The module is as small as 7.0 mm × 7.0 mm × 1.11 mm. Distinguishing features from the ESP32-PICO-D4 chips are:

- New silicone wafer ECO V3,
- 16 kB SRAM in RTC,
- chip size 7.0 mm × 7.0 mm × 1,11 mm.

ESP32-PICO-V3-02

The ESP32-PICO-V3-02[82] is based on ESP32-PICO-V3 with additional SPi flash and SPI PSRAM. Distinguishing features from the ESP32-PICO-V3 chips are:

- Internal built-in SPI flash memory 8 MB,
- Internal built-in SPI PSRAM memory 2 MB,
- additional GPIO pin - GPIO20,
- For chip security purposes, flash pins DI, DO, /HOLD, /WP and PSRAM pins SI/SIO0, SO/SIO1, SIO2, and SIO3 are not led out.

ESP32-PICO Modules

The company also produces ready-made modules using the ESP32-PICO SOCs [83] [84].

These modules combine ESP32 microcontroller and additional components mounted on PCB with EM shield (table 16).

Table 16: Espressif ESP32-PICO modules

Module	Chip	Number of cores	Flash, MB	PSRAM, MB	Ant.	Dimensions, mm
ESP32-PICO-MINI-02 (figure 92)	ESP32-PICO-V3-02	2	8	2	PCB	13.2 × 16.6 × 2.4
ESP32-PICO-MINI-02U (figure 93)	ESP32-PICO-V3-02	2	8	2	IPEX	13.2 × 11.2 × 2.4
ESP32-PICO-V3-ZERO (figure 94) for Alexa Connect Kit (ACK)	ESP32-PICO-V3	2	4	–	PCB&IPEX	16 × 23 × 2.3

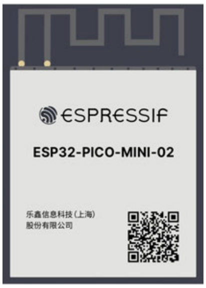


Figure 92: ESP32-PICO-mini-02

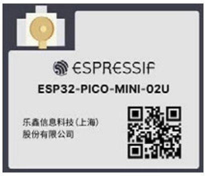


Figure 93: ESP32-PICO-mini-02U

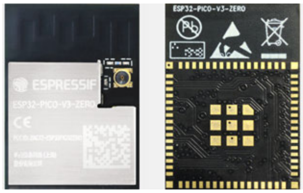


Figure 94: ESP32-PICOV3-ZERO

ESP32 Development Kits

To accelerate the design of circuits, developers can use specially prepared sets with ESP32, which are ready to use. The original Espressif best-known small development boards are:

- ESP32-DevkitC (figure 95),
- ESP32-PICO-KIT-V4 (figure 96),
- ESP32-PICO-KIT-1 with ESP32-PICO-V3 (figure 97),
- ESP32-PICO-DevKitM-2 with ESP32-PICO-Mini-02 (figure 98).

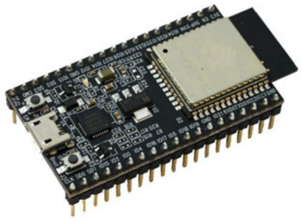


Figure 95: ESP-32-DevkitC[85]

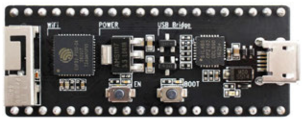


Figure 96: ESP-32-PICO-KIT-V4[86]

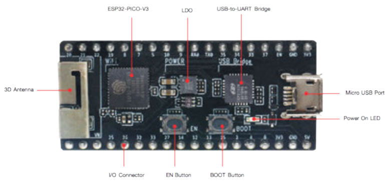


Figure 97: ESP-32-PICO-KIT-1[87]

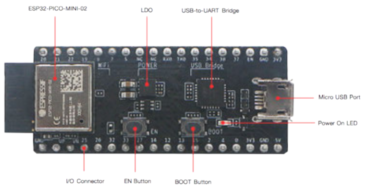


Figure 98: ESP-32-PICO-DEVKITM-2[88]

General Purpose Input-Output (GPIO) Connector
Each ESP32 is equipped with a standard 38/40-pis male connector containing universal GPIO ports, VCC 3.3/5 V, GND, CLK, I2C/SPI bus pins, which developers can use to connect their external sensors, switches and other controlled devices to the ESP32 board and then program their behaviour within the code loaded to the board.

- ESP32-DevkitC v2 pins (figure 99).

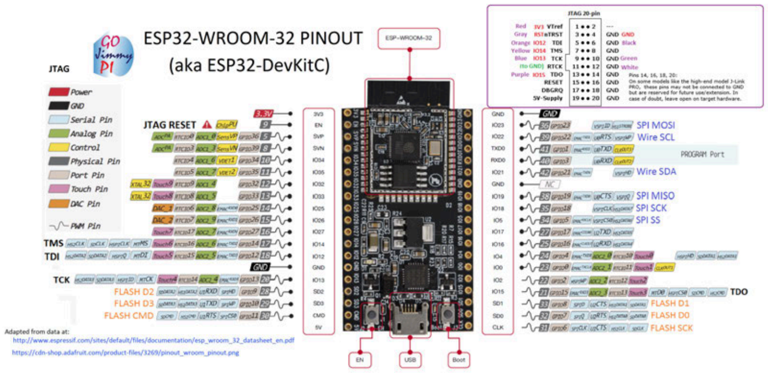


Figure 99: ESP32-DevkitC pins

- ESP32-PICO D4 pins (figure 100).

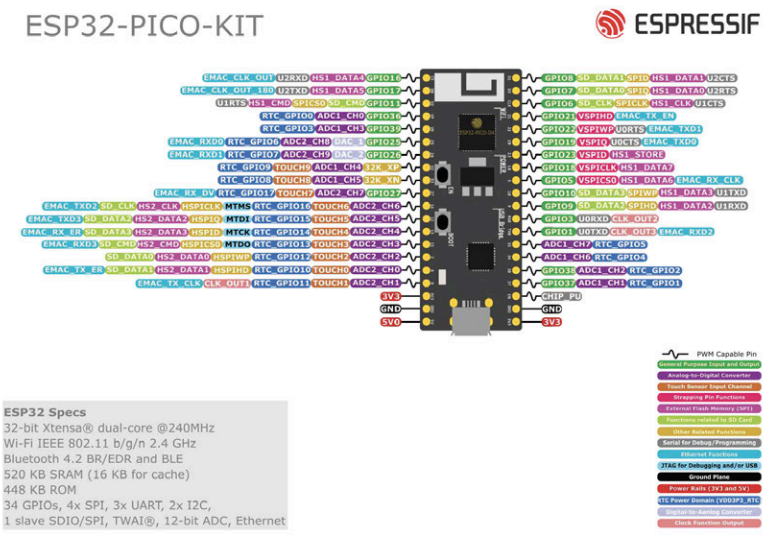

ESP32-PICO-KIT
ESPRESSIF
PWM Capable Pin
General Purpose Input and Output
Analog-to-Digital Converter
Touch Sensor Input Channel
Strapping Pin Functions
External Flash Memory (SPI)
Functions related to SD Card
Other Related Functions
Serial for Debug/Programming
Ethernet Functions
JTAG for Debugging and/or USB
Ground Plane
Power Rails (3V3 and 5V)
RTC Power Domain (VDD3P3_RTC)
Digital-to-Analog Converter
Clock Function Output
ESP32 Specs
32-bit Xtensa® dual-core @240MHz
Wi-Fi IEEE 802.11 b/g/n 2.4 GHz
Bluetooth 4.2 BR/EDR and BLE
520 KB SRAM (16 KB for cache)
448 KB ROM
34 GPIOs, 4x SPI, 3x UART, 2x I2C,
1 slave SDIO/SPI, TWAI®, 12-bit ADC, Ethernet

Figure 100: ESP32-Pico Kit pins [89]

- ESP32 Wemos Pro pins (figure 101).

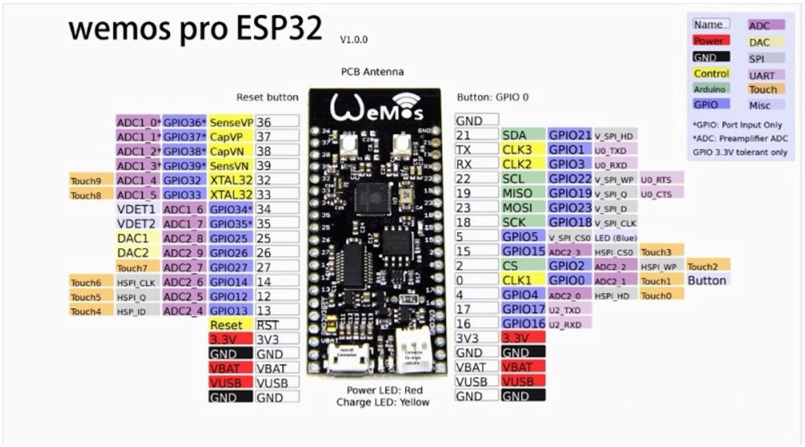


Figure 101: ESP32 Wemos Pro pins

In addition to modules for developers, small microcomputers with ESP processors are also produced. They are very convenient to use. They often include one or two buttons, an RGB LED or LCD, and everything enclosed in a case and ready for use in small projects. One of them is the ESP-PICO-D4 based M5 Atom-lite (figure 102):

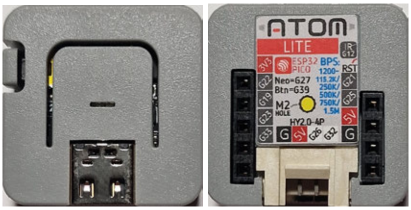


Figure 102: M5ATOM-lite top&bottom view

An additional advantage of such a module for use in mini projects is the available housing with a prototype PCB shown in figure 103

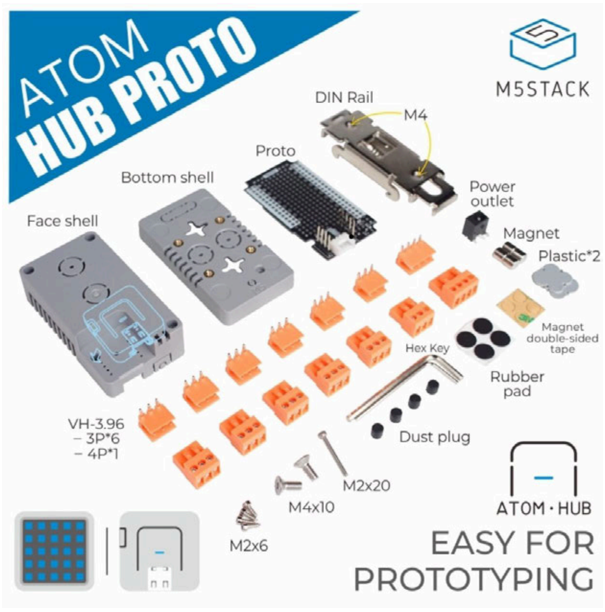


Figure 103: Housing with proto board for Atom -lite[90].

ESP32-Sx Family

ESP32-S2

ESP32-S2 General Information

The Espressif ESP32-S2 family is a series of low-power, single-core microcontrollers built on the Espressif IoT platform. They feature a highly integrated SoC (System on Chip) architecture, combining a CPU, WiFi connectivity, and various peripherals in a compact package. The ESP32-S2 chips are designed for IoT applications, smart home devices, wearables and more, offering enhanced security features, low power consumption, and support for various communication protocols. These microcontrollers are known for their cost-effectiveness and capabilities of connected devices. ESP32-S2 SoC is based on an Xtensa single-core 32-bit LX7 microcontroller with an additional ultra-low power (ULP) coprocessor with a Wi-Fi 2.4GHz radio and numerals peripherals. For now, the ESP32-S2 series includes the following chips in mass production:

- ESP32-S2 (figure 104) ,
- ESP32-S2F (figure 105) .

Figure 104: ESP32-S2

Figure 105: ESP32-S2F

ESP32-S2 Architecture Overview

Figure 106 shows functional block diagram of ESP32-S2 chip[91]. The main common features of the ESP32-S2 are:

Processors

- **Main processor:** • Xtensa® single-core 32-bit LX7 microprocessor, up to 240 MHz
  - **Cores**: 1
- **Ultra low power coprocessor:**
  - **Cores**: 1

Wireless connectivity

- **WiFi:** 802.11 b/g/n/(802.11n @ 2.4 GHz up to 150 Mbit/s) with simultaneous Infrastructure BSS Station mode/SoftApp mode/Promiscuous mode.

Memory: Internal memory

- **ROM:** 128 kB (for booting and core functions),
- **SRAM:** 320 kB (for data and instruction),
- **RTC SRAM:** 16 kB (for data storage and main CPU during RTC Boot from the deep-sleep mode),
- **Embedded flash:**
  - 0 MB (ESP32-S2, ESP32-S2R2 chips),
  - 2 MB (ESP32-S2FH2 chip),
  - 4 MB (ESP32S2FH4, ESP32FN4R2 chips).
- **Embedded PSRAM:**

  - 0 MB (ESP32-S2, ESP32-S2FH2, ESP32S2FH4 chips),
  - 2 MB (ESP32FN4R2, ESP32-S2R2 chips).

Peripheral Input/Output

- 43 programmable GPIOs,
- 2 × I²C (Inter-Integrated Circuit,
- 2 x UART (universal asynchronous receiver/transmitter),
- 4 × SPI (Serial Peripheral Interface),
- 1 × I²S (Integrated Inter-IC Sound),
- 1 x RMT (TX/RX),
- Motor PWM (pulse width modulation),
- LED PWM up to 8 channels,
- DMA controller,
- 1 x TWAI controller compatible with CAN Spec. 2.0,
- 4 x pulse counters,
- 1 x full-speed USB OTG,
- 1 x DVP 8/16 camera interface (I2S),
- 1 x LCD serial interface (SPI),
- 1 x LCD parallel interface.

Analog interfaces

- 2 x 13-bit ADCs up to 20 channels,
- 2 x 8-bit DACs,
- 14 x touch sensing GPIO,
- 1 x temperature sensor.

Security

- Secure boot,
- Flash encryption,
- IEEE 802.11 standard security features are all supported, including WFA, WPA/WPA2 and WAPI,
- 4096-bit OTP, up to 1792-bit for customers,
- Cryptographic hardware acceleration:
  - AES-128/192/256,
  - HMAC,
  - RSA,
  - random number generator (RNG).

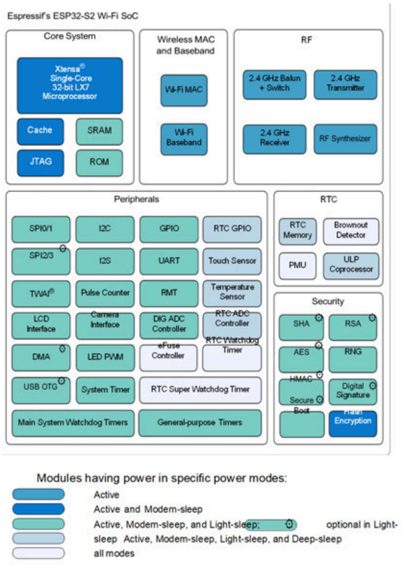


Figure 106: ESP32-S2 Functional block diagram

ESP32-S2 Modules

The company also produces ready-made modules for easier implementation in user systems. These modules combines ESP32-S2 microcontroller and additional components mounted on PCB with EM shield [92](table 17):

Table 17: Espressif ESP32-S2 modules

Module	Chip Embedded	Dimensions (mm)	Pins	Flash (MB)	PSRAM (MB)	Antenna	Development Board
ESP32-S2-MINI-2 (figure 107)	ESP32-S2FH4 ESP32-S2FN4R2	15.4×20×2.4	65	4	0,2	PCB	ESP32-S2-DevKitM-1
ESP32-S2-MINI-2U	ESP32-S2FH4 ESP32-S2FN4R2	15.4×15.4×2.4	65	4	0,2	IPEX	ESP32-S2-DevKitM-1
ESP32-S2-SOLO-2 (figure 108)	ESP32-S2 ESP32-S2R2	18×25.5×3.1	41	4	0,2	PCB	ESP32-S2-DevKitC-1
ESP32-S2-SOLO-2U	ESP32-S2 ESP32-S2R2	18×19.2×3.2	41	4	0,2	IPEX	ESP32-S2-DevKitC-1
ESP32-S2-MINI-1	ESP32-S2FH4 ESP32-S2FN4R2	15.4×20×2.4	65	4	0,2	PCB	ESP32-S2-DevKitM-1
ESP32-S2-MINI-1U (figure 109)	ESP32-S2FH4 ESP32-S2FN4R2	15.4×15.4×2.4	65	4	0,2	IPEX	ESP32-S2-DevKitM-1
ESP32-S2-SOLO	ESP32-S2 ESP32-S2R2	18×25.5×3.1	40	4,8,16	0,2	PCB	ESP32-S2-DevKitC-1
ESP32-S2-SOLO-U	ESP32-S2 ESP32-S2R2	18×19.2×3.2	40	4,8,16	0,2	IPEX	ESP32-S2-DevKitC-1

Figure 107: ESP32-S2-mini2

Figure 108: ESP32-S2-solo2

Figure 109: ESP32-S2-mini-1U

ESP32-S2 Development Kits

For convenience, used by users of all skill levels, Espressif produces entry-level development boards using the ESP32-S2 SOCs. Those boards integrate complete Wi-Fi functions. Most ESP32-S2 I/O pins are broken out to the pin headers on both sides for easy interfacing. Users can connect peripherals with jumper wires or mount the development kit on a breadboard. Many different companies offer ready-made boards with processors. The original Espressif best-known small development boards are:

- ESP32-S2-DevkitM [93](figure 110) ,
- ESP32-S2-DevkitC [94](figure 111) ,

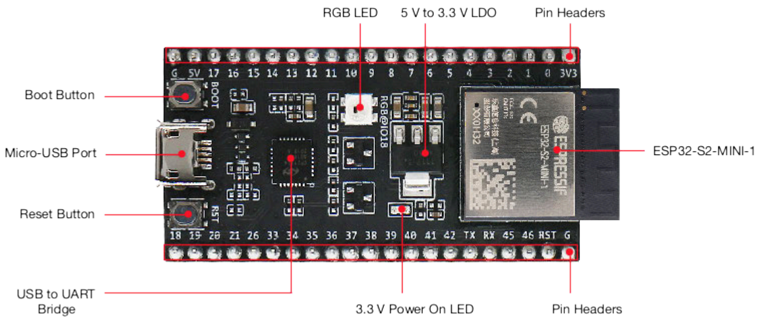


Figure 110: ESP32-S2-DevkitM

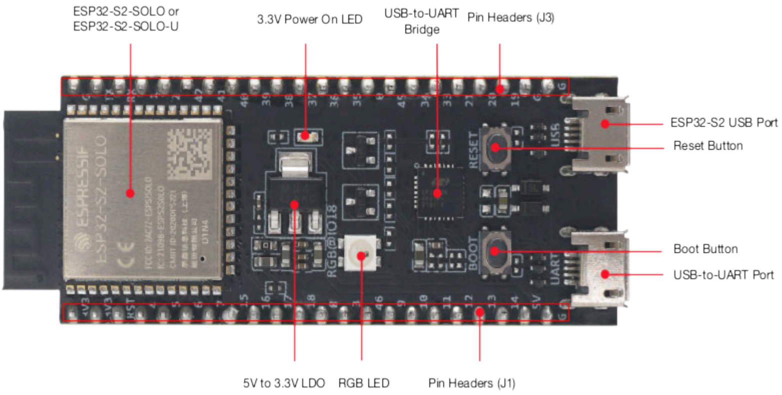


Figure 111: ESP32-S2-DevkitC

ESP32-S3

ESP32-S3 General Information

The ESP32-S3[95][96] is an advanced version within Espressif S family, offering improved performance and expanded capabilities compared to its predecessors. ESP32-S3 is an MCU with a dual-core 32-bit Xtensa LX7 microprocessor, dual ULP coprocessors with Wifi 2.4 GHz and Bluetooth LE radio, and numerous useful peripherals. ESP32-S3 offers enhanced processing power, lower power consumption, and improved IoT and wireless connectivity application features. ESP32-S3 is designed for mobile systems, Industrial and Home Automation, Health Care devices, Touch and Proximity Sensing, wearable electronics, and Internet-of-Things (IoT) applications. In addition, ESP32-S3 includes support for vector instructions in the MCU, which provides acceleration for neural network computing and signal processing workloads. The ESP32-S3 is the first low-cost microcontroller with a built-in peripheral that can drive TTL displays, and it can come with enough PSRAM to buffer those large images. For now, the ESP32-S3 family includes the following chips in mass production:

- ESP32-S3 (figure 112) ,
- ESP32-S3-Pico-1 (figure 113).

Figure 112: ESP32-S3

Figure 113: ESP32-S3-PICO-1

ESP32-S3 Architecture Overview

Figure 114 shows a functional block diagram of the ESP32-S3 chip. ESP32-S3's main common features of the ESP32-S3 are:

Processors

- **Main processor:** • Xtensa® dual-core 32-bit LX7 microprocessor, up to 240 MHz:
  - **Cores**: 2

- **Ultra low power coprocessor:**
  - **Cores**: 2
  - ULP-RISC-V coprocessor - based on RISC-V instruction set architecture:
    - Support for RV32IMC instruction set,
    - Thirty-two 32-bit general-purpose registers,
    - 32-bit multiplier and divider,
    - Support for interrupts,
    - Booted by the CPU, its dedicated timer, or RTC GPIO.

  - ULP-FSM coprocessor - based on finite state machine:
    - Support for common instructions, including arithmetic, jump, and program control instructions,
    - Support for on-board sensor measurement instructions,
    - Booted by the CPU, its dedicated timer, or RTC GPIO.

Wireless connectivity

- **WiFi:** 802.11 b/g/n/mc (802.11n @ 2.4 GHz up to 150 Mbit/s) with simultaneous Infrastructure BSS Station mode/SoftApp mode/Promiscuous mode.
- **Bluetooth:**
  - Low Energy Bluetooth 5, Bluetooth mesh,
  - Speed 125kbps, 500 kbps, 1 Mbps, 2 Mbps,
  - Internal sharing antenna with WiFi.

Memory: Internal memory:

- **ROM:** 384 kB (booting and core functions),
- **SRAM:** 512 kB (for data and instruction),
- **RTC SRAM:** 16 kB (for data storage and main CPU during RTC Boot from the deep-sleep mode),
- **Embedded flash:**
  - 0 MB (ESP32-S3, ESP32-S3R2, ESP32-S3R8, ESP32-S3R8V chips),
  - 4 MB (ESP32-S3FH4R2 chip),
  - 8 MB (ESP32-S3FN8 chip).

- **Embedded PSRAM:**
  - 0 MB (ESP32-S3, ESP32-S3FN8 chips),
  - 2 MB (ESP32-S3R2, ESP32-S3FH4R2 chips),
  - 8 MB (ESP32-S3R8, ESP32-S3R8V chips).

Peripheral Input/Output:

- 45 programmable GPIOs,
- 2 × I^{2}C (Inter-Integrated Circuit,
- 3 x UART (universal asynchronous receiver/transmitter),
- 4 × SPI (Serial Peripheral Interface),
- 2 × I^{2}S (Integrated Inter-IC Sound),
- 1 x RMT (TX/RX),
- Motor PWM (pulse width modulation),
- LED PWM up to 8 channels,
- DMA controller with 5 transmit and 5 receive channels,
- 1 x TWAI controller compatible with CAN Spec. 2.0,
- 4 x pulse counters,
- 1 x full-speed USB OTG,
- 1 × USB Serial/JTAG controller,
- 1 x DVP 8/16 camera interface (I2S),
- 1 x LCD parallel interface,
- 1 × SD/MMC host controller.

Analog interfaces:

- 2 x 12-bit ADCs up to 20 channels,
- 14 x touch sensing GPIO,
- 1 x temperature sensor.

Low power management:

- Power Management Unit with five power modes,
- Ultra-low-power (ULP) coprocessors.

Security:

- Secure boot,
- Flash encryption,
- IEEE 802.11 standard security features are all supported, including WFA, WPA/WPA2 and WAPI,
- 4096-bit OTP, up to 1792-bit for customers,
- Cryptographic hardware acceleration:
  - AES-128/192/256,
  - Hash (FIPS PUB 180-4),
  - HMAC,
  - RSA,
  - Digital signature,
  - random number generator (RNG).

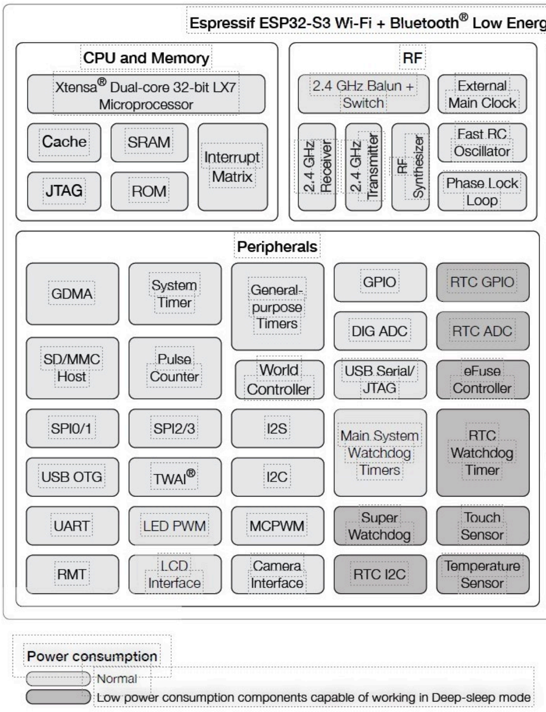


Figure 114: ESP32-S3 Functional block diagram

ESP32-S3-PICO-1 has all the functions of ESP32-S3 but integrates all peripheral components, including a crystal oscillator, decoupling capacitors, SPI flash/PSRAM, and RF matching links, within a single package. Figure 115 shows a functional block diagram of the ESP32-S3-PICO-1 chip.

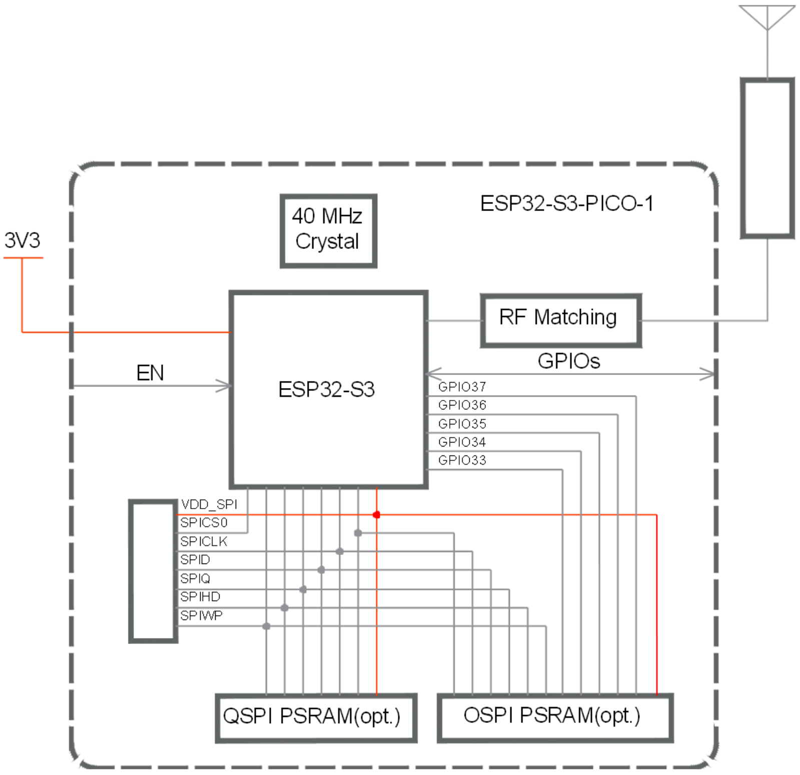


Figure 115: ESP32-S3-PICO-1 Functional block diagram

ESP32-S3 Modules

The company also produces ready-made modules[97][98][99]for easier implementation in user systems. These modules combine ESP32-S2 microcontroller, antenna and additional components mounted on PCB with EM shield [100] (table 18):

Table 18: Espressif ESP32-S3 modules

Module	Chip Embedded	Dimensions (mm)	Pins	Flash (MB)	PSRAM (MB)	Antenna	Development Board
ESP32-S3-WROOM-1 (figure 116)	ESP32-S3 ESP32-S3R2 \\ESP32-S3R8	18×25.5×3.1	41	4,8,16	0,2,8	PCB	ESP32-S3-DevKitC-1 ESP32-S3-DevKitC-1 ESP32-S3-BOX-3 ESP32-S3-BOX ESP32-S3-EYE ESP32-S3-Korvo-1 ESP32-S3-Korvo-2 ESP32-S3-LCD-Ev-Board
ESP32-S3-WROOM-1U	ESP32-S3 ESP32-S3R2 ESP32-S3R8	18×19.2×3.2	41	4,8,16	0,2,8	IPEX	ESP32-S3-DevKitC-1
ESP32-S3-WROOM-2 (figure 117)	ESP32-S3R8V	18×25.5×3.1	41	16,32	8	PCB	ESP32-S3-DevKitC-1
ESP32-S3-MINI-1 (figure	ESP32-S3FN8	15.4×15.4×2.4	65	8	N/A	PCB	ESP32-S3-DevKitM-1

Module	Chip Embedded	Dimensions (mm)	Pins	Flash (MB)	PSRAM (MB)	Antenna	Development Board
118)	ESP32-S3FH4R2						
ESP32-S3-MINI-1U	ESP32-S3FN8 ESP32-S3FH4R2	15.4×15.4×2.4	65	8	N/A	IPEX	ESP32-SM-DevKitM-1

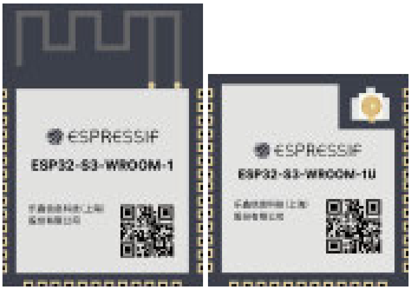


Figure 116: ESP32-S3-Wroom-1/1U

Figure 117: ESP32-S3-Wroom-2

Figure 118: ESP32-S3-Wroom-1/1U

ESP32-S3 Development Kits

To facilitate the use of ESP32-S3, Espressif and other companies produce different development kits to suit different needs and present different processor functions. The original Espressif best-known small development boards are:

- ESP32-S3-DevkitM,
- ESP32-S3-DevkitC,

- ESP32-S3-BOX-3,
- ESP32-S3-BOX,
- ESP32-S3-EYE,
- ESP32-S3-Korvo-1,
- ESP32-S3-Korvo-2,
- ESP32-S3-LCD-Ev-Board.

For this book, we present only a few of the most popular, universal for various applications development boards:

- ESP32-S3-DevkitM(figure 119) ,
- ESP32-S3-DevkitC(figure 120) ,
- Waveshare ESP32-PICO-1[101] (figure 121),
- M5Stamp-S3[102] (figure 122).

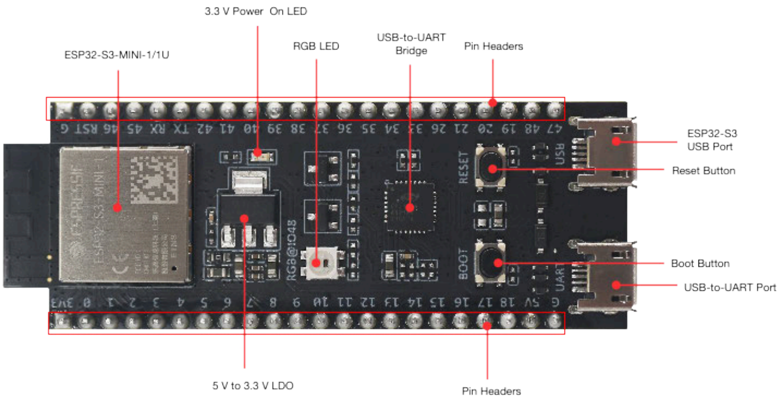


Figure 119: ESP32-S3-DevkitM

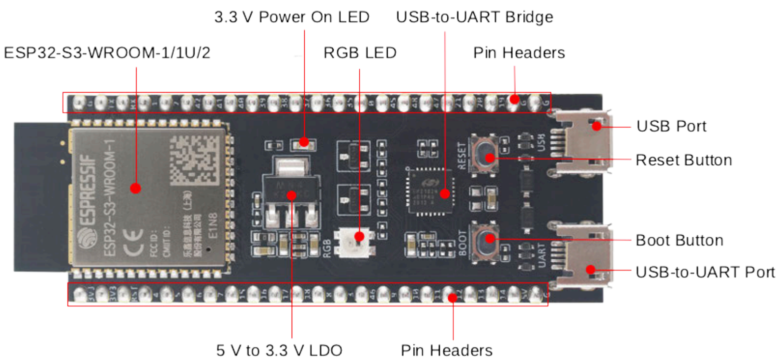


Figure 120: ESP32-S3-DevkitC

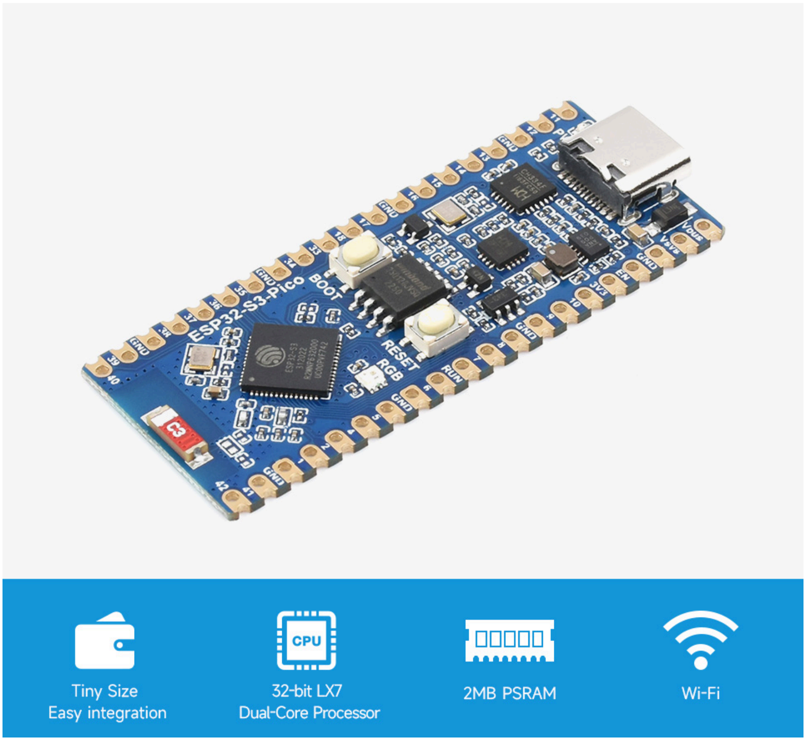


Figure 121: Waveshare ESP32-S3-PICO-1

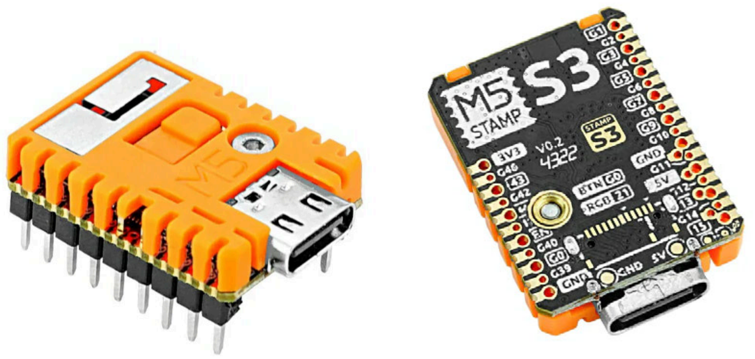


Figure 122: M5Stamp-S3 with pin headers

ESP32-S2&S3 chip comparison

Table 19 provides a brief comparison of the most essential features of the ESP32-S2 & ESP32-S3 systems[103]

Table 19: ESP32-S2 & ESP32-S3 family brief comparison

Feature	ESP32 Series	ESP32-S2 Series	ESP32-S3 Series
Launch year	2016	2020	2020
Core	Xtensa® dual-/single core 32-bit LX6	Xtensa® single-core 32-bit LX7	Xtensa® dual-core 32-bit LX7
Wi-Fi protocols	802.11 b/g/n, 2.4 GHz	802.11 b/g/n, 2.4 GHz	802.11 b/g/n, 2.4 GHz
Bluetooth®	Bluetooth v4.2 BR/EDR and Bluetooth Low Energy	✖	Bluetooth 5.0
Typical frequency	240 MHz (160 MHz for ESP32-S0WD)	240 MHz	240 MHz
SRAM	520 KB	320 KB	512 KB
ROM	448 KB for booting and core functions	128 KB for booting and core functions	384 KB for booting and core functions
Embedded flash	2 MB, 4 MB, or none, depending on variants	2 MB, 4 MB, or none, depending on variants	8 MB or none, depending on variants
External flash	Up to 16 MB device, address 11 MB + 248 KB each time	Up to 1 GB device, address 11.5 MB each time	Up to 1 GB device, address 32 MB each time
External RAM	Up to 8 MB device, address 4 MB each time	Up to 1 GB device, address 11.5 MB each time	Up to 1 GB device, address 32 MB each time
Cache	✔ Two-way set associative	✔ Four-way set associative, independent instruction cache and data cache	✔ Four-way or eight-way set associative for instruction cache; four-way set associative for data cache, 32-bit data/instruction bus width
Peripherals			
ADC	Two 12-bit, 18 channels	Two 12-bit, 20 channels	Two 12-bit SAR ADCs, 20 channels
DAC	Two 8-bit channels	Two 8-bit channels	✖
Timers	Four 64-bit general-purpose timers, and three watchdog timers	Four 64-bit general-purpose timers, and three watchdog timers	Four 54-bit general-purpose timers, and three watchdog timers
Temperature sensor	✖	1	1
Touch sensor	10	14	14

Feature	ESP32 Series	ESP32-S2 Series	ESP32-S3 Series
Hall sensor	1	✖	✖
GPIO	34	43	45
SPI	4	4	4
LCD interface	1	1	1
UART	3	2 [1]	3
I2C	2	2	2
I2S	2, can be configured to operate with 8/16/32/40/48-bit resolution as an input or output channel.	1, can be configured to operate with 8/16/24/32/48/64-bit resolution as an input or output channel.	2, can be configured to operate with 8/16/24/32-bit resolution as an input or output channel.
Camera interface	1	1	1
DMA	Dedicated DMA to UART, SPI, I2S, SDIO slave, SD/MMC host, EMAC, BT, and Wi-Fi	Dedicated DMA to UART, SPI, AES, SHA, I2S, and ADC Controller	General-purpose, 5 TX channels, 5 RX channels
RMT	8 channels	4 channels [1], can be configured to TX/RX channels	8 channels [2], 4 TX channels, 4 RX channels
Pulse counter	8 channels	4 channels [1]	4 channels [1]
LED PWM	16 channels	8 channels [1]	8 channels [1]
MCPWM	2, six PWM outputs	✖	2, six PWM outputs
USB OTG	✖	1	1
TWAI® controller (compatible with ISO 11898-1)	1	1	1
SD/SDIO/MMC host controller	1	✖	1
SDIO slave controller	1	✖	✖
Ethernet MAC	1	✖	✖
ULP	ULP FSM	PicoRV32 core with 8 KB SRAM, ULP FSM	PicoRV32 core with 8 KB SRAM, ULP FSM
Debug Assist	✖	✖	✖
Security			
Secure boot	✔	✔ Faster and safer, compared with ESP32	✔ Faster and safer, compared with ESP32
Flash encryption	✔	✔ Support for PSRAM encryption. Safer, compared with ESP32	✔ Support for PSRAM encryption. Safer, compared with ESP32
OTP	1024-bit	4096-bit	4096-bit
AES	✔ AES-128, AES-192, AES-256 (FIPS PUB 197)	✔ AES-128, AES-192, AES-256 (FIPS PUB 197); DMA support	✔ AES-128, AES-256 (FIPS PUB 197); DMA support
HASH	SHA-1, SHA-256, SHA-384, SHA-512 (FIPS PUB 180-4)	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256, SHA-512/t (FIPS PUB 180-4); DMA support	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256, SHA-512/t (FIPS PUB 180-4); DMA support
RSA	Up to 4096 bits	Up to 4096 bits	Up to 4096 bits
RNG	✔	✔	✔
HMAC	✖	✔	✔
Digital signature	✖	✔	✔
XTS	✖	✔ XTS-AES-128, XTS-AES-256	✔ XTS-AES-128, XTS-AES-256

Feature	ESP32 Series	ESP32-S2 Series	ESP32-S3 Series
Other			
Deep-sleep (ULP sensor-monitored pattern)	100 µA (when ADC work with a duty cycle of 1%)	22 µA (when touch sensors work with a duty cycle of 1%)	TBD
Size	QFN48 5*5, 6*6, depending on variants	QFN56 7*7	QFN56 7*7

1. **Note** 1: Reduced chip area compared with ESP32
2. **Note** 2: Reduced chip area compared with ESP32 and ESP32-S2
3. **Note** 3: Die size: ESP32-S2 < ESP32-S3 < ESP32

ESP32-Cx Family

ESP32-C2 General Information

The ESP32-C2 (ESP8684) [104] family is a series of microcontrollers developed by Espressif Systems. It's based on the RISC-V architecture and is designed to offer ultra-low power and small size for various IoT (Internet of Things) applications. This family of microcontrollers has been designed to target simple, high-volume, and low-data-rate IoT applications, such as smart plugs and smart light bulbs. ESP32-C2 is also supported by Espressif's AIoT Private Cloud platform, ESP RainMaker® and supports Matter, a smart-home connectivity protocol that runs on any IP-supporting network stack.

The ESP32-C2 microcontrollers come with several distinctive features:

- RISC-V Core: The ESP32-C2 is based on the RISC-V architecture, an open-source instruction set architecture (ISA). This differs from the ESP32 series' usual Tensilica Xtensa LX6 architecture.
- Connectivity: Like other ESP32 modules, the ESP32-C2 features built-in Wi-Fi and Bluetooth 5 LE connectivity. This allows it to connect to the internet and communicate with other devices over short distances.
- Low Power Consumption: ESP32-C2, like other ESP32 variants, supports low-power modes, which is crucial for battery-powered and energy-efficient IoT applications.
- Rich Peripheral Interface Support: It includes a variety of peripherals such as UART, I2C, SPI, ADC, and more, making it versatile for different applications.
- Security Features: The ESP32-C2 family includes various security features, such as secure boot, flash encryption, secure storage, and cryptographic accelerators.
- Compact Form Factor: The ESP32-C2 family is designed in a very compact form factor (4mm x 4mm), which is crucial for applications with limited space or miniaturization.
- Cost-Effective Solution: These microcontrollers offer a cost-effective solution for IoT applications without compromising essential features and performance.

For now the ESP32-C2 family includes the following chips in mass production (figure 123):

- ESP8684.

Figure 123: ESP32-C2

ESP32-C2

ESP32-C2 Architecture Overview

Figure 124[105] shows functional block diagram of ESP32-C2 chip. The main common features of the ESP32-C2 are:

Processors

- **Main processor:** 32-bit RISC-V single-core CPU,
  - **Cores**: 1 up to 120 MHz,
  - External main crystal clock,
  - External 32 kHz crystal oscillator for RTC or internal RC.

Wireless connectivity

- **WiFi:** 802.11 b/g/n (802.11n @ 2.4 GHz up to 72.2 Mbit/s) with simultaneous Infrastructure BSS Station mode/SoftApp mode/Promiscuous mode,
- **Bluetooth:** v5.0 Bluetooth Low Energy (BLE) (speed: 125 Kbps - 2 Mbps) with multiple advertisement sets

Memory: Internal memory

- **Embedded flash** 1, 2, 4 MB,
- **ROM:** 576 kB (for booting and core functions),
- **SRAM:** 272 kB (16kB for cache),
- **eFuse** - 1 Kbit -256 bits reserved for encryption key and device ID.

Peripheral Input/Output

- 14 x GPIO,
- 3 × SPI (Serial Peripheral Interface),
- 2 x UART (universal asynchronous receiver/transmitter),
- 1 × I²C Master (Inter-Integrated Circuit),
- LED PWM up to 6 channels,
- 1 x 12-bit ADCs (analogue-to-digital converter) up to 5 channels,
- General DMA controller (GDMA), with 1 transmit channel and 1 receive channel.

Power Modes

- Active Mode,
- Modem-sleep mode,
- Light-sleep mode,
- Deep-sleep mode.

Security

- Secure boot,
- Flash encryption,
- 1024-bit OTP, up to 256-bit for customers,
- Cryptographic hardware acceleration:
  - SHA1/SHA224/SHA256 (FIPS PUB 180-4),
  - ECC,
  - random number generator (RNG),
  - clock glitch filter.

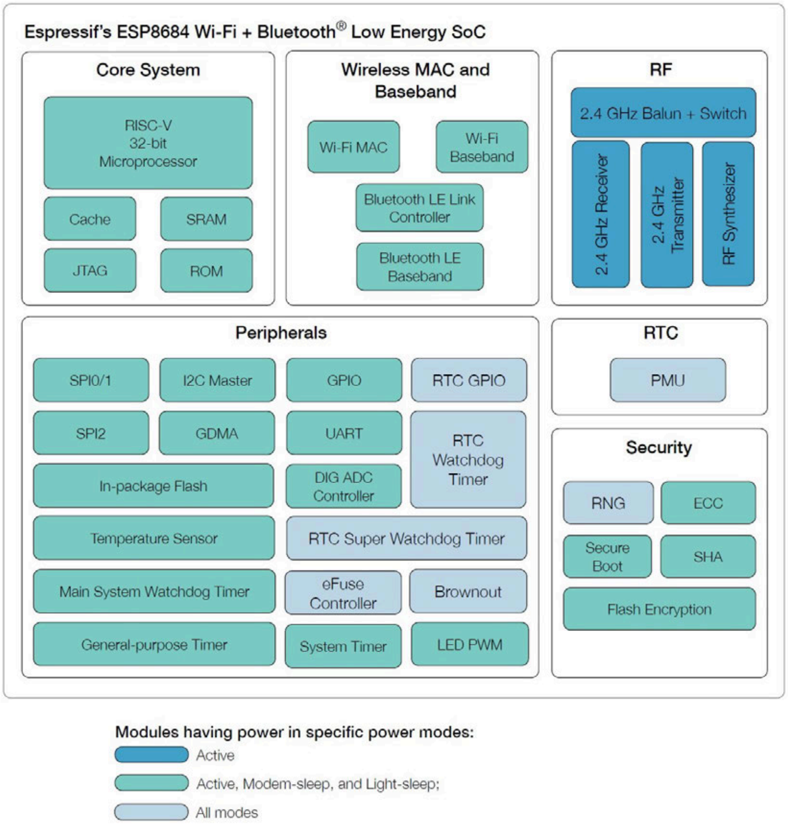


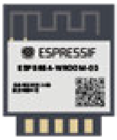
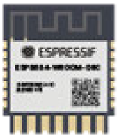
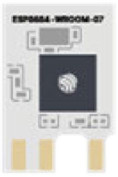
Figure 124: ESP32-C2 functional block diagram

For now the ESP32-C2 family includes the following chips in mass production (table 20):

Table 20: Espressif ESP32-C2 chips

Module	Chip Embedded	Dimensions (mm)	Pins	GPIO	Flash (MB)	PSRAM (MB)	Antenna type	Development Board
ESP8684-MINI-1	ESP8684H2 ESP8684H4	13.2×16.6×2.4	53	14	1, 2, 4	N/A	PCB	ESP8684-DevKitM-1

Module	Chip Embedded	Dimensions (mm)	Pins	GPIO	Flash (MB)	PSRAM (MB)	Antenna type	Development Board
ESP8684-MINI-1U	ESP8684H2 ESP8684H4	13.2×12.5×2.4	53	14	1, 2, 4	N/A	IPEX	ESP8684-DevKitM-1
ESP8684-WROOM-01C	ESP8684H2 ESP8684H4	24×16×3.1	22	14	2, 4	N/A	PCB	N/A
ESP8684-WROOM-02C	ESP8684H2 ESP8684H4	18x20x3.2	18	14	2, 4	N/A	PCB	N/A
ESP8684-WROOM-02UC	ESP8684H2 ESP8684H4	18x20x3.2	18	14	2, 4	N/A	IPEX	ESP8684-DevKitC-02

Module	Chip Embedded	Dimensions (mm)	Pins	GPIO	Flash (MB)	PSRAM (MB)	Antenna type	Development Board
 ESP8684-WROOM-03	ESP8684H2 ESP8684H4	15×17.3×2.8	11	8	2, 4	N/A	PCB	N/A
ESP8684-WROOM-04C	ESP8684H2 ESP8684H4	24×16×3.1	17	13	2, 4	N/A	PCB	N/A
ESP8684-WROOM-05	ESP8684H2 ESP8684H4	15×17.3×2.8	7	5	2, 4	N/A	PCB	N/A
 ESP8684-WROOM-06C	ESP8684H2 ESP8684H4	15.8×20.3×2.7	21	14 or 5	2, 4	N/A	PCB	N/A
 ESP8684-WROOM-07	ESP8684H2 ESP8684H4	8.5×12.7×1.9	6	3	2, 4	N/A	Solder pad for external monopole antenna	N/A

1. **Note** 1: When surface mounted, the module has 14 available GPIOs; when vertically soldered, the module has 5 available GPIOs.

ESP32-C3

ESP32-C3 General Information

The ESP32-C3 family is a series of microcontrollers developed by Espressif Systems. It's based on the RISC-V architecture and is designed to offer low-power and cost-effective solutions for various IoT (Internet of Things) applications. These chips integrate WiFi connectivity, have low power consumption, and offer different peripheral interfaces. They suit diverse IoT projects, enabling developers to create connected devices efficiently. The new ESP32-C3 family is known for its compact size, low power consumption, and integration of WiFi capabilities. These microcontrollers balance performance and power efficiency, making them suitable for battery-powered IoT devices. They support a variety of interfaces like SPI, I2C, UART, and ADC, enabling connectivity and interactions with various sensors and devices. This family of microcontrollers is viral in smart home devices, wearables, and other IoT applications that require wireless connectivity.

The ESP32-C3 microcontrollers come with several distinctive features:

- RISC-V Core: One of the notable aspects of the ESP32-C3 family is the use of the RISC-V instruction set architecture, which provides efficiency and flexibility. This architecture allows for customization and optimization, balancing performance and power consumption.
- WiFi Connectivity: These chips integrate WiFi connectivity, enabling devices to connect to wireless networks, making them ideal for IoT applications that require internet connectivity.
- Low Power Consumption: The ESP32-C3 family is designed to focus on low power consumption, which is essential for battery-powered or energy-efficient IoT devices. This makes them suitable for applications where power efficiency is a priority.
- Rich Peripheral Interface Support: The microcontrollers have various peripheral interfaces, such as SPI, I2C, UART, PWM, and ADC. These interfaces allow easy integration with multiple sensors, displays, and other devices, enhancing the versatility of applications that can be developed.
- Security Features: The ESP32-C3 family includes various security features like secure boot, flash encryption, secure storage, and cryptographic accelerators. These elements contribute to the overall security of the devices developed using these microcontrollers.
- Compact Form Factor: The ESP32-C3 family is designed in a compact form factor, which is advantageous for applications where limited space or miniaturization is a concern.
- Cost-Effective Solution: These microcontrollers offer a cost-effective solution for IoT applications without compromising essential features and performance.

For now the ESP32-C3 family includes the following chips in mass production (table 21):

Table 21: Espressif ESP32-C3 chips

SoC	Variants	Core	Dimensions (mm)	Pins	RAM (kB)	Flash (MB)	PSRAM (MB)
ESP32-C3(figure 125)	ESP32-C3 ESP32-C3FH4 ESP32-C3FH4X	Single Core	QFN 5×5	32	400 KB RAM, 384 KB ROM, 8 KB RTC SRAM	4	N/A

SoC	Variants	Core	Dimensions (mm)	Pins	RAM (kB)	Flash (MB)	PSRAM (MB)
ESP8686(figure 126)	ESP8686H4	Single Core	QFN 4×4	24	400 KB RAM, 384 KB ROM, 8 KB RTC SRAM	4	N/A
ESP8685(figure 127)	ESP8685H2 ESP8685H4	Single Core	QFN 4×4	28	400 KB RAM, 384 KB ROM, 8 KB RTC SRAM	2, 4	N/A

Figure 125: ESP32-C3

Figure 126: ESP8686

Figure 127: ESP8685

ESP32-C3 Architecture Overview

Figure 128 shows a functional block diagram of the ESP32-C3 chip. Main common features of the ESP32-C3 are: [106]

Processors

- **Main processor:** 32-bit RISC-V single-core CPU,
  - **Cores**: 1 up to 160 MHz,
  - External main crystal clock,
  - External 32 kHz crystal oscillator for RTC or internal RC.

Wireless connectivity

- **WiFi:** 802.11 b/g/n (802.11n @ 2.4 GHz up to 150 Mbit/s) with simultaneous Infrastructure BSS Station mode/SoftApp mode/Promiscuous mode.
- **Bluetooth:** v5.0 Bluetooth Low Energy (BLE) (speed: 125 Kbps - 2 Mbps) with multiple advertisement sets

Memory: Internal memory

- **Embedded flash** 4 MB

- **ROM:** 384 kB (for booting and core functions).
- **SRAM:** 400 kB (16kB for cache).
- **RTC fast SRAM:** 8 kB
- **eFuse** - 4 Kbit - 1792 bits reserved for encryption key and device ID

Peripheral Input/Output

- 22 or 16 GPIO
- 2 x 12-bit ADCs (analog-to-digital converter) up to 6 channels,
- General DMA controller (GDMA), with 3 transmit channels and 3 receive channels,
- 1 × I^2C (Inter-Integrated Circuit),
- 2 x UART (universal asynchronous receiver/transmitter),
- 1 × TWAI® controller compatible with ISO 11898-1 (CAN Specification 2.0),
- 3 × SPI (Serial Peripheral Interface),
- 1 × I^2S (Integrated Inter-IC Sound),
- LED PWM up to 6 channels,
- Internal temperature sensor,
- USB Serial/JTAG controller.

Security

- Secure boot,
- Flash encryption,
- 4096-bit OTP, up to 1792-bit for customers,
- Cryptographic hardware acceleration:
  - AES-128/256,
  - SHA accelerator,
  - RSA accelerator,
  - random number generator (RNG),
  - digital signature.

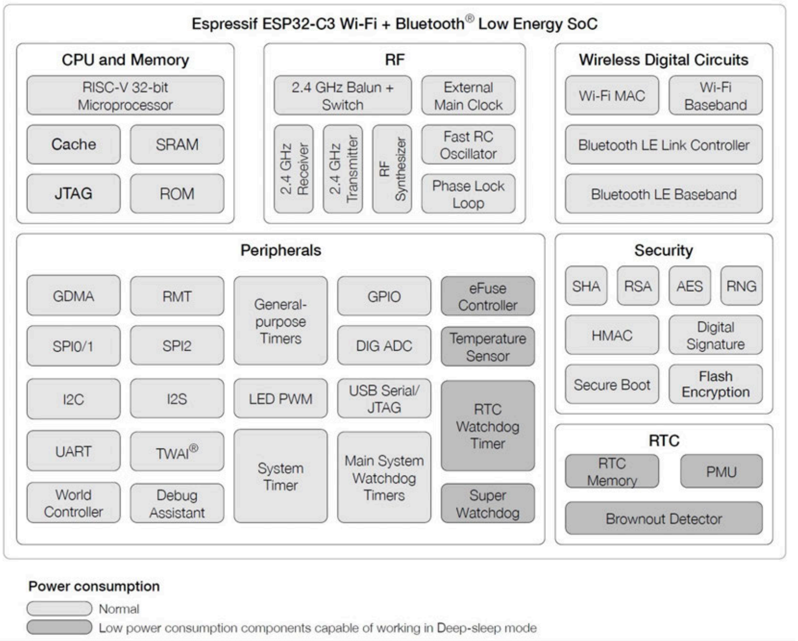


Figure 128: ESP32-C3 functional block diagram

ESP32-C3 Modules

Espressif also produces modules that are more integrative and more convenient for amateurs and developers to use. The following modules are currently available:

- ESP32-C3-Mini-1/1U[107](figure 129) ,
- ESP32-C3-WROOM-02/02U[108](figure 130).

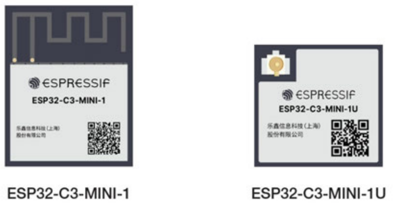


Figure 129: ESP32-C3-Mini-1/1U

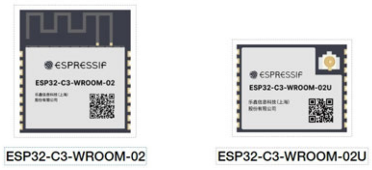


Figure 130: ESP32-C3-Wroom-02/02U

ESP32-C3 Development Kits

Development kits are the most convenient for quick application or to check the capabilities of processors. Espressif manufactures them and many companies specialising in producing prototype circuits. The following are some of the most versatile modules

- Espressif - ESP32-C3-DevkitM-1[109](figure 131),
- Espressif - ESP32-C3-DevkitC-02[110](figure 132),
- Espressif - ESP32-C3-LCDKit [111](figure 133)
- Adafruit - QT Py ESP32-C3 WiFi Dev Board with STEMMA QT[112] (figure 134),
- Seeed Studio - XIAO ESP32C3 [113] (figure 135),
- M5stack - M5Stamp-C3 [114] (figure 136).

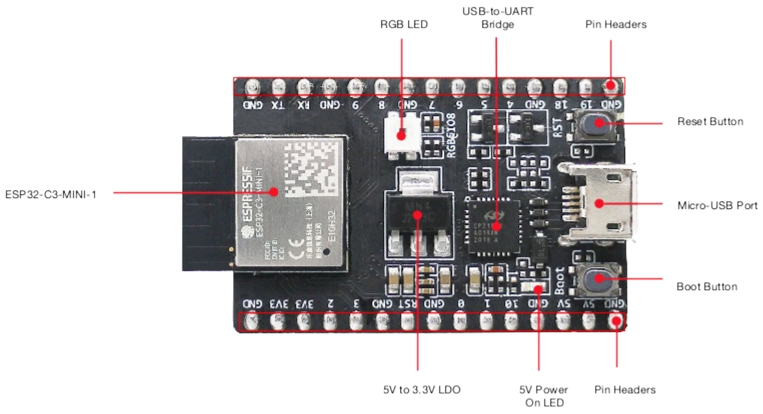


Figure 131: Espressif - ESP32-C3-DevkitM-1

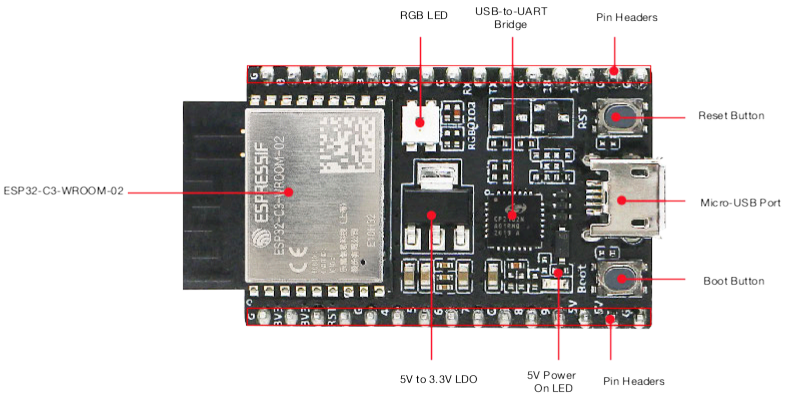


Figure 132: Espressif - ESP32-C3-DevkitC-02

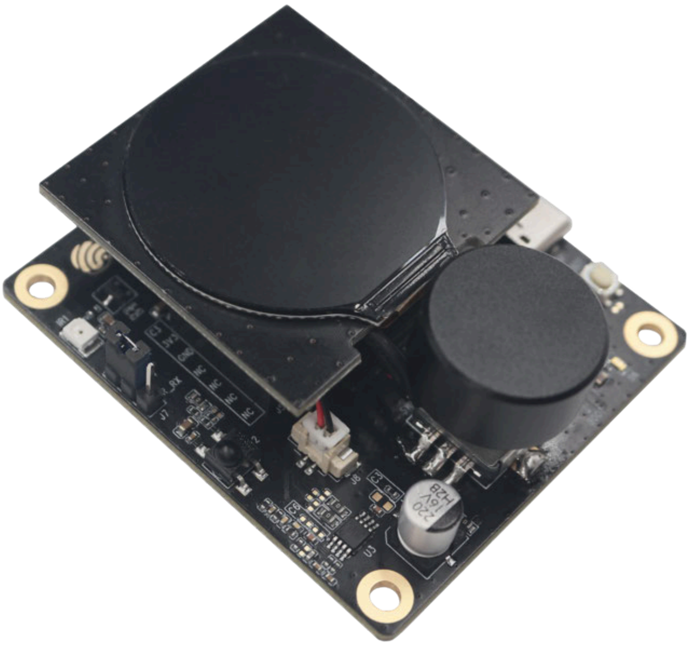
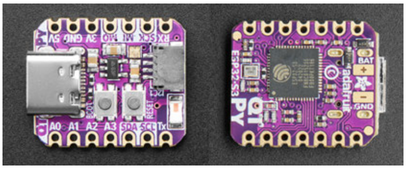


Figure 133: Espressif - ESP32-C3-LCDKit



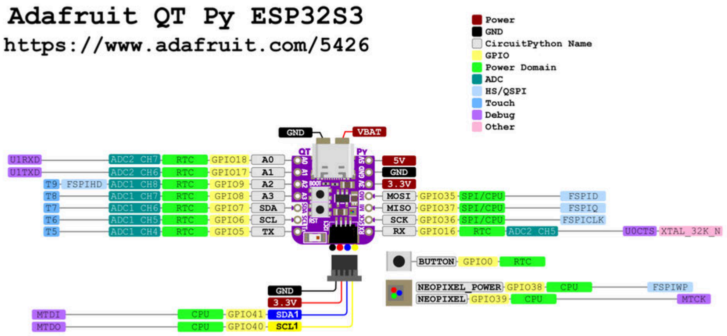
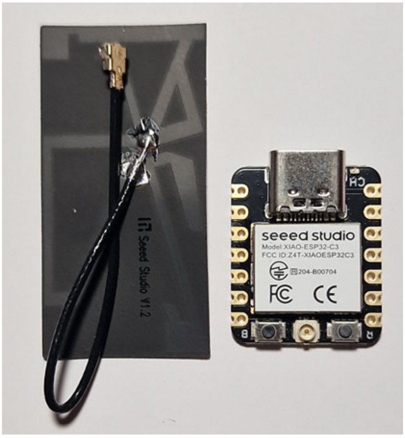

Adafruit QT Py ESP32S3
https://www.adafruit.com/5426
Power
GND
CircuitPython Name
GPIO
Power Domain
ADC
HS/QSPI
Touch
Debug
Other
GND
VBAT
U1RXD ADC2 CH7 RTC GPIO18 A0
U1TXD ADC2 CH6 RTC GPIO17 A1
T9 FSPIHD ADC1 CH8 RTC GPIO9 A2
T8 ADC1 CH7 RTC GPIO8 A3
T7 ADC1 CH6 RTC GPIO7 SDA
T6 ADC1 CH5 RTC GPIO6 SCL
T5 ADC1 CH4 RTC GPIO5 TX
5V
GND
3.3V
MOSI GPIO35 SPI/CPU FSPID
MISO GPIO37 SPI/CPU FSPIQ
SCK GPIO36 SPI/CPU FSPICLK
RX GPIO16 RTC ADC2 CH5 U0CTS XTAL_32K_N
BUTTON GPIO0 RTC
NEOPIXEL_POWER GPIO38 CPU FSPIWP
NEOPIXEL GPIO39 CPU MTCK
GND
3.3V
MTDI CPU GPIO41 SDA1
MTDO CPU GPIO40 SCL1

Figure 134: Adafruit - QT Py ESP32-C3



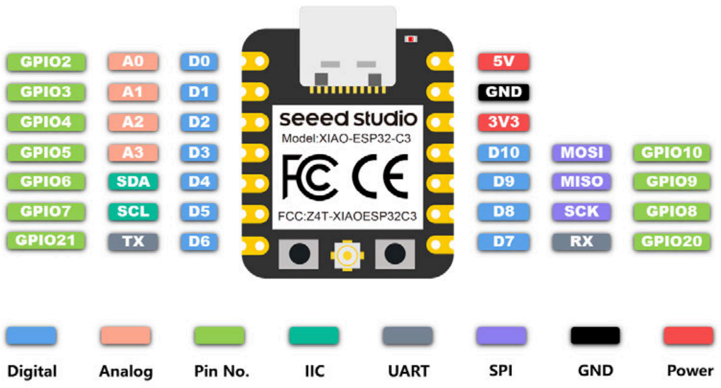


Figure 135: Seeed Studio - XIAO ESP32C3

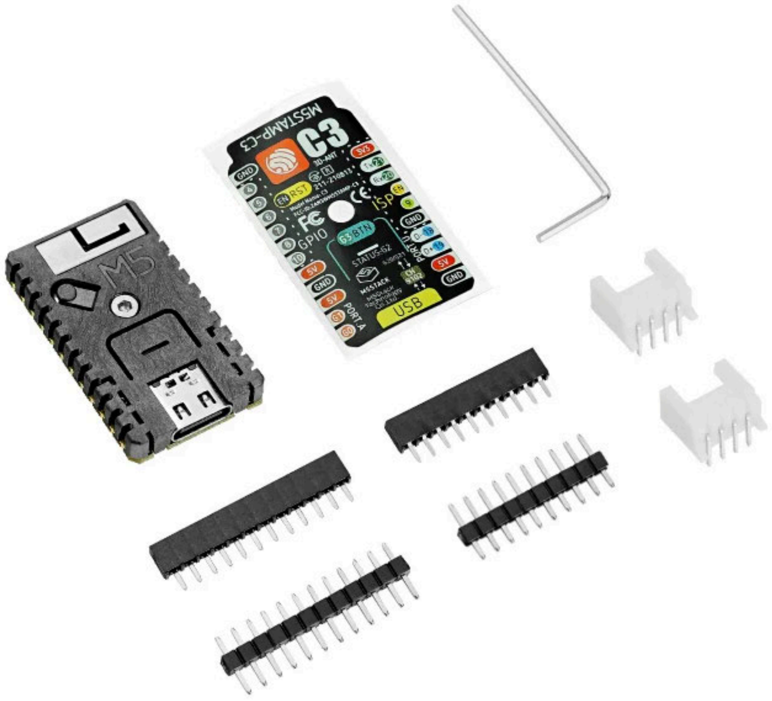


Figure 136: M5Stamp-C3

A M5Stamp-C3u version with built-in JTAG interface is also available

ESP32-C3 chip comparison

The Esp32-C3 as a more modern one, can successfully replace the oldest family of ESP8266 chips, so table 22 provides a brief comparison of the essential features of the ESP8266 & ESP32-S3 systems [115].

Table 22: Esp8266 & ESP32-C3 family brief comparison

Feature	ESP8266	ESP32-C3 Series
Launch year	2014	2020
Core	Xtensa® single core 32-bit LX6	32-bit single-core RISC-V
Wi-Fi protocols	802.11 b/g/, 2.4 GHz up to 72.2. Mbps	802.11 b/g/n, 2.4 GHz up to 150 Mbps
Bluetooth®	✖	Bluetooth 5.0
Typical frequency	80 MHz	160 MHz
SRAM	160kB	400 KB
ROM	384 KB	384 KB for booting and core functions
Embedded flash	✖	4 MB or none, depending on variants
RTC memory	768B	8kB
Cache	32KB instruction	16kB
PMU	✔	✔
Peripherals		
ADC	10-bit	Two 12-bit SAR ADCs, at most 6 channels
DAC	✖	✖
Timers	2 x 23 - bit	Two 54-bit general-purpose timers, and three watchdog timers
Temperature sensor	1	1
Touch sensor	✖	✖
Hall sensor	✖	✖
GPIO	17	22
SPI	2	3
LCD interface	✖	✖
UART	2 – One Tx only	2
I2C	1- only software	1
I2S	1	1, can be configured to operate with 8/16/24/32-bit resolution as an input or output channel.
Camera interface	✖	✖
DMA	✖	General-purpose, 3 TX channels, 3 RX channels
RMT	1 x TX + 1 x RX	4 channels [2], 2 TX channels, 2 RX channels
Pulse counter	✖	✖

Feature	ESP8266	ESP32-C3 Series
LED PWM	5 channels	6 channels
PWM	✖/software 8 ch	✖
TWAI® controller (compatible with ISO 11898-1)	✖	1
SD/SDIO/MMC host controller	✖	✖
SDIO slave controller	✖	✖
Ethernet MAC	✖	✖
Debug Assist JTAG	✖	1
Security		
Secure boot	✖	✔ Faster and safer, compared with ESP32,
Flash encryption	✖	✔ Safer, compared with ESP32, XTS-AES-128
OTP	1024-bit	4096-bit
AES	✖	✔ AES-128, AES-256 (FIPS PUB 197); DMA support
HASH	SHA-1, SHA-256, SHA-384, SHA-512 (FIPS PUB 180-4)	SHA-1, SHA-224, SHA-256 (FIPS PUB 180-4); DMA support
RSA	Up to 4096 bits	Up to 3072 bits
RNG	✔	✔
HMAC	✖	✔
Digital signature	✖	✔
XTS	✖	✔ XTS-AES-128
Other		
Light sleep	2 mA	130µA
Deep Sleep	20 µA	5 µA
Hibernation	-	-
Power off	0.5 µA	1µA
Size	QFN32 5*5	QFN32 5*5

ESP32-C6

ESP32-C6 General Information

ESP32-C6 is Espressif's first WiFi 6 SoC integrating 2.4 GHz WiFi 6, Bluetooth 5.3 (Low Energy) and the 802.15.4 protocol. It is based on a high-performance (HP) 32-bit RISC-V processor, which can be clocked up to 160 MHz, and also has a low-power (LP) 32-bit RISC-V processor, which can be clocked up to 20 MHz. It has a 320KB ROM, a 512KB SRAM and works with external flash. The ESP32-C6, with its support for WiFi 6 and Bluetooth 5.3, can be a potential candidate for devices seeking to integrate into the Matter standard. Matter intends to create a universal standard for smart home devices to ensure interoperability and ease of use across different brands and ecosystems. Devices equipped with the ESP32-C6 can potentially comply with the Matter standard to ensure compatibility with other Matter-certified devices. They can be used to develop various other Matter-ecosystem solutions, such as Matter Gateways, Thread Border Routers or Zigbee Matter Bridges. However, adherence to the Matter standard involves hardware and software considerations, and manufacturers must ensure their devices meet the required protocols for certification.

ESP32-C6 Architecture Overview

Figure 137 shows a functional block diagram of the ESP32 chip. Main common features of the ESP32-C6 are: [116]

Processors

- **Main processor:** 32-bit RISC-V single-core CPU up to 160MHz,
  - **Cores**: 1,
  - External main crystal clock,
  - External 32 kHz crystal oscillator for RTC or internal RC.

- **Low-power processor:** up to 20MHz
  - **Cores**: 1,
  - External main crystal clock,
  - External 32 kHz crystal oscillator for RTC or internal RC.

Wireless connectivity

- **WiFi:**(802.11ax 20MHz only non-AP mode),
- **WiFI:**(802.11b/g/n @ 2.4 GHz up to 150 Mbit/s) with simultaneous Infrastructure BSS Station mode/SoftApp mode/Promiscuous mode,
- **Bluetooth:** v5.3 Bluetooth Low Energy (BLE) (speed: 125 Kbps - 2 Mbps) with multiple advertisement sets,
- **IEEE 802.15.4-2015:** up to 250 kbps; Thread 1.3; ZigBee 3.0.

Memory: Internal memory

- **Embedded flash** 4 MB,
- **ROM:** 320 kB (booting and core functions),
- **HP SRAM:** 510 kB,
- **LP SRAM:** 16 kB,
- **RTC fast SRAM:** 8 kB,
- **eFuse** - 4 Kbit - 1792 bits reserved for encryption key and device ID.

Peripheral Input/Output

- 30xGPIO (QFN40) or 22xGPIO (QFN32),
- General DMA controller (GDMA), with 3 transmit channels and 3 receive channels,
- 1 × I^{2}C (Inter-Integrated Circuit),
- 2 x UART,
- 1 x Low-Power UART,
- 2 × TWAI® controller compatible with ISO 11898-1 (CAN Specification 2.0),
- 2 × SPI (Serial Peripheral Interface for flash),

- 1 × SPI (Serial Peripheral Interface universal),
- 1 × I^{2}S (Integrated Inter-IC Sound),
- 1 × SDIO 2.0 slave controller,
- 1 × Motor Control PWM (MCPWM),
- LED PWM up to 6 channels,
- 1 x USB Serial/JTAG controller,
- 1 x Remote control peripheral (TX/RX),
- 1 x Parallel IO interface (PARLIO),
- 1 x 12-bit SAR ADCs (analog-to-digital converter) up to 7 channels,
- 1 x temperature sensor.

Security

- Secure boot,
- Flash encryption,
- External Memory Encryption and Decryption (XTS_AES),
- 4096-bit OTP, up to 1792-bit for customers,
- Trusted execution environment (TEE) controller and access permission management (APM),
- Cryptographic hardware acceleration:
  - AES-128/256,
  - ECC,
  - SHA accelerator,
  - RSA accelerator,
  - HASH (FIPS PUB 180-4),
  - random number generator (RNG),
  - digital signature.

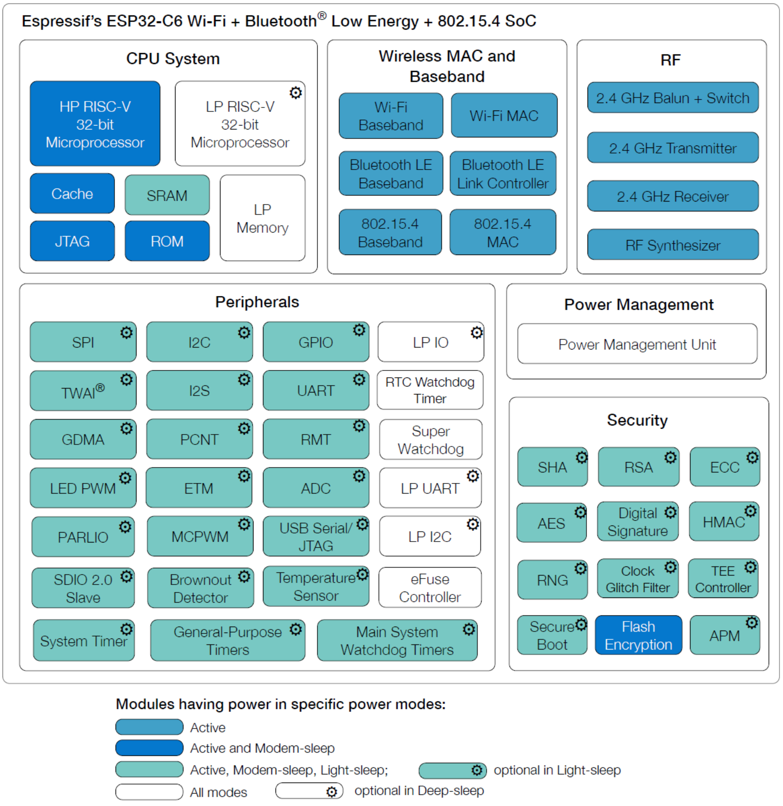


Figure 137: ESP32-C6 functional block diagram

ESP32-C6 Modules

The following modules are currently available (table 23):

Table 23: Espressif ESP32-C6 modules

Module	Chip embedded	Dimensions (mm)	Pins	Development board
ESP32-C6-Mini-1/1U (figure 138)	ESP32-C6FH4	13.2×16.6×2.4 13.2×12.5×2.4	53	ESP32-C6-DevKitM-1
ESP32-C6-WROOM-02/02U (figure 139)	ESP32-C6	18×25.5×3.2 18×19.2×3.2	28	ESP32-C6-DevKitC-1

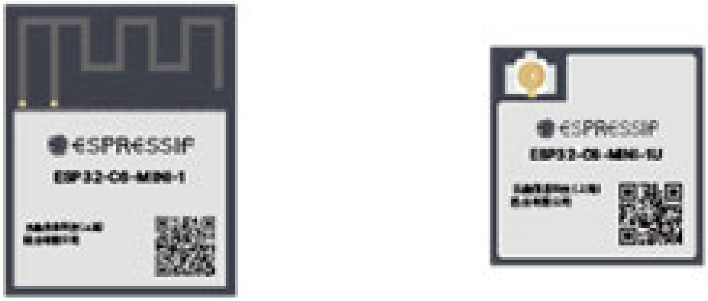


Figure 138: ESP32-C6-Mini-1/1U

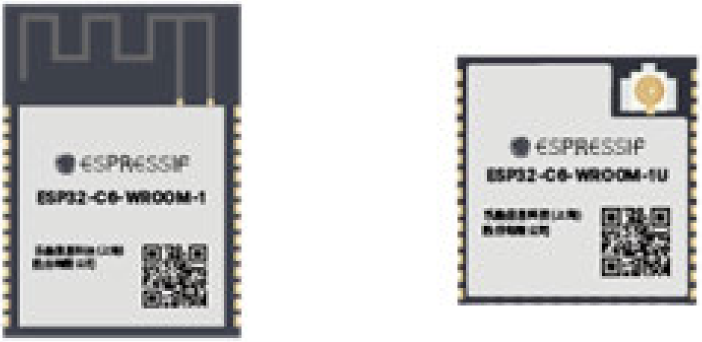
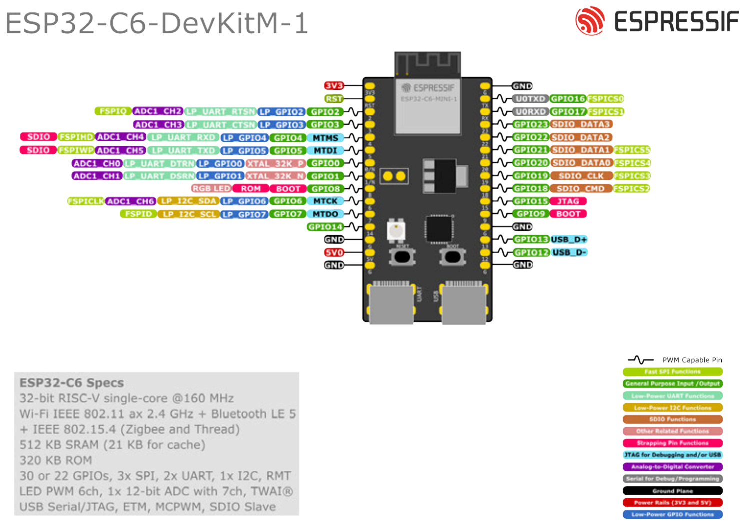


Figure 139: ESP32-C6-Wroom-02/02U

ESP32-C6 Development Boards

There are not many prototype kits with ESP32-C6 SOCs on the market yet. Two sets released by the manufacturer deserve special attention. They are both entry-level development boards:

- Espressif - ESP32-C6-DevkitM-1 [117](figure 140),
- Espressif - ESP32-C6-DevkitC-1[118](figure 141) .


ESP32-C6-DevKitM-1
ESPRESSIF
3V3
RST
FSPIQ ADC1_CH2 LP_UART_RTSN LP_GPIO2 GPIO2
ADC1_CH3 LP_UART_CTSN LP_GPIO3 GPIO3
SDIO FSPIHD ADC1_CH4 LP_UART_RXD LP_GPIO4 GPIO4 MTMS
SDIO FSPIWP ADC1_CH5 LP_UART_TXD LP_GPIO5 GPIO5 MTDI
ADC1_CH0 LP_UART_DTRN LP_GPIO0 XTAL_32K_P GPIO0
ADC1_CH1 LP_UART_DSRN LP_GPIO1 XTAL_32K_N GPIO1
RGB LED ROM BOOT GPIO8
FSPICLK ADC1_CH6 LP_I2C_SDA LP_GPIO6 GPIO6 MTCK
FSPID LP_I2C_SCL LP_GPIO7 GPIO7 MTDO
GPIO14
GND
5V0
GND
GND
U0TXD GPIO16 FSPICS0
U0RXD GPIO17 FSPICS1
GPIO23 SDIO_DATA3
GPIO22 SDIO_DATA2
GPIO21 SDIO_DATA1 FSPICS5
GPIO20 SDIO_DATA0 FSPICS4
GPIO19 SDIO_CLK FSPICS3
GPIO18 SDIO_CMD FSPICS2
GPIO15 JTAG
GPIO9 BOOT
GND
GPIO13 USB_D+
GPIO12 USB_D-
GND
PWM Capable Pin
Fast SPI Functions
General Purpose Input /Output
Low-Power UART Functions
Low-Power I2C Functions
SDIO Functions
Other Related Functions
Strapping Pin Functions
JTAG for Debugging and/or USB
Analog-to-Digital Converter
Serial for Debug/Programming
Ground Plane
Power Rails (3V3 and 5V)
Low-Power GPIO Functions

ESP32-C6 Specs
32-bit RISC-V single-core @160 MHz
Wi-Fi IEEE 802.11 ax 2.4 GHz + Bluetooth LE 5
+ IEEE 802.15.4 (Zigbee and Thread)
512 KB SRAM (21 KB for cache)
320 KB ROM
30 or 22 GPIOs, 3x SPI, 2x UART, 1x I2C, RMT
LED PWM 6ch, 1x 12-bit ADC with 7ch, TWAI®
USB Serial/JTAG, ETM, MCPWM, SDIO Slave

Figure 140: Espressif ESP32-C6-DEVKITM-1

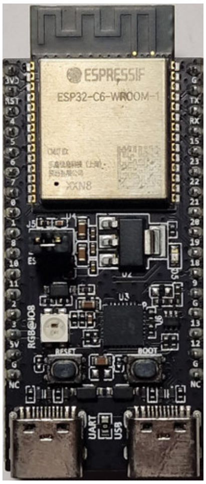


Figure 141: Espressif ESP32-C6-DEVKITC-1

They allow you to test all processor functions, including WiFi, Bluetooth LE, Zigbee, and Thread.

When purchasing the ESP32-C6 development board, please note that they may contain pre-production versions of ESP32-C6 that may not have full functionality implemented.

ESP32-Hx Family

ESP32-H2

ESP32-H2 General Information

ESP32-H2 is a family of microcontrollers (SoC) that combines IEEE 802.15.4 connectivity with Bluetooth 5 (LE). The system does not have a Wi-Fi protocol, but Thread and Zigbee protocols are available. ESP32-H2 has been certified as a “Zigbee-Compliant Platform” and has officially become a “Thread-Certified 1.3.0 Component”. The SoC is powered by a single-core, 32-bit RISC-V microcontroller that can be clocked up to 96 MHz. The ESP32-H2 has been designed especially for connected devices with low power consumption and security in mind. ESP32-H2 has 320 KB of SRAM with 16 KB of Cache, 128 KB of ROM, 4 KB LP of memory, and a built-in 2 MB or 4 MB SiP flash. It has 19 programmable GPIOs supporting ADC, SPI, UART, I2C, I2S, RMT, GDMA and LED PWM. For now, the ESP32-H2 family documentation is available as preliminary information only.

ESP32-H2 Architecture Overview

Figure 142 shows a functional block diagram of the ESP32-H2 chip. Main common features of the ESP32-H2 are [119]:

Processors

- **Main processor:** 32-bit RISC-V single-core CPU up to 96MHz,
  - **Cores**: 1

Wireless connectivity

- **Bluetooth:** v5.0 Bluetooth Low Energy (BLE) (speed: 125 Kbps - 2 Mbps),
- **802.15.4-2015:** up to 250 kbps; stacks include Thread 1.3, Zigbee 3.0, Matter, HomeKit, MQTT.

Memory: Internal memory

- **Embedded flash** 2 or 4 MB,
- **ROM:** 128 kB (for booting and core functions),
- **SRAM:** 320 kB (for data and instructions),

- **LP memory:** 4 KB of SRAM that can be accessed by the CPU. It can retain data in deep sleep mode,
- **eFuse** - 4 Kbit: 1792 bits are reserved for user data, such as encryption key and device ID.

Peripheral Input/Output

- 19 x GPIO,
- 2 x 12-bit ADCs (analog-to-digital converter) up to 5 channels,
- Internal temperature sensor,
- 3 × SPI (Serial Peripheral Interface),
- 2 x UART (universal asynchronous receiver/transmitter),
- 2 × I^2C (Inter-Integrated Circuit),
- 1 × I^2S (Integrated Inter-IC Sound),
- LED PWM up to 6 channels,
- General DMA with - 3 x Tx + 3 x Rx,
- PWM for Motor control,
- 1 × TWAI® controller compatible with ISO 11898-1 (CAN Specification 2.0),
- 1 x Parallel IO controller (PARLIO),
- USB Serial/JTAG controller.

Security

- Secure boot,
- Flash encryption,
- 4096-bit OTP, up to 1792-bit for customers,
- Cryptographic hardware acceleration:
  - AES-128/256,
  - SHA accelerator,
  - RSA accelerator 3072 bit,
  - random number generator (RNG),
  - digital signature.

Since the processor documentation is only available for the pre-production version, it may change in the final version

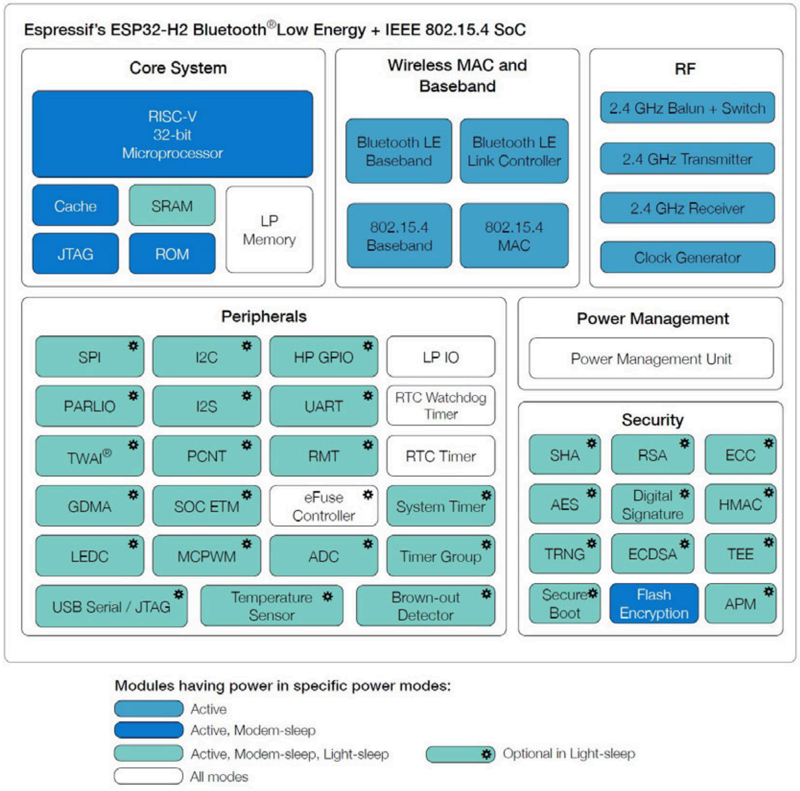


Figure 142: ESP32-H2 functional block diagram

ESP32-H2 Development Boards

There are not many prototype kits with ESP32-H2 SOCs on the market yet. One of them is produced by the Espressif company itself:

- Espressif - ESP32-H2-DevkitM-1[120]

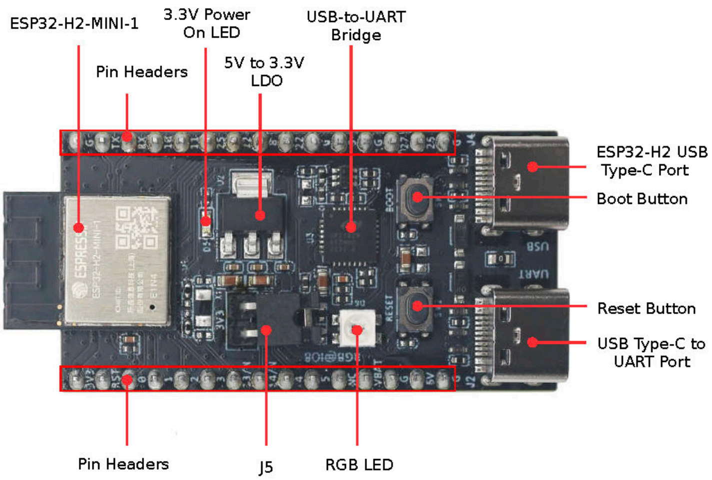


Figure 143: ESP32-H2-DevkitM-1

5.1.3. Nordic Semiconductor

Nordic Semiconductor is a company established in Norway that develops and produces exciting electronic elements capable of transmitting data with radio signals. Among other advanced chips, their portfolio includes the nRF52 family of SoCs, which support low-power radio communication with chosen popular protocols, including Bluetooth Low Energy, ZigBee, NFC, Matter and Thread. They achieved high popularity due to the low power consumption possible by implementing an advanced, adaptive power management system. There are a few versions of chips which support different combinations of protocols. The simplest nRF52805 supports BLE only, while the most advanced nRF52840 is a fully multiprotocol chip with all previously mentioned standards support. A non-exhaustive list of features and nRF boards is present in table 24.

Hardware

Nordic Semiconductor's series of SoCs includes a powerful and efficient ARM® core that, together with advanced power management, allows the creation of advanced projects with ultra-wol power consumption. All SoCs have Flash memory for programs, eliminating the need for connecting the external memory. Flash can be programmed during production, but it also can be re-programmed after implementation in a ready product with OTA (Over The Air). There are some development boards available for nRF52 series SoCs. Nordic Semiconductor produces development kits for nRF52, nRF52833 and nRF52840. For the last one, there is also a USB dongle. Other companies also offer their development boards for the nRF52 series with very interesting Arduino Nano 33 BLE, Adafruit Feather nRF52 Bluefruit LE, and Adafruit Feather nRF52840 Sense among them.

Processor

All nRF52 SoCs are built with 64 MHz ARM® Cortex-M4 core working at 64MHz of clock. The central processor is supported by a floating point calculation unit (FPU).

Memory

Different versions contain different sizes of RAM and Flash memory. The **nRF52805**, **nRF52810**, and **nRF52811** have 192kB of Flash and 24 kB of RAM. The nRF52820 has 256kB of Flash and 32 kB of RAM. The **nRF52832** can have 256 or 512 kB of Flash and 32 or 64 KB of RAM, depending on the version. The nRF 52833 has 512 kB of Flash and 128kB of RAM, and the **nRF52840** has 1MB of Flash for the program and 256kB of RAM for data.

Networking

The nRF52 family was developed to support low-power radio communication using a 2.4GHz band. This set of protocols includes Bluetooth 5.4, Thread, Matter, Zigbee, and Bluetooth Mesh. nRF52 family of SoCs does not support WiFi directly or wired Ethernet network protocol. To achieve such functionality, it is necessary to use a network chip connected via SPI.

Peripherals

The nRF SoCs are equipped with a rich set of peripherals, including:

- GPIO – General Purpose Input Output lines,
- TWI – Two Wire interface that can work in both master or slave mode,
- SPI – Serial Peripheral Interface working in master or slave mode,
- UART – Universal asynchronous receiver/transmitter,
- Timer – timer/counter unit,
- RTC – Real-time counter,
- WDT – Watchdog timer.

In selected models, additional units are available:

- QSPI – Quad SPI and high-speed SPI in some versions,
- I2S – Inter-IC sound interface,
- USB – Universal serial bus device,
- PDM – Pulse Density Modulation (PDM),
- PWM – Pulse Width Modulation,
- COMP – Analog comparator with low power version LPCOMP,
- SAADC – Successive approximation analog-to-digital converter.

All peripherals are connected to the processor via PPI (Programmable Peripheral Interconnect), ensuring flexible use of units. Peripherals have input signals **Task** that trigger their operation and output signals **Event**, which inform of some situation. These signals can be connected, creating hardware dependencies between units, making it possible to synchronise the operation of peripherals without the need for processor use. Some units can be handled with an internal DMA mechanism (EasyDMA) that supports data transmission between peripherals and memory without code execution. To enable safe wireless transmission, some cryptographic hardware units are included:

- RNG – Random Number Generator,
- ACL – Access contol list,
- AAR – Accelerated address resolver,
- CCM – Cipher block chaining - message authentication code,
- ECB – AES electronic codebook,
- Cryptocell.

The Cryptocell is a particular security subsystem developed by ARM®, which provides a device's root of trust (RoT) and cryptographic services.

Table 24: Hardware summary for nRF family SoCs

Version	nRF52805	nRF52810	nRF52811	nRF52820	nRF52832	nRF52833	nRF52840
Bluetooth 5.4	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Bluetooth Mesh				Yes	Yes	Yes	Yes
Thread				Yes		Yes	Yes
Matter							Yes
ZigBee				Yes		Yes	Yes

Version	nRF52805	nRF52810	nRF52811	nRF52820	nRF52832	nRF52833	nRF52840
Flash size [kB]	192	192	192	256	256/512	512	1024
RAM size [kB]	24	24	24	32	32/64	128	256
FPU					Yes	Yes	Yes
Advanced SPI						High speed	High speed, QSPI
USB				Yes		Yes	Yes
Analog	ADC	ADC, Comp	ADC, Comp	Comp	ADC, Comp	ADC, Comp	ADC, Comp
Others		PWM, PDM	PWM, PDM		PWM, PDM, I2S	PWM, PDM, I2S	PWM, PDM, I2S
No. of GPIOs	10	15-32	15-32	18	32	18-42	48

The most popular boards

Nordic Semiconductor created development boards for their SoCs. The most popular is nRF52840 DK (figure 144), which includes the nRF52840 chip, SEGGER J-Link debugger, the micro USB port for flashing, debugging and serial data communication, four user buttons, and four user LEDs. The board has a shape similar to Arduino, with all 48 GPIOS available at the header connectors, and is shipped with an NFC antenna.

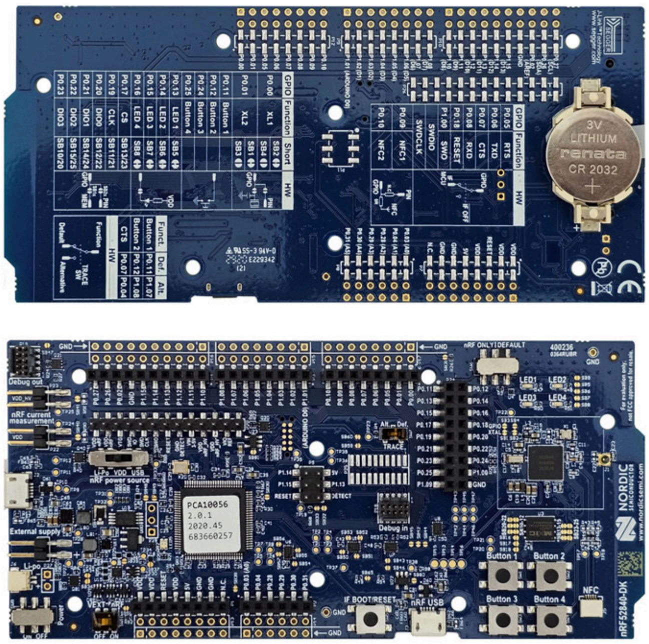


Figure 144: nRF52840 DK development board

Another board created by Nordic Semiconductor is a USB dongle (figure 145) with the same nRF52840 SoC built-in. It doesn't have a debugger but supports uploading the code via USB using the bootloader. It has 15 GPIO lines available.

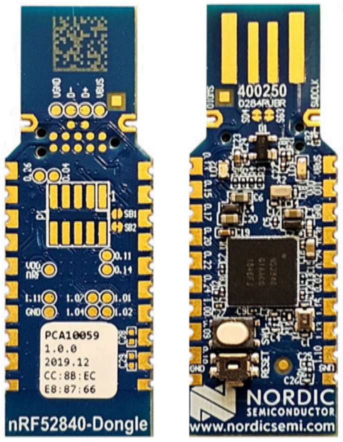


Figure 145: nRF52840 USB dongle

Adaftuit developed an exciting board nRF52840 Feather Sense (figure 146). It contains the nRF52840 SoC and a set of different sensors, making it helpful in developing sensor devices with wireless connectivity. This board also has a Li-Po battery charger and a digital RGB LED.
Sensors include:

- LSM6DS33 accelerometer and gyro,
- LIS3MDL magnetometer,
- APDS9960 proximity, light, colour and gesture sensor,
- MP34DT01-M microphone,
- SHT-30 humidity sensor,
- BMP280 temperature and air pressure sensor.

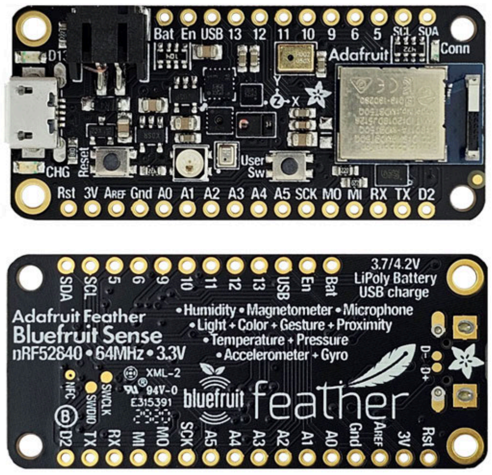

Bat En USB 13 12 11 10 9 6 5 SCL SDA
Adafruit
Conn
D13
CHG
Reset
User Sw
Rst 3V AREF Gnd A0 A1 A2 A3 A4 A5 SCK MO MI RX TX D2
SDA SCL 5 6 9 10 11 12 13 USB En Bat
3.7/4.2V LiPoly Battery USB charge
Adafruit Feather Bluefruit Sense
nRF52840 • 64MHz • 3.3V
• Humidity • Magnetometer • Microphone
• Light + Color + Gesture + Proximity
• Temperature + Pressure
• Accelerometer + Gyro
bluefruit feather
NFC SWDIO SWCLK
D2 TX RX MI MO SCK A5 A4 A3 A2 A1 A0 Gnd AREF 3V Rst

Figure 146: Adafruit nRF52840 Feather Sense

5.1.4. STM32

The STM32 family is developed and manufactured by STMicroelectronics. They are considered advanced and efficient and are known for great technical documentation, versatility, performance, energy efficiency, and reliability. They are also highly configurable and provide a wide range of features.
For a long time, STM32 delivered MCUs without radio modules; thus, they required external radio communication interfaces for IoT applications. Recently, a series of chips have been available with built-in radio modules, primarily using IEEE 802.15.4 (Zigbee, Thread, and other wireless sensor network protocols) rather than 802.11 (WiFi).

STM manufacturer provides developers with development kits, some of which can accommodate popular Arduino shields. They also offer a development SDK based on the popular Eclipse platform (implemented with Java, thus cross-platform). They also ensure at least 10 years of availability and support.

STM32 MCUs are known for their energy-efficient operation, making them suitable for battery-powered and low-power IoT applications.

Thanks to the built-in performance options such as an independent vectorised interrupt system and DMA, industrial grade series can handle video processing, TFT displays, and so on.

Hardware

STM32 SoCs use ARM Cortex-based cores, starting from M0 to M7 [121]. Some of the SoCs integrate 2 cores, such as in the case of the radio-equipped models, where the main core is supported by the extra one (usually M0+), which handles wireless communication protocols. All MCUs are 32-bit. Some STM32 MCUs tend to tolerate a broader range of powering voltages. Thus, they may operate on raw battery cells without needing a voltage conversion and stabilisation.

There are 5 major series of the microcontrollers and microprocessors manufactured by STM:

- High-performance series with Cortex M3 up to Cortex M7.
- Mainstream series with Cortex M0 to Cortex M7.
- Ultra-low power series with Cortex M0+, via Cortex M4, up to Cortex M33.
- Wireless series, with Cortex M4, Cortex M33 and radio coprocessor Cortex M0+.
- Industrial grade MP1 microprocessor series, with a mixture of Cortex A7 and Cortex M4 cores (some chips use only A7 core).

The MP1 series is a raw microprocessor that requires external RAM, Flash, and Input-Output; it is also currently extended with 64-bit versions. It works with Linux and Android and can be equipped with Neural Processing Units (NPU) and 3D graphics processing units (GPU). As they are RAW microprocessors, they are not considered in the scope of this book to be directly IoT applications, eventually in a scenario of the advanced Fog class devices.

Beside STM32 series there is also a SPC5 series, designated for automotive industry. Those MCUs are Power PC architecture-based.

Processor

All STM32 use ARM Cortex cores, single, double or in pair with another ARM core coprocessor, such as in the case of the industrial grade (MP1) microprocessors and wireless (STM32W) microcontrollers series.

Maximum frequencies depend on the ARM Core model and are between 32MHz for Cortex M0+ cores and 550MHz for H7. Industrial series MP1 hits even 1GHz.

The majority of the MCUs are marked as F family. This series is currently replaced with a next-generation G family of chips.

Memory

Built-in RAM, flash, and EEPROM sizes depend on the family of chips and the exact model within this family. Ultra-low-power devices such as STM32L0 microcontroller have only 2kB of RAM, 128B of EEPROM, and 16kB of flash. Conversely, the STM32H7 microcontroller can have up to 1184kB of RAM and 2MB of built-in flash. Most MCUs can extend the memory externally with SPI (even up to dual QSPI interface). Each STM32 series has its variations that vary in the built-in memory size.

Networking

Only the STM32 W series provides radio connectivity integrated into the MCU. Currently, there are 4 chip series (and each has its variations regarding enclosure size, memory size (both RAM and flash), number of GPIO pins available, and some advanced functions such as secure keys management, secure boot, etc.:

- STM32 WL series [122] introducing LoRaWAN, Sigfox, W-MBUS, mioty, and virtually other protocols compatible with (G)FSK, (G)MSK, and BPSK modulations in a single chip,
- STM32 WB0 series [123] designed for energy-efficient applications and Bluetooth 5.3 only,
- STM32 WB series [124] with Zigbee, Thread (OpenThread), Matter and Bluetooth 5.4 and BLE, Zephyr and Cordio stacks,
- STM32 WBA series [125] with Bluetooth 5.3.

Each series has variations, e.g., the STM32 WL series has STM32WLE5 and STM32WL54 that do not support LoRa, as well as versions STM32WLE5 and STM32WL55 with LoRa.

Peripherals

The STM32 family provides all peripherals and interfaces, but availability and amount depend on the family, series, and particular model. STM32 MCUs connect the CPU core to various peripheral modules using a peripheral bus matrix. This matrix allows for flexible routing of communication between the CPU and peripherals. Each peripheral block has associated control registers allowing configuring and controlling their operation. Those registers can be used to set parameters, turn features on or off, and monitor the status of the peripherals.

Peripherals include:

- GPIO,
- timers (including hardware-based pulse generation such as PWM and watchdog timers),
- embedded system protocol interfaces UART (USART), SPI (even up to dual QSPI), I2C, CAN,
- ADC and DAC converters,
- USB, Ethernet, SDIO, camera (CSI), display (DSI),
- RTC interface,
- DMA,
- interrupt controller,
- security and cryptography functions modules.

STM32 has efficient and highly configurable clocks, an NVIC interrupt controller (Nested Vectored Interrupt Controller), and a DMA that, along with timers, provide great capabilities for Real-Time applications of high performance and reliability. Figure 147 presents a sample STM32G4 configuration for clocks.

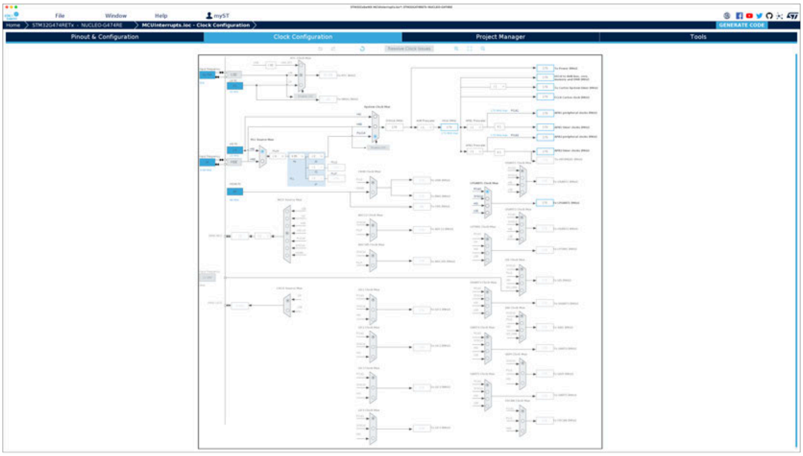


Figure 147: STM32G4 clock configuration capabilities

Video subsystem

Some STM32 MCUs provide computing performance high enough to handle image and video processing, e.g. STM32F7 and STM32H7 series have hardware-accelerated jpeg (and thus mjpeg) encoding and decoding. MP1 series can be equipped with an optional GPU for 3D acceleration. Some of the MCUs include a built-in TFT display controller.

Hardware summary

STM32 shares a common ARM architecture but, depending on the family, has different cores and, thus, performance and applications. The following chapters show a more in-depth review of the STM32 MCU hardware.

The most popular boards

STM provides developers with popular development boards virtually for any family of MCUs. There are also available 3rd party development boards.

There are three types of development boards available (obviously, not for all series):

- Nucleo series that share pinout with Arduino, enabling an easy use of Arduino shields. They provide developers with a built-in ST-link hardware debugger.
- Discovery kits, bigger in size and usually rich in connectors, frequently equipped with external sensors such as MEMS (gyro, accelerometer), microphone, LEDs and so on. They provide developers with a built-in ST-link hardware debugger.
- Evaluation boards are a more advanced version of the Discovery Kits, equipped with a display, external memory, etc. Their purpose is to demonstrate all capabilities of the particular MCU.

Sample USB stick, Nucleo kit and development kit for STM32WB55 are present in figures: 148, 149 and 150, respectively.

Figure 148: STM32WB55 USB stick

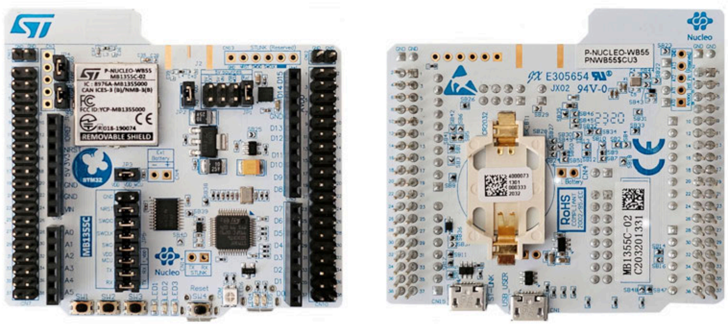


Figure 149: STM32WB55 Nucleo board

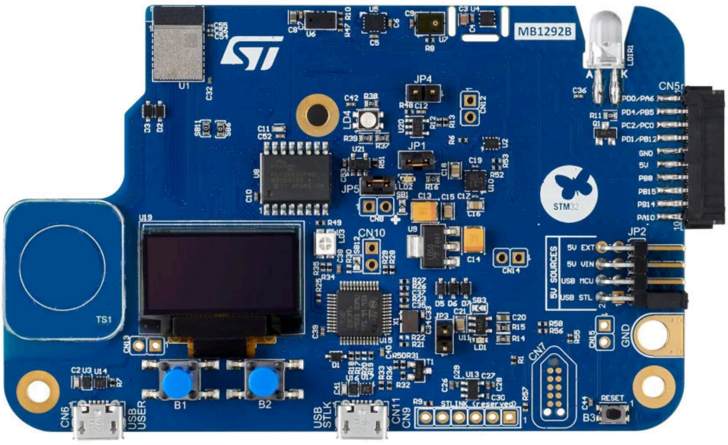


Figure 150: STM32WB55 development board

Sample 3rd party evaluation board for STM32F1 MCU is present in the figure 151.

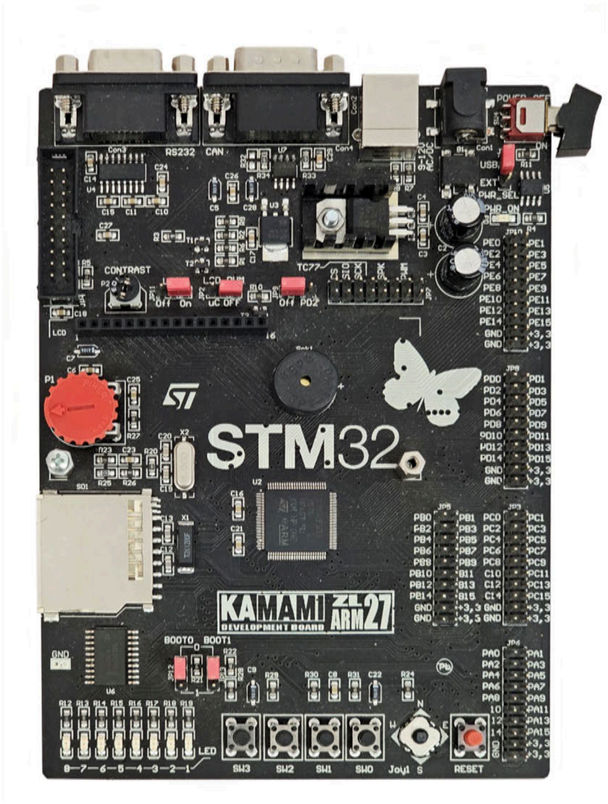


Figure 151: SMT32F1 evaluation board

STM32 Performance Series

The STM32 Performance series features performance ARM Cortex cores such as M4, M7, and M33 with high clock frequencies. This family, even if energy efficient in computing, is intended not to work in energy-constrained environments. Performance series offers bigger RAM and Flash memories, advanced features oriented toward creating rich user

interfaces, such as Chrome-ART Graphic Accelerator, and HDMI support. This series also provides developers advanced instruction sets such as DSP and FPU. STM32 Performance Series chips are bigger, more expensive, and more power-consuming. This last factor, however, can be controlled with advanced power management features.
Applications that benefit from the STM32 Performance series include industrial automation, high-performance IoT devices, motor control, multimedia, audio processing, and more. Performance Series MCUs are often used in applications where there is a need for power and feature-rich peripherals such as displays, cameras, video processing, etc.
A short review of the MCUs and their features is listed in table 25.
Note each row in the table represents a family of devices, and a particular configuration depends on the exact MCU model. Thus, developers, when looking for specific features, should refer to the manufacturer's specifications and list of currently available devices [126].

Table 25: STM32 High Performance series

Series:	CPU (Core/Cores)	RAM	Flash / EEPROM	Interfaces (*-not all chips/ versions)	Voltage	Other features (*-not all chips/ versions)
STM32F2	Cortex M3@120MHz	Up to 128kB	128kB to 1MB	2xUSB OTG FS/HS SDIO USART SPI I2C 2xCAN FS+audio PLL 3×12 bit ADC 2×12 bit DAC* Ethernet IEEE1588 Camera* FSCM*	1.7V to 3.6V	ART - Adaptive Real-Time Accelerator 16 and 32-bit Timers
STM32F4	Cortex M4@84MHz-180MHz	32kB to 384kB	64kB to 2056kB	SDIO USART SPI I2C 2xCAN* I2S+audio PLL SAI* SPDIF RX* MIPI DSI* USB 2.0 OTG FS* 12 bit ADC 12 bit DAC* DFSDM Ethernet IEEE1588* Dual Quad-SPI*/QSPI* Camera* FSCM* SDRAM*	1.7V to 3.6V	Random number generator Chrom-ART Graphic Accelerator* ART Accelerator* TFT LCD Controller* 16 and 32-bit Timers
STM32H5	Cortex M33@250MHz (with DSP+FPU)	32kB/640kB	128kB/ 1024kB to 2048kB	USART SPI I2C 12 bit DAC SDMMC* 2xSDMMC FMC* OctoSPI* 1 or 2 12-bit ADC (5 Msps)* Op-amp* Comparator*	SMPS/ LDO or LDO only 1.62V to 3.6V	Random Number Generator TrustZone* Advanced Encryption Services (AES/SAES PKA OTFDEC HUK ST-iRoT)* ART Accelerator Vbat Battery Voltage Mode 16 and 32-bit Timers SHA DMA

Series:	CPU (Core/Cores)	RAM	Flash / EEPROM	Interfaces (*-not all chips/ versions)	Voltage	Other features (*-not all chips/ versions)
				1 or 2 CAN-FD DCM HDMI-CEC Ethernet IEEE1588*		Digital Temperature Sensor Unique ID
STM32F7	Cortex M7@216MHz (with FPU, single or double precision) L1 cache (Instruction/Data) from 4kB/4kB to 16kB/16kB	From 256kB (includes 64k DTCM) to 512KB (including 128kB DTCM)	64kB to 2048kB	2xUSB OTG FS/HS USART UART SDIO* SPI I2C 1, 2 or 3 CAN v2.0 HDMI-CEC Ethernet IEEE1588* FMC MDIO slave* Camera* Dual mode Quad-SPI I2S+audio PLL 2xSAI 2×12 bit DAC SPDIF-RX 3×12-bit ADC DF SDM* MIPI-SDI* USB HS PHY*	1.7V to 3.6V	TFT LCD controller* 16 and 32-bit timers AES/TDES crypto hardware acceleration* HASH hardware acceleration* JPEG codec hardware accelerated* Chrom-ART Graphic Accelerator
STM32H7	Single core Cortex M7@(280MHz, 480MHz or 550MHz) Double cores Cortex M7@480MHz+Cortex M4@240MHz (with FPU, single or double precision)	564kB(including 128kB DTCM, 432kB system +4kB backup) to 1.4MB(including 128kB DTCM, 64kB ITCM + 16kB backup)	Dual bank, from 128kB to 2MB	2xUSB OTG FS/HS 2xSDMMC USART UART SDIO* SPI I2C 3 CAN (2xFD+1xTT) HDMI-CEC Ethernet IEEE1588* FMC Dual-mode Quad-SPI or 2xOcto-SPI* Camera 3xI2S+audio PLL 4xSAI 2×12 bit DAC SPDIF-RX 3×16-bit ADC (3.6 Msps) Op-amp* Comparator* MIPI-DSI*	“SMPS/ LDO or LDO only 1.62V to 3.6V”	TFT LCD Controller* 16 and 32-bit Timers HR-Timer* Crypto Hardware Acceleration* HASH Hardware Acceleration* JPEG Codec Hardware Accelerated* 4xDMA Flash and RAM Acceleration Chrom-ART Graphic Acceleration Security Services Option* Random Number Generator*

STM32 Mainstream Series

The STM32 Mainstream series is considered a versatile family of MCUs. It is a reasonable balance between power consumption, cost and application flexibility. It is worth

mentioning that the Mainstream series is considered a long-shelf-life one, with support and availability of chips exceeding 10 years.
Mainstream series CPUs are based on ARM Cortex M0/M0+, M3 and M4 and RAM flash memory sizes are in a wide range, from 16kB for simple applications up to 1MB to handle even the most demanding tasks. However, there are no radio modules built in, so IoT applications require an external RF coprocessor/radio device.

A short review of the MCUs and their features is presented in table 26. Note each row in the table represents a family of devices, and a particular configuration depends on the exact MCU model. Thus, developers, when looking for specific features, should refer to the manufacturer's specifications and list of currently available devices [127].

Table 26: STM32 Mainstream series

Series:	CPU (Core/Cores)	RAM (*-not all chips/versions)	Flash / EEPROM	Interfaces (*-not all chips/versions)	Voltage (*-not all chips/ versions)	Other features (*-not all chips/ versions)
STM32C0	Cortex M0+@48MHz	6kB or 12kB	16kB or 32kB	I2C SPI I2S 2xUSART ADC	2.0V to 3.6V	5×16-bit Timers IWDG (Independent Watchdog) WWDG (Window Watchdog)
STM32F0	Cortex M0@48MHz	From 4kB up to 32kB 20B backup data*	From 16kB to 256kB	2xI2C 2xSPI I2S up to 8 USART* CAN* USB* 12-bit DAC* CEC (HDMI-CEC)* 12-bit ADC	1.8V for low-voltage product line* 2.0V to 3.6V* 2.4V to 3.6V*	2xWatchdog Hardware CRC Internal RC PLL RTC Calendar 16 and 32-bit Timers Temperature Sensor Multiple Channel DMA Comparator* Unique ID Touch Sense*
STM32G0	Cortex M0+@64MHz	up to 144 (SRAM)	16kB to 512kB Securable Memory Area*	USART SPI I2C 12-bit ADC (2.5 MSPS) 12-bit DAC 2 channel* Low-Power UART* USB-C Power Delivery* USB DEV/HOST 2.0 FS CAN-FD*	1.7V to 3.6V* 2.0V to 3.6V*	2xWatchdog RTC PLL Main Oscillator and 32kHz Oscillator Temperature Sensor AES-256* Random Number Generator* DMA Comparator* 32-bit Timer* 16-bit MC Timer 16-bit Timer Low-Power Timer*
STM32F1	Cortex M3@(24/36/48/72)MHz	4kB to 96kB	16kB to 1MB	USART SPI I2C 3×12-bit ADC 2×12-bit DAC USB 2.0 FS* FSMC* CAN 2.0B* I2S* SDIO* Ethernet IEEE1588* CEC (HDMI-CEC)*	2.0V to 3.6V, GPIOs are 5V tolerant	16 and 32-bit Timers Temperature Sensor 3-phase MC Timer*
STM32F3	Cortex M4@72MHz (DSP+FPU)	16kB to 80kB CCM-SRAM*	16kB to 512kB	USART SPI I2C USB 2.0 FS* CAN 2.0B* I2S Up to 4 12-bit ADC* 3×16-bit ADC*	1.8V for low-voltage product line 2.0V to 3.6V	Routine Booster (CCM) DMA 16 and 32-bit Timers Hardware CRC Low and High-Speed Oscillators RTC Temperature Sensor Capacitive Touch

Series:	CPU (Core/Cores)	RAM (*-not all chips/versions)	Flash / EEPROM	Interfaces (*-not all chips/versions)	Voltage (*-not all chips/ versions)	Other features (*-not all chips/ versions)
						Sensing 2xUltrafast Comparators* Up to 7 Fast Comparators* Up to 4 Op-amp (PGA)* High-Resolution Timer* Advanced 16-bit PWM Timer
STM32G4	Cortex M4@170MHz (DSP+FPU)	Up to 112kB* 128kB* CCM-SRAM up to 16kB* CCM-SRAM 32kB*	32kB to 512kB Flash Memory with ECC	USART SPI I2C SAI 3/5 12-bit ADC* 4/7 12-bit DAC FSMC* Quad-SPI CAN-FD USB-C Power Delivery USB 2.0 DEV/FS*	1.71V to 3.6V	Math Acceleration (FMAC, Cordic) 4/6 Op-amps (PGA)* Up to 3 Ultrafast Comparators* ART Accelerator Advanced Motor Control Timers Multiple DMA with DMAMUX PLL, Temperature Sensor Vbat Battery Voltage Mode High Resolution Timer*

STM32 Low Power Series

The STM32 Low Power series features a powerful yet energy-efficient family of devices based on the ARM Cortex M0+, M4 and M33. Advanced power management and availability of several power modes enable the construction of battery-powered devices capable of operating for months or even years without a need for recharge, e.g. STM32L4 chips can go as low as down to 0.34uA (micro Amper) of power consumption in their lowest power mode. Still, the versatility of the series brings chips with flash memory up to 2MB for resource-demanding applications.

A short review of the MCUs and their features is listed in table 27. Note each row in the table represents a family of devices, and a particular configuration depends on the exact MCU model. Thus, developers, when looking for specific features, in particular energy efficiency and power saving options, should refer to the manufacturer's specification and list of currently available devices [128].

Table 27: STM32 Ultra Low Power series

Series:	CPU (Core/Cores)	RAM (*-not all chips/ versions)	Flash / EEPROM	Interfaces (*-not all chips/ versions)	Voltage (*-not all chips/ versions)	Other features (*-not all chips/versions)
SMT32L0	Cortex M0@32MHz	up to 20kB	128kB/ 192kB flash 512B/6K EEPROM	12-bit ADC (1.14Msps) USART USART SPI I2C LP UART PVD (Programmable Voltage Detector)* USB 2.0 FS* 2×12-bit DAC*	Dynamic Voltage Scaling Down to 1.8V* Down to 1.65V*	Dynamic Voltage Scaling 5 Clock Sources Advanced RTC with Calibration 16-bit Timers Low Power Timers 2x Watchdog DMA AES-128 Temperature Sensor* Segment LCD Driver (4×52/ 8×48)* Random Number Generator*

Series:	CPU (Core/Cores)	RAM (*-not all chips/ versions)	Flash / EEPROM	Interfaces (*-not all chips/ versions)	Voltage (*-not all chips/ versions)	Other features (*-not all chips/versions)
						Touch Sense* 2xUltra Low PPower Comparators
STM32L4	Cortex-M4@80MHz (DSP+FPU)	40kB to 320kB	64kB to 1024kB	USART UART SPI I2C Quad-SPI 1/2 SAI+Audio PLL* SWP (Bank Swap Pin) 1/2 CAN* 1/2 12-bit DAC* FSMC 4/8 Sigma Delta Interfaces* 1-3 12-bit ADC (5Msps) with 16-bit Over-sampling USB 2.0 OTG* USD 2.0 DEV*	1.71V to 3.6V	ART Accelerator Chrom-ART Graphic Accelerator 16 and 32-bit Timers Temperature Sensor Vbat Battery Voltage Mode Unique ID Capacitive Touch Sensing Hardware Crypto AES-128/ 256* Hardware SHA-256* 1/2 Op Amps
STM32L4+	Cortex-M4@120MHz (DSP+FPU)	320/640kB	512kB to 2048kB	USART UART SPI I2C 2xQuad-SPI SAI+Audio PLL* CAN Camera Interface 2×12-bit DAC* SDIO FSMC 4/8 Sigma Delta Interfaces* 1/3 12-bit ADC (5Msps) with 16-bit Over-sampling USB 2.0 OTG TFT Display Interface* MIPI-DSI*	1.71V to 3.6V	2xWatchdog
STM32L5	Cortex M33@110MHz (TrustZone+DSP+FPU)	256kB	256kB to 512kB	USART UART SPI I2C Octo-SPI FMC* SAI+Audio PLL* CAN-FD 2x 4ch Sigma Delta Interfaces* 2×12-bit DAC* 2×12-bit ADC (5 Msps) with 16-bit Over-sampling USB 2.0 DEV USB 2.0 HS USB-C Power Delivery	1.71V to 3.6V	16 and 32-bit Timers ART Accelerator Hardware Accelerated SHA Random Number Generator Temperature Sensor Vbat Battery Voltage Mode Unique ID Capacitive Touch Sensing 2xOp Amps 2xComparators Hardware Crypto AES, PKA, OTFDEC 128/256-bit*
STM32U5	Cortex M33@160MHz (TrustZone+DSP+FPU)	274kB/ 786kB/ 2514kB Dual bank flash*	128kB to 4096kB	SDIO 1/2 Octo-SPI* Hexadeca-SPI* FSMC* HSPI* USB 2.0 FS* USB 2.0 HS* USB-C Power Delivery* TFT-LCI and DSI Interfaces* 2×12-bit DAC* 1/2 14-bit ADC (2 Msps)* 1×12-bit ADC (2Msps) USART UART LPUART SPI I2C	1.71V to 3.6V	16 and 32-bit Timers 2xAdvanced Motor Control Timers 4xUltra Low Power Timers Neo-Chrom GPU* Chrom-ART Graphic Accelerator* Hardware Crypto AES 128/ 256, PKA, OTFDEC 128/ 256-bit* 2xWatchdog RTC 2xOp Amps 2xComparators Hardware Accelerated SHA and MD5 Random Number Generator Capacitive Touch Sensing LPDMA

Series:	CPU (Core/Cores)	RAM (*-not all chips/ versions)	Flash / EEPROM	Interfaces (*-not all chips/ versions)	Voltage (*-not all chips/ versions)	Other features (*-not all chips/versions)
				CAN-FD ADF* Camera* MDF* SAI* SD/MMC*		Temperature Sensor Unique ID

STM32 Wireless Series

The STM32 Wireless series is the only chip family with built-in wireless capabilities. This series uses ARM Cortex M4 and additional Arm Cortex M0+ as a radio coprocessor or, eventually, a single ARM Cortex M0+ core in low power version.
The STM32 Wireless series features built-in radio modules (sub-gigahertz or 2.4GHz) for IoT protocols such as:

- Bluetooth/BLE,
- LoRa/LoRaWAN,
- Matter,
- Zigbee,
- Thread/OpenThread,
- sigFox,
- mioty,
- M-Bus,
- 2FSK,
- 2GFSK,
- BPSK,
- GMSK.

Applications that benefit from the STM32 Wireless series include industrial automation, IoT devices, and smart homes.

A short review of the MCUs and their features is presented in table 28. Note each row in the table represents a family of devices, and a particular configuration depends on the exact MCU model. Thus, developers, when looking for specific features, should refer to the manufacturer's specification and list of currently available devices [129].

Table 28: STM32 Wireless series

Series:	CPU (Core/ Cores)	RAM (*-not all chips/ versions)	Flash / EEPROM	Interfaces (*-not all chips/ versions)	Voltage (*-not all chips/ versions)	Wireless communication (*-not all chips/ versions)	Other features (*-not all chips/versions)
STM32WL	Cortex M4@48MHz Cortex M0+@48MHz*	Up to 64kB	Up to 256kB	USART LPUART SPI I2C I2S 1×12-bit DAC 1×12-bit ADC	1.8V to 3.6V LDO with DC-to-DC converter built-in	Multi-Modulation Sub-GHz Radio 150MHz-960MHz 2xProgrammable Power Outputs LoRa* (G)FSK	16 and 32-bit Timers ART Accelerator Hardware Crypto AES 128/256, PKA Random Number

Series:	CPU (Core/ Cores)	RAM (*-not all chips/ versions)	Flash / EEPROM	Interfaces (*-not all chips/ versions)	Voltage (*-not all chips/ versions)	Wireless communication (*-not all chips/ versions)	Other features (*-not all chips/versions)
						(G)MSK BPSK	Generator PCROP/WRP Temperature Sensor Unique ID DMA 2xUltra Low Power Comparators RTC Low Power Timer
STM32WB0	Cortex M0+@64MHz	64kB	512kB	SPI LPUART USART I2C I2S IrDA 1×12-bit ADC	1.7V to 3.6V	2.4GHz BLE 5.3	RTC Watchdog Random Number Generator ECC RSA Low Power Timer DMA RTC 16-bit Timers Unique ID Hardware Crypto AES 128/256, PKA, RSA Vbat Monitoring Temperature Sensor
STM32WB	Cortex M4@64MHz Cortex M0+@32MHz	48kB to 256kB	256kB to 1024kB	SPI LPUART USART I2C I2S SAI* Quad-SPI* 1×12-bit ADC USB 2.0 FS*	1.71V to 3.6V* 2.0V to 3.6V*	2.4GHz BLE 5.2	16 and 32-bit Timers 1xComparator Hardware Crypto AES 128/256, PKA Random Number Generator Temperature Sensor Unique ID Vbat monitoring RTC Low Power Timer
STM32WBA	Cortex M33@100MHz (MPU+DSP+FPU)	96kB* 128kB* 512B OTP	512kB/ 1024kB	I2C SPI LPUART USART 1×12-bit ADC (2.5Mspip) hardware oversampling	1.71V to 3.6V	2.4GHz BLE 5.4	16 and 32-bit Timers 2xWatchdog IR Timer RTC Hardware Crypto AES/S-AES 128/ 256, PKA SHA Random Number Generator Temperature Sensor Unique ID Vbat monitoring ART-Accelerator Low Power Timer 1xComparator Capatitive Touch Sensing

5.1.5. Raspberry Pi General Information

The Raspberry Pi is a series of small single-board computers developed in the UK by the Raspberry Pi Foundation to promote modern computer science in schools and create electronic communities. Adding the 40-pin GPIO connector to the computer board allows developers to improve their programming skills and opens new horizons in controlling processes and devices unavailable for desktop computers. According to the Raspberry Pi Foundation, the board's sales in July 2017 reached nearly 15 million units. The first generation of this new board type was developed and then released in February 2012 – **Raspberry Pi Model B**. Each Raspberry Pi board contains hardware modules which together make it a wholly usable PC like a computer whose size fits the typical credit card (85/56 mm) size and small power consumption < 3.5 W. This makes this kind of single board computer one of the most popular in the developers' community. Today, thousands of hardware implementation projects exist for users who want to learn modern hardware and software controlling units and include them in their projects.

A dozen even more powerful clones share a familiar concept, size, and connectors (mostly GPIO, USB, Ethernet and CSI/DSI) with genuine RPIs, such as OrangePi, BananaPi and others. They differ in CPU (MCU), GPU, and RAM; some are even more powerful than genuine Raspberries. Still, they are ARM-based and powered with Android or Linux.

Because the power consumption in the latest devices, such as Raspberry Pi 4 and 5, can exceed 20W, they are considered mains powered and, thus, in the IoT ecosystem, play the role of gateways, routers and, in general, fog-class devices rather than edge-class. Still, this classification is fuzzy as there are dozens of examples of how to use Raspberry Pi, e.g. sensors-network component.

Besides advanced fog-class devices, Raspberry recently started to hit the edge-class IoT development market (low-powered, end-node devices) with their RP2040 MCU.

Raspberry Pi Fog Class Devices Hardware Review

Hardware

Hardware boards (depending on the manufactured model) contain interfaces: Ethernet, Bluetooth, WiFi, USB, AUDIO, HDMI and GPIO ports [130]. The Raspberry Pi boards have evolved through several versions varying in memory capacity, System on Chips (SoC) and processor units. The first generation models of Raspberry Pi used the Broadcom BCM2835 (ARMv6 architecture) based on a 700 MHz ARM11176JZF-S processor and VideoCore IV graphics processing Unit (GPU). Models Pi 1 and B+, developed later, use the five-point USB/Ethernet hub chip, while the Pi 1 Model B only contains two. The Pi Zero USB port is connected directly to the SoC and uses the (OTG) micro USB port.

Processor

The first Raspberry Pi 2 models use the 900 MHz Broadcom BCM2836 SoC 32-bit quad-core ARM Cortex-A7 processor with a shared 256 KB L2 cache. After these earlier models, the Raspberry Pi 2 V1.2 has been upgraded to a Broadcom BCM2837 SoC equipped with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor. Next, the Raspberry Pi 3 series uses the same SoC. They use the Broadcom BCM2837 SoC with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor with a 512 KB shared L2 cache. The Raspberry Pi 3B+ uses the same processor (BCM2837B0) running at 1.4 GHz. The Raspberry Pi 4 is based on Broadcom BCM2711, a quad-core Cortex-A72 64-bit SoC at 1.5 GHz. Raspberry Pi 5 works with a maximum of 2.4GHz. The following Raspberry Pi generations will be increasingly powerful, but their power consumption is also rising, forcing developers to use CPU and GPU heatsinks and more robust power sources.

RAM

The initial Raspberry Pi boards were designed with 128 MB RAM, which was allocated between the GPU and CPU by default. In the newer edition (including Model B and Model A), the RAM was extended to 256 MB and split into the regions. The default split was 192 MB (RAM for CPU), which is sufficient for standalone 1080p video decoding or 3D modelling. Models B with 512 MB RAM initially, memory was split into files released (arm256_start.elf, arm384_start.elf, arm496_start.elf) for 256 MB, 384 MB and 496 MB CPU RAM (and 256 MB, 128 MB and 16 MB video RAM). The Raspberry Pi 2 and 3 are shipped with 1 GB of RAM. The Raspberry Pi 4 can have 1, 2, 4 or even 8 GB of RAM. The Raspberry Pi Zero and Zero W contains 512 MB of RAM.

Networking

The Model A, A+ and Pi Zero have no dedicated Ethernet interface and can be connected to a network using an external USB Ethernet or WiFi adapter. In Models B and B+, the Ethernet port is built-in to the USB Ethernet adapter using the SMSC LAN9514 chip. The Raspberry Pi 3 and Pi Zero W (wireless) models are equipped with 2.4 GHz WiFi 802.11n (150 Mbit/s) and Bluetooth 4.1 (24 Mbit/s) based on Broadcom BCM43438 FullMAC chip. The Raspberry Pi 3 also has a 10/100 Ethernet port. The latest Raspberry Pi 4 contains a dual band 2.4 / 5 GHz WiFi network adapter (IEEE 802.11ac), Bluetooth 5.0, and Gigabit Ethernet.

Peripherals

The Raspberry Pi may be controlled with any generic USB keyboard and mouse. It can also use USB storage, USB to MIDI converters, and virtually any other device/component which is USB compatible. Other peripherals can be attached through the various pins and connectors on the surface of the Raspberry Pi.

Video subsystem

The video controller supports standard modern TV resolutions, such as HD and Full HD, and higher. It can emit 640 × 350 EGA; 640 × 480 VGA; 800 × 600 SVGA; 1024 × 768 XGA; 1280 × 720 720p HDTV; 1280 × 768 WXGA variant; 1280 × 800 WXGA variant; 1280 × 1024 SXGA; 1366 × 768 WXGA variant; 1400 × 1050 SXGA+; 1600 × 1200 UXGA; 1680 × 1050 WXGA+; 1920 × 1080 1080p HDTV; 1920 × 1200 WUXGA. Higher resolutions, such as up to 2048 × 1152, may work or even 3840 × 2160 at 15 Hz. Although the Raspberry Pi 3 does not include H.265 hardware decoders, the CPU is more powerful than its predecessors, potentially fast enough for software decoding H.265-encoded videos. The Raspberry Pi 3 GPU runs at a higher clock frequency – 300 or 400 MHz, compared to 250 MHz in previous versions. The Raspberry Pi can generate 576i and 480i composite video signals, as used on old-style (CRT) TV screens and less-expensive monitors through standard connectors – either RCA or 3.5 mm phono connector, depending on the models. The television signal standards supported are PAL-BGHID, PAL-M, PAL-N, NTSC and NTSC-J. The Raspberry Pi 4 has two micro HDMI connectors that support 4K displays with a refreshing rate of 60Hz.

Real-Time Clock

None of the current Raspberry Pi models has a built-in real-time clock. Developers who need real clock time in their project can retrieve the time from a network time server (NTP) or use the external RTC module connected to the board via SPI or I^{2}C interface. To save the file system consistency, the Raspberry Pi automatically saves time on shutdown and reloads time at boot. One of the best RTC solutions for keeping the proper board time is to use the I^{2}C DS1307 chip containing a hardware clock with a battery power backup.

Hardware Specification

Following tables 29, 30, 31, 32 and 33 present technical details of the RPI fog class IoT devices.

Table 29: Raspberry Pi Models A Comparative Table

Version	Model A		
	RPi 1 Model A	**RPi 1 Model A+**	**RPi 3 Model A+**
Release date	2/1/2013	11/1/2014	11/1/2018
Target price (USD)	25	20	25
Instruction set	ARMv6Z (32-bit)		ARMv8 (64-bit)
SoC	Broadcom BCM2835		Broadcom BCM2837B0
FPU	VFPv2; NEON not supported		VFPv4 + NEON
CPU	1× ARM1176JZF-S 700 MHz		4× Cortex-A53 1.4 GHz
GPU	Broadcom VideoCore IV @ 250 MHz (BCM2837: 3D part of GPU @ 300 MHz, video part of GPU @ 400 MHz)		
	OpenGL ES 2.0 (BCM2835, BCM2836: 24 GFLOPS / BCM2837: 28.8 GFLOPS)		
	MPEG-2 and VC-1 (with license), 1080p30 H.264/MPEG-4 AVC high-profile decoder and encoder (BCM2837: 1080p60)		
Memory (SDRAM)	256 MB (shared with GPU)	512 MB (shared with GPU) as of 4 May 2016. Older boards had 256 MB (shared with GPU)	
USB 2.0 ports	1 (direct from BCM2835 chip)		1 (direct from BCM2837B0 chip)
Video input	15-pin MIPI camera interface (CSI) connector, used with the Raspberry Pi camera or Raspberry Pi NoIR camera		
Video outputs	HDMI (rev 1.3) composite video (RCA jack), MIPI display interface (DSI) for raw LCD panels	HDMI (rev 1.3), composite video (3.5 mm TRRS jack), MIPI display interface (DSI) for raw LCD panels	
Audio inputs	As of revision 2 boards via I^{2}S		
Audio outputs	Analog via 3.5 mm phone jack; digital via HDMI and, as of revision 2 boards, I^{2}S		
On-board storage	SD, MMC, SDIO card slot (3.3 V with card power only)	MicroSDHC slot	
On-board network	None		2.4 GHz and 5 GHz IEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2/BLE
Low-level peripherals	8× GPIO plus the following, which can also be used as GPIO: UART, I^{2}C bus, SPI bus with two chip selects, I^{2}S audio +3.3 V, +5 V, ground	17× GPIO plus the same specific functions, and HAT ID bus	
Power ratings	300 mA (1.5 W)	200 mA (1 W)	
Power source	5 V via MicroUSB or GPIO header		
Size	85.60 mm × 56.5 mm (3.370 in × 2.224 in), excluding	65 mm × 56.5 mm × 10 mm	65 mm x 56.5 mm

	protruding connectors	(2.56 in × 2.22 in × 0.39 in), same as HAT board	
Weight	31 g (1.1 oz)	23 g (0.81 oz)	
Console	Adding a USB network interface via tethering or a serial cable with an optional GPIO power connector		
Generation	1	1 +	3+
Obsolescence Statement	n/a	n/a	in production until at least January 2023
Type	**Model A**		

Table 30: Raspberry Pi Models B Comparative Table

Version	**Model B**					
	RPi 1 Model B	**RPi 1 Model B+**	**RPi 2 Model B**	**RPi 2 Model B v1.2**	**RPi 3 Model B**	**RPi 3 Model B+**
Release date	April–June 2012	7/1/2014	2/1/2015	10/1/2016	2/1/2016	3/14/2018
Target price (USD)	35	25	35			
Instruction set	ARMv6Z (32-bit)		ARMv7-A (32-bit)	ARMv8-A (64/32-bit)		
SoC	Broadcom BCM2835		Broadcom BCM2836	Broadcom BCM2837		Broadcom BCM2837B0
FPU	VFPv2; NEON not supported		VFPv3 + NEON	VFPv4 + NEON		
CPU	1× ARM1176JZF-S 700 MHz		4× Cortex-A7 900 MHz	4× Cortex-A53 900 MHz	4× Cortex-A53 1.2 GHz	4× Cortex-A53 1.4 GHz
GPU	Broadcom VideoCore IV @ 250 MHz (BCM2837: 3D part of GPU @ 300 MHz, video part of GPU @ 400 MHz)					
	OpenGL ES 2.0 (BCM2835, BCM2836: 24 GFLOPS / BCM2837: 28.8 GFLOPS)					
	MPEG-2 and VC-1 (with license), 1080p30 H.264/MPEG-4 AVC high-profile decoder and encoder (BCM2837: 1080p60)					
Memory (SDRAM)	512 MB (shared with GPU) as of 4 May 2016. Older boards had 256 MB (shared with GPU)		1 GB (shared with GPU)			
USB 2.0 ports	2 (via on-board 3-port USB hub)	4 (via on-board 5-port USB hub)				
Video input	15-pin MIPI camera interface (CSI) connector, used with the Raspberry Pi camera or Raspberry Pi NoIR camera					
Video outputs	HDMI (rev 1.3), composite video (RCA jack), MIPI display interface (DSI) for raw LCD panels	HDMI (rev 1.3), composite video (3.5 mm TRRS jack), MIPI display interface (DSI) for raw LCD panels				
Audio inputs	As of revision 2 boards via I²S					
Audio outputs	Analog via 3.5 mm phone jack; digital via HDMI and, as of revision 2 boards, I²S					
On-board storage	SD, MMC, SDIO card slot	MicroSDHC slot			MicroSDHC slot, USB Boot Mode	
On-board network	10/100 Mbit/s Ethernet (8P8C) USB adapter on the USB hub				10/100 Mbit/s Ethernet,	10/100/1000 Mbit/s Ethernet (real speed max 300 Mbit/s),
					802.11b/g/n single band 2.4 GHz wireless,	802.11b/g/n/ac dual band 2.4/5 GHz wireless,
					Bluetooth 4.1 BLE	Bluetooth 4.2 LS BLE
Low-level peripherals	8× GPIO plus the following, which can also be used as GPIO:	17× GPIO plus the same specific functions and HAT ID bus				

Version	Model B					
	UART, I^{2}C bus, SPI bus with two chip selects, I^{2}S audio +3.3 V, +5 V, ground.					
	An additional 4× GPIO are available on the P5 pad if the user is willing to make solder connections					
Power ratings	700 mA (3.5 W)	200 mA (1 W) average when idle, 350 mA (1.75 W) maximum under stress (monitor, keyboard and mouse connected)	220 mA (1.1 W) average when idle, 820 mA (4.1 W) maximum under stress (monitor, keyboard and mouse connected)		300 mA (1.5 W) average when idle, 1.34 A (6.7 W) maximum under stress (monitor, keyboard, mouse and WiFi connected)	459 mA (2.295 W) average when idle, 1.13 A (5.661 W) maximum under stress (monitor, keyboard, mouse and WiFi connected)
Power source	5 V via MicroUSB or GPIO header					
Size	85.60 mm × 56.5 mm (3.370 in × 2.224 in), excluding protruding connectors				85.60 mm × 56.5 mm × 17 mm (3.370 in × 2.224 in × 0.669 in)	
Weight	45 g (1.6 oz)					
Console	Adding a USB network interface via tethering or a serial cable with optional GPIO power connector					
Generation	1	1 +	2	2 ver 1.2	3	3+
Obsolescence	n/a	n/a	n/a	n/a	n/a	in production until at least January 2023
Statement						
Type	**Model B**					

Table 31: Raspberry Pi Models Compute Module Comparative Table

Version	Compute Module*			
	Compute Module 1	**Compute Module 3** & **Compute Module 3 Lite**	**Compute Module 3+** & **Compute Module 3+ Lite**	**Compute Module 4** & **Compute Module 4 Lite**
Release date	April 2014	January 2017	January 2017	October 2020
Instruction set	ARMv6Z (32-bit)	ARMv8-A (64/32-bit)		
SoC	Broadcom BCM2835	Broadcom BCM2837	Broadcom BCM2837B0	Broadcom BCM2711
FPU	VFPv2; NEON not supported	VFPv4 + NEON		
CPU	1× ARM1176JZF-S 700 MHz	4× Cortex-A53 1.2 GHz		4× Cortex-A72 1.5 GHz
GPU	Broadcom VideoCore IV @ 250 MHz (BCM2837: 3D part of GPU @ 300 MHz, video part of GPU @ 400 MHz)			Broadcom VideoCore VI @ 500 MHz
Memory (SDRAM)	512 MB (shared with GPU)	1 GB (shared with GPU)		1,2,4,8 GB
USB 2.0 ports	1 (direct from BCM2835 chip)	1 (direct from BCM2837 chip)	1 (direct from BCM2837B0 chip)	1
Video input	2× MIPI camera interface (CSI)			2-lane MIPI CSI camera interface, 4-lane MIPI CSI camera interface
Video outputs	1xHDMI			2xHDMI
On-board storage	4 GB eMMC flash memory chip	4 GB eMMC flash memory chip or MicroSDHC slot for Lite version	8/16/32 GB eMMC flash memory chip or MicroSDHC slot for Lite version	8/16/32 GB eMMC flash memory chip or MicroSDHC slot for Lite version

Version	Compute Module*			
On-board network	None			10/100/1000Mbps Ethernet b/g/n/ac dual-band (2.4/5GHz) WiFi 5.0 BLE (optional)
Low-level peripherals	46× GPIO, some of which can be used for specific functions, including I²C, SPI, UART, PCM, PWM			28 × GPIO supporting either 1.8v or 3.3v signalling and peripheral options
Power ratings	200 mA (1 W)	700 mA (3.5 W)	not rated	not rated
Power source	2.5–5 V, 3.3 V, 2.5–3.3 V, and 1.8 V			5V
Size	67.6 mm × 30 mm (2.66 in × 1.18 in) 67.6 mm × 31 mm (2.66 in × 1.22 in) 55 mm x 40 mm			
Obsolescence	manufacturing until at least January 2026	manufacturing until at least January 2028		
Type	Compute Module*			

Table 32: Raspberry Pi Models Zero Comparative Table

Version	Zero		
	RPi Zero PCB v1.2	**RPi Zero PCB v1.3**	**RPi Zero W**
Release date	11/1/2015	5/1/2016	2/28/2017
Target price (USD)	5		10
Instruction set	ARMv6Z (32-bit)		
SoC	Broadcom BCM2835		
FPU	VFPv2; NEON not supported		
CPU	1× ARM1176JZF-S 1 GHz		
GPU	Broadcom VideoCore IV @ 250 MHz (BCM2837: 3D part of GPU @ 300 MHz, video part of GPU @ 400 MHz)		
	OpenGL ES 2.0 (BCM2835, BCM2836: 24 GFLOPS / BCM2837: 28.8 GFLOPS)		
	MPEG-2 and VC-1 (with license), 1080p30 H.264/MPEG-4 AVC high-profile decoder and encoder (BCM2837: 1080p60)		
Memory (SDRAM)	512 MB (shared with GPU)		
USB 2.0 ports	1 Micro-USB (direct from BCM2835 chip)		
Video input	None	MIPI camera interface (CSI)	
Video outputs	Mini-HDMI, 1080p60, composite video via marked points on PCB for optional header pins		
Audio inputs	As of revision 2 boards via I²S		
Audio outputs	Mini-HDMI, stereo audio through PWM on GPIO		
On-board storage	MicroSDHC		
On-board network	None		802.11b/g/n single band 2.4 GHz wireless, Bluetooth 4.1 BLE
Low-level peripherals	17× GPIO plus the same specific functions and HAT ID bus		
Power ratings	100 mA (0.5 W) average when idle, 350 mA (1.75 W) maximum under stress (monitor, keyboard and mouse connected)		
Power source	5 V via MicroUSB or GPIO header		
Size	65 mm × 30 mm × 5 mm (2.56 in × 1.18 in × 0.20 in)		
Weight	9 g (0.32 oz)		
Console	Adding a USB network interface via tethering or a serial cable with an optional GPIO power connector		
Generation	PCB ver 1.2	PCB ver 1.3	W (wireless)
Obsolescence	n/a, or see PCB ver 1.3	Zero is currently stated as being not before January 2022	n/a
Statement			

Version	**Zero**
Type	**Zero**

Table 33: Raspberry Pi 4B & 5 Models Table

Version	**Model B**	
	Pi4	**Pi5**
Release date	01/06/2019	28/09/2023
SoC	Broadcom BCM2711	Broadcom BCM2712
CPU	1,5 GHz quad-core ARM-8 Cortex-A72 (64-bit)	2.4 GHz quad-core Cortex-A76 (64bit)
GPU	Broadcom VideoCore VII	
Memory (SDRAM)	1GB/2GB/4GB/8GB	4GB/8GB
USB ports	USB 2×2.0 2×3.0	
Video outputs	Composite (PAL/NTSC) 2x micro HDMI	
Audio outputs	2xmicro HDMI	
On-board storage	MicroSD	MicroSD with SDR104
On-board network	100/1000 Ethernet (RJ45), WiFi (2.4-5 GHz 802.11b/g/n/ac) Bluetooth 5.0 BLE	
Low-level peripherals	40x GPIO, CSI, DSI	40x GPIO, 2CSI/DSI, PCIe 2.0
Power source	5V/AA UCB-C, PoE or GPIO	
Size	85,60 mm × 56,50 mm	
Weight	46 g	
OS systems	Raspbian, Windows 10 IoT Core, OSMC_Pi2, NOOBS, RISC OS, Ubuntu MATE, Linux Q83, Android, Android TV	
Type	**Model B**	

Raspberry Pi Boards

As for today, on the market, a few models of Raspberry Pi boards are available, from tiny ones to more powerful ones. Users can choose the right board to fit the price and functionality of their project development needs. Figures 152, 153, 154, 155, 156, 157, 158, 159, 160 and 161, are presenting Raspberry Pi models, starting from the simplest and finishing on the most advanced and most modern ones.

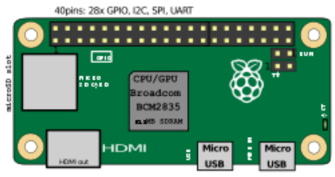


Figure 152: Raspberry Pi Zero [131].

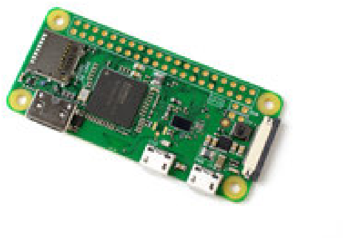


Figure 153: Raspberry Pi Zero W[132].

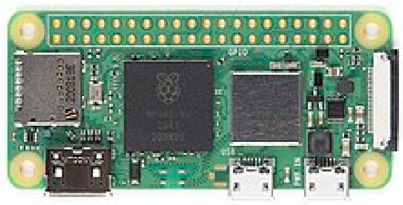


Figure 154: Raspberry Pi Zero 2 W[133].

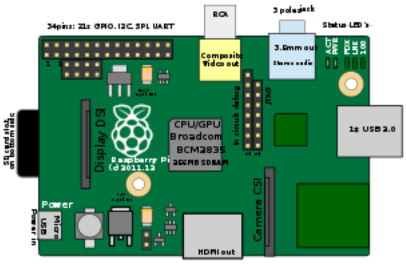


Figure 155: Raspberry Pi 1 Model A

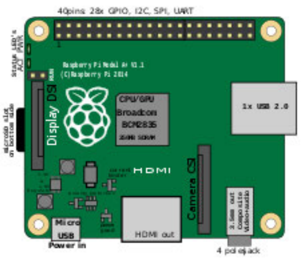


Figure 156: Raspberry Pi 1 Model A+ revision 1.1 [134].

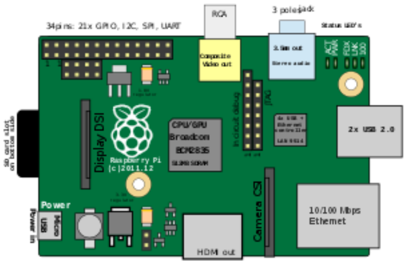


Figure 157: Raspberry Pi 1 Model B revision 1.2 [135].

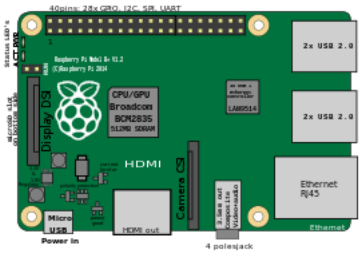


Figure 158: Raspberry Pi 2 [136].

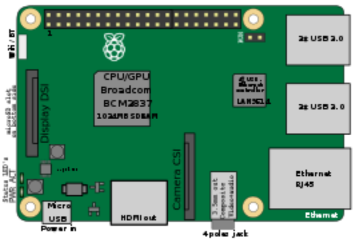


Figure 159: Raspberry Pi 3 [137].

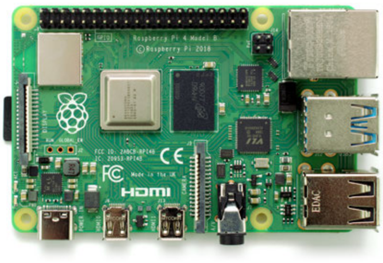


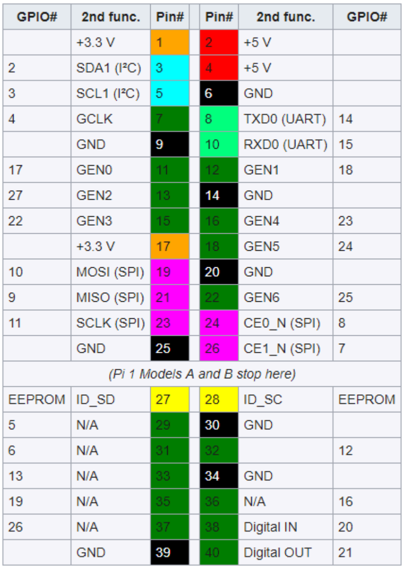
Figure 160: Raspberry Pi 4 [138].

Figure 161: Raspberry Pi 5 [139].

General-Purpose Input-Output (GPIO) Connector

Each Raspberry Pi model is equipped with a standard 34/40-pis male connector containing universal GPIO ports, VCC 3.3/5V, GND, CLK, I2C/SPI bus pins, which developers can use to connect their external sensors, switches and other controlled devices to the Raspberry Pi board and then program their behaviour within the code loaded to the board.

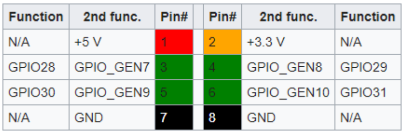
- Raspberry Pi 1 Models A+ and B+, Pi 2 Model B, Pi 3 Model B and Pi Zero (and Zero W) GPIO J8 have a 40-pin pinout. Raspberry Pi 1 Models A and B have only the first 26 pins.



GPIO#	2nd func.	Pin#	Pin#	2nd func.	GPIO#
	+3.3 V	1	2	+5 V	
2	SDA1 (I²C)	3	4	+5 V	
3	SCL1 (I²C)	5	6	GND	
4	GCLK	7	8	TXD0 (UART)	14
	GND	9	10	RXD0 (UART)	15
17	GEN0	11	12	GEN1	18
27	GEN2	13	14	GND	
22	GEN3	15	16	GEN4	23
	+3.3 V	17	18	GEN5	24
10	MOSI (SPI)	19	20	GND	
9	MISO (SPI)	21	22	GEN6	25
11	SCLK (SPI)	23	24	CE0_N (SPI)	8
	GND	25	26	CE1_N (SPI)	7
(Pi 1 Models A and B stop here)					
EEPROM	ID_SD	27	28	ID_SC	EEPROM
5	N/A	29	30	GND	
6	N/A	31	32		12
13	N/A	33	34	GND	
19	N/A	35	36	N/A	16
26	N/A	37	38	Digital IN	20
	GND	39	40	Digital OUT	21

Figure 162: Raspberry Pi 1 pins

- Model B rev. 2 also has a pad (P5 on the board and P6 on the schematics) of 8 pins, offering access to 4 GPIO connections.



Function	2nd func.	Pin#	Pin#	2nd func.	Function
N/A	+5 V	1	2	+3.3 V	N/A
GPIO28	GPIO_GEN7	3	4	GPIO_GEN8	GPIO29
GPIO30	GPIO_GEN9	5	6	GPIO_GEN10	GPIO31
N/A	GND	7	8	GND	N/A

Figure 163: Raspberry Pi 2 & 3 pins

HDMI Port

Each Raspberry Pi model is equipped with the standard, mini or micro HDMI port, which allows the user to connect the monitor or TV set to the board. The electronic schematic is shown in the picture.

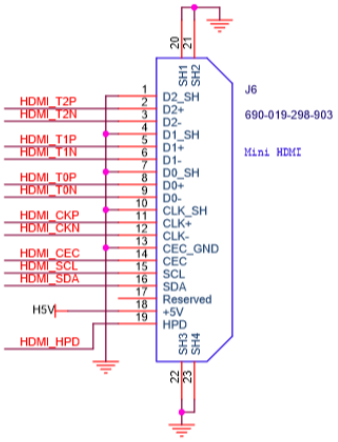


Figure 164: Raspberry HDMI port connection schematic

Camera Port CSI

Raspberry Pi boars Zero, 1, A+, 2, 3, 4 and 5 are equipped with a Camera interface (CSI) port, allowing the user to connect the CCD camera following the MIPI standard.

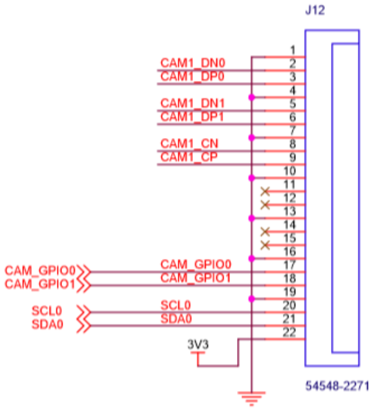


Figure 165: Raspberry CSI camera schematic [140].

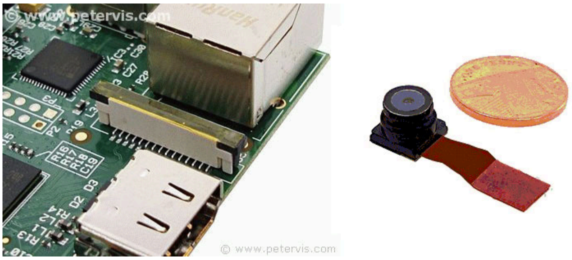


Figure 166: Raspberry CSI camera view [141].

Display Port (DSI)

Raspberry Pi boards 2 to 5 have an LCD Display interface(DSI) port, allowing the user to connect the LCD touch display to the board. The official Raspberry Pi LCD touch display shown in the figure below is 800 x 480 dpi 7“ in size and can be connected to the Raspberry board using the DSI interface. Such an assembly can be used in the projects to display a controlling application view, and the ability to handle fingers and a touchscreen controls the project behaviour. The LCD can be mounted in portrait/landscape orientation, fitting the best user needs.

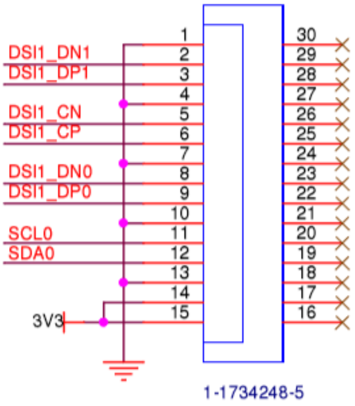


Figure 167: Raspberry DSI display port schematic [142].

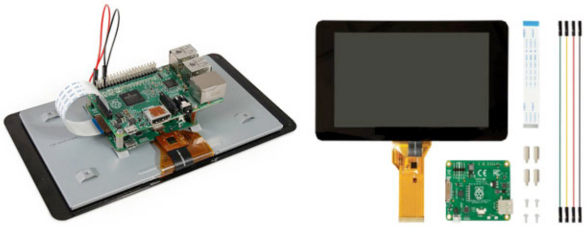


Figure 168: Raspberry DSI LCD display kit [143].

USB and LAN Ports

Raspberry PI models Zero, 1, A+, 2, 3, 4 and 5 contain USB ports (from 1 up to 4), and all but Zero also have a LAN port for TCP/IP network connections. These ports can be used for mouse/keyboard connection or if the software has the appropriate driver installed to handle other USB devices.

Since generation 4, devices are equipped with 2 USB 3.0 ports as in figure 169.

Starting with Raspberry 3B+, the Ethernet port is a gigabit one that can reach up to 1Gbps, theoretically. Prior 3B (including) it is fast Ethernet, 100Mbps.

In RPI 3B+, gigabit Ethernet is connected internally to the USB 2.0 controller with a maximum throughput of about 480Mbps (practical 200Mbsp); thus, the maximum transfer is limited.

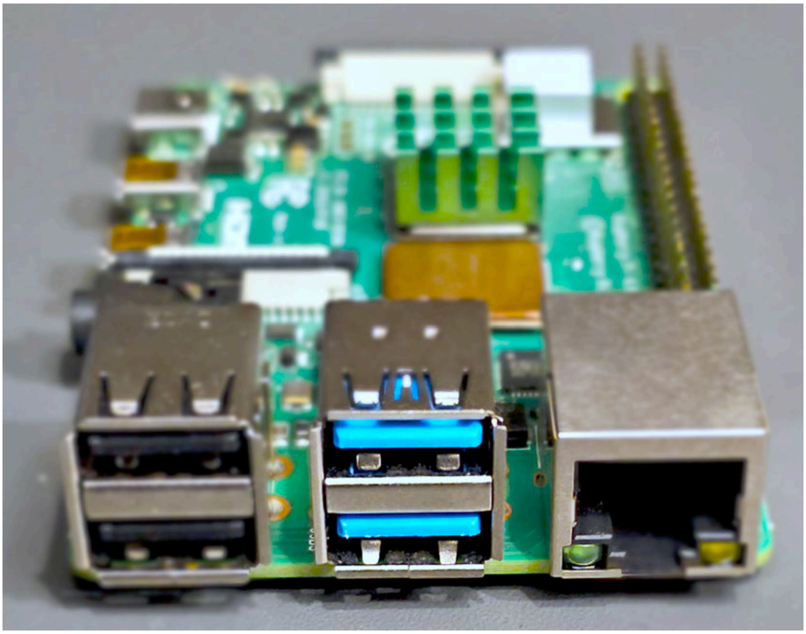


Figure 169: Raspberry 4 LAN/USB ports view

Raspberry Pi Edge Class Devices Hardware Review

The Raspberry Pi Pico is the MCU development board that uses the chip RP2040, designed by Raspberry Pi in 2019.

Hardware

It is intended as a low-cost, low-power device with big computational possibilities and connectivity features. This device is intended to work with constrained power sources, mainly battery-powered. The MCU integrates all features, including 6 banks of RAM, an interrupt controller, DMA, timers, oscillators, I/O, voltage regulator and ROM in a single enclosure.
A compact, 7x7mm chip exposes 26 GPIOs and is one of the most affordable MCUs, estimated at 4 USD/piece only.

Currently, there are 2 types of development boards available: Raspberry Pi Pico and Raspberry Pico W. The last one provides wireless connectivity. It is also possible to have just MCUs (RP2040) as chips to be soldered; thus, third-party development boards are available in the market. A genuine RPi Pico W development board is present in the figure 170.
With a built-in voltage regulator, the input voltage range is wide and starts from 1.8V up to 5.5V.

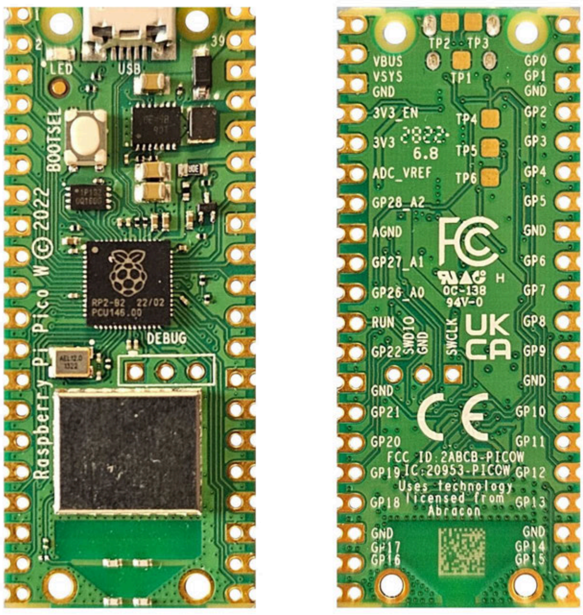


Figure 170: Rapsberry Pi Pico W development board

CPU

The CPU is an ARM Cortex-M0+ (double core) running up to 133 MHz (scalable). It supports DMA. There is no FPU, however. A Nested Vector Interrupt Controller is also present, along with a 24-bit timer. CPU and NIC can be put into the very low power mode.

Memory

RPI Picos have 264kB of internal RAM (SRAM) and 2MB of built-in QSPI flash with the capability for an extension with external one up to 16MB. RAM uses DMA to perform CPU-less transfers.
There is a 16kB ROM that contains bootloaders, USB mass storage UF2 support and utility libraries such as FPU implementation.

Networking

Only the Pico W series includes a built-in radio that is 802.11n (2.4 GHz WiFi) and Bluetooth 5.2.
IoT-specific protocols are supported only with external modules.

Peripherals

The Pico MCU includes a rich set of peripheral interfaces:

- 26 multipurpose GPIO inputs/outputs,
- 2xUART,
- 2xSPI,
- 2xI2C,
- 15 PWM channels,
- 4xADC 12-bit (500ksps) converters where only 3 are usable, with a temperature sensor (for compensation),
- 8 programmable state machines,
- USB 1.1 controller (PHY) with HOST and DEV.

5.2. Sensors and Sensing

A sensor is an element that can turn a physical outer stimulus into an output signal, which can then be used for further analysis, management, or making decisions (figure 171). People also use sensors like eyes, ears and skin to gain information about the outer world and act according to their aims and needs. Sensors can be divided into many categories according to the measured parameter of the environment.

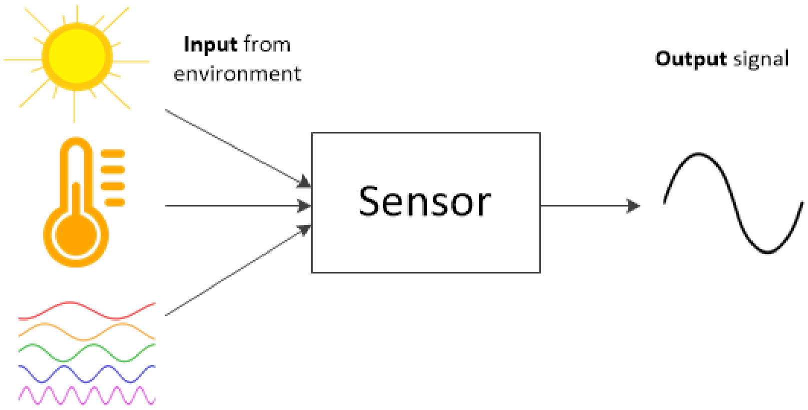


Figure 171: Environment sensing data flow

Usually, every natural phenomenon – temperature, weight, speed, etc. – needs specially customised sensors that can change phenomena into electric signals, usually the voltage, that microprocessors or other devices could use. Sensors can be divided into many groups according to the physical nature of their operations – **touch**, **light**, an **electrical characteristic**, **proximity** and **distance**, **angle**, **environment** and other sensors.

5.2.1. Touch Sensors

Button

A **pushbutton** is an electromechanical sensor that connects or disconnects two points in a circuit when force is applied. The button output discrete value is either *HIGH* or *LOW* (figure 172).

Figure 172: Pushbutton

A **microswitch**, also called a miniature snap-action switch, is an electromechanical sensor that requires very little physical force and uses a tipping-point mechanism. The microswitch has three pins, two of which are connected by default. When the force is applied, the first connection breaks and one of the pins is connected to the third pin (figure 173).

Figure 173: Microswitch

The most common use of a pushbutton is as an input device. Both force solutions can be used as simple object detectors or end switches in industrial devices. The button can be connected to any available digital pin configured as input or input with a pull-up. In the configuration presented in figure 174, a pull-up resistor is connected externally, so enabling it in the microcontroller is unnecessary.

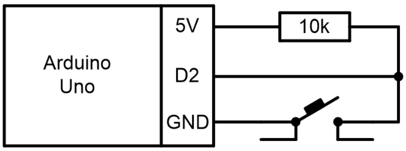


Figure 174: Schematics of Arduino Uno and a push button

An example code:

```
int buttonPin = 2; //Initialization of a push button pin number
int buttonState = 0; //A variable for reading the push button status

void setup() {
  Serial.begin(9600);  //Begin serial communication
  pinMode(buttonPin, INPUT); //Initialize the push button pin as an input
}

void loop() {
  //Read the state of the pin where a button is connected
  //it is LOW if a button is pressed, HIGH otherwise
  //the buttonState variable holds the compliment state of the buttonPin
  buttonState = !digitalRead(buttonPin);
  //Check if the push button is pressed. If it is, the buttonState variable is HIGH
  if (buttonState == HIGH) {
    //Print out text in the console
    Serial.println("The button state variable is HIGH - it is pressed.");
  } else {
    Serial.println("The button state variable is LOW - it is not pressed.");
  }
  delay(10); //Delay in between reads for stability
}
```

Force Sensor

A force sensor predictably changes resistance depending on the applied force to its surface. Force-sensing resistors are manufactured in different shapes and sizes, and they can measure not only direct force but also tension, compression, torsion and other mechanical forces. Because the force sensor changes its resistance linearly, it should be connected to the analogue input. Connecting another resistor to form the voltage divider is also required, as shown in the figures 175 and 176. The internal ADC of the microcontroller measures the voltage.
Force sensors are used as control buttons, object presence detectors, or to determine weight in electronic scales.

Figure 175: Force sensitive resistor (FSR)

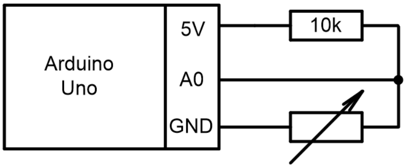


Figure 176: The voltage is measured by applying and measuring constant voltage to the sensor

An example code:

```
//Force Sensitive Resistor (FSR) is connected to the analogue 0 pin
int fsrPin = A0;
//The analog reading from the FSR resistor divider
int fsrReading;

void setup(void) {
  //Begin serial communication
  Serial.begin(9600);
  //Initialize the FSR analogue pin as an input
  pinMode(fsrPin, INPUT);
}

void loop(void) {
  //Read the resistance value of the FSR
  fsrReading = analogRead(fsrPin);
  //Print
  Serial.print("Analog reading = ");
  Serial.println(fsrReading);
  delay(10);
}
```

Capacitive Sensor

Capacitive sensors are a range of sensors that use capacitance to measure changes in the surrounding environment. A capacitive sensor is a capacitor charged with a certain amount of current until the threshold voltage is reached. A human finger, liquids or other conductive or dielectric materials that touch the sensor can influence the sensor's charge time and voltage level. Measuring charge time and voltage level gives information about changes in the environment. Ready-to-use sensors include an electronic element that performs measurements and returns digital information at the output so that they can be connected directly to a digital input pin.
Capacitive sensors are input devices and can measure proximity, humidity, fluid level and other physical parameters or serve as an input for electronic device control. Sample sensor and its connection are presented in the figures 177 and 178.

Figure 177: Touch button module

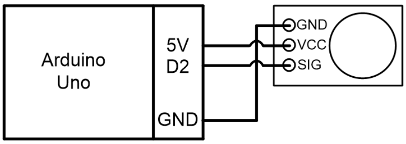


Figure 178: Arduino and capacitive sensor schematics

```
//Capacitive sensor is connected to the digital 2 pin
int touchPin = 2;

//The variable that stores digital value read from the sensor
boolean touchReading = LOW;
//The variable that stores the previous state of the sensor
boolean lastState = LOW;

void setup() {
  //Begin serial communication
  Serial.begin(9600);
  //Initialize the capacitive sensor analogue pin as an input
  pinMode(touchPin, INPUT);
}

void loop() {
  //Read the digital value of the capacitive sensor
  touchReading = digitalRead(touchPin);
  //If the new touch has appeared
  if (currentState == HIGH && lastState == LOW){
    Serial.println("Sensor is pressed");
    delay(10); //short delay
  }
  //Save the previous state to see relative changes
  lastState = currentState;
}
```

Most of the buttons and switches are of simple construction, so they are subject to a debouncing, as a single press or release of the switch may trigger many changes in the signal (not just a single swap from *LOW* to *HIGH* or opposite). This is because applying force and moving connectors is imperfect and may involve vibration and twinkling. We discuss this problem in the programming patterns chapter.

5.2.2. Light Sensors

Photoresistor

A photoresistor is a sensor that perceives light waves from the environment. The resistance of the photoresistor is changing depending on the intensity of light. The higher the intensity of the light, the lower the sensor's resistance. A light level is determined by applying a constant voltage through the resistor to the sensor, forming a voltage divider, and measuring the resulting voltage. Photoresistors are cheap, but the resistance is influenced by temperature and changes slowly, so they are used in applications where speed and accuracy are not crucial.
Photoresistors are often utilised in energy-effective street lighting control.
A symbol, sample photoresistor, and connection circuit are present in figures 179, 180 and 181.

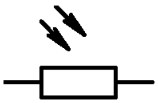


Figure 179: A photoresistor symbol

Figure 180: A photoresistor

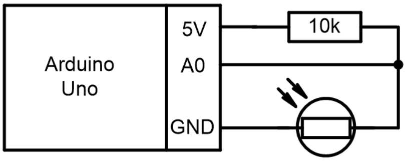


Figure 181: Arduino and photoresistor sensor schematics

As shown in the figure 181, the photoresistor connected gives a lower voltage level while the light is more intense. Results can be read with the following example code. The value will be just a number not expressed in any units, e.g. Lux. To express light intensity in Luxes, additional calculations must be encoded in the program.

```
//Define an analog A0 pin for photoresistor
int photoresistorPin = A0;
//The analogue reading from the photoresistor
int photoresistorReading;

void setup()
{
    //Begin serial communication
    Serial.begin(9600);
    //Initialise the analogue pin of a photoresistor as an input
    pinMode(photoresistorPin, INPUT);
}

void loop()
{
    //Read the value of the photoresistor
    photoresistorReading = analogRead(photoresistorPin);
    //Print out the value of the photoresistor reading to the serial monitor
    Serial.println(photoresistorReading);
    delay(10); //Short delay
}
```

Photodiode

A photodiode is a sensor that converts light energy into electrical current. A current in the sensor is generated by exposing a p-n junction of a semiconductor to the light. Information about the light intensity can be determined by measuring a voltage level. Photodiodes react to changes in light intensity very quickly, so they can be used as receivers of light-based data transmission systems (e.g. fibre data communication). Solar cells are just large photodiodes. A symbol, sample photodiode and connection circuit are present in figures 182, 183 and 184.
Photodiodes are used as precise light-level sensors, receivers for remote control, electrical isolators (optocouplers), and proximity detectors.

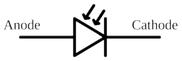


Figure 182: A photodiode symbol

Figure 183: A photodiode

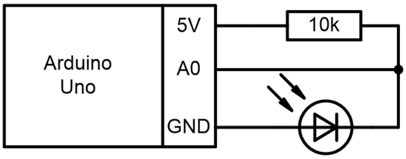


Figure 184: Arduino and photodiode sensor schematics

Although the photodiode can generate current, the schematic in figure 184 shows its connection similar to the photoresistor in the previous example. In such a circuit, the photodiode changes its current according to a change in light intensity, resulting in the voltage change at the microcontroller's analogue input. As in the example for a photoresistor, the higher the light intensity, the lower the voltage. An example code:

```
//Define an analog A0 pin for photodiode
int photodiodePin = A0;
//The analogue reading from the photodiode
int photodiodeReading;

void setup()
{
    //Begin serial communication
    Serial.begin(9600);
    //Initialise the analogue pin of a photodiode as an input
    pinMode(photodiodePin, INPUT);
}

void loop()
{
    //Read the value of the photodiode
    photodiodeReading = analogRead(photodiodePin);
    //Print out the value of the photodiode reading to the serial monitor
    Serial.println(photodiodeReading);
    delay(10); //Short delay
}
```

Phototransistor

The phototransistor is a typical bipolar transistor with a transparent enclosure that exposes the base-emitter junction to light. In a bipolar transistor, the current that passes through the collector and emitter depends on the base current. In the phototransistor,

the collector-emitter current is controlled with light. A phototransistor is slower than a photodiode but can conduct more current; additionally, it amplifies the incoming signal. In specific conditions, if the light is completely off or intense enough to make the output current maximal, a phototransistor can be considered a light-controlled electronic switch (e.g. in optocouplers, which are usually connected to digital inputs of the microcontroller and provide physical separation between devices).
Phototransistors are used as optical switches, proximity sensors and electrical isolators.
A symbol, sample phototransistor device, and circuit are present in figures 185, 186 and 187.

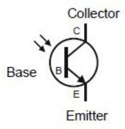


Figure 185: A phototransistor symbol

Figure 186: An phototransistor

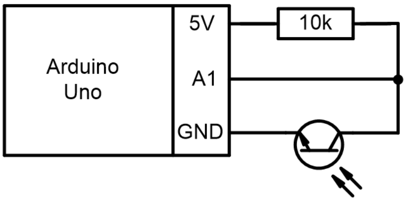


Figure 187: Arduino and phototransistor schematics

An example code:

```
//Define an analog A1 pin for phototransistor
int phototransistorPin = A1;
//The analogue reading from the phototransistor
int phototransistorReading;

void setup()
{
    //Begin serial communication
    Serial.begin(9600);
    //Initialise the analogue pin of a phototransistor as an input
    pinMode(phototransistorPin, INPUT);
}

void loop()
{
    //Read the value of the phototransistor
    phototransistorReading = analogRead(phototransistorPin);
    //Print out the value of the phototransistor reading to the serial monitor
    Serial.println(phototransistorReading);
    delay(10); //short delay
}
```

5.2.3. Optical Sensors

Optocoupler

An optocoupler is a device that combines light-emitting and receiving devices in one package. Mainly, it combines the infrared light-emitting diode (LED) and a phototransistor.
There are three main types of optocouplers:

- an **optocoupler of a closed pair configuration** is enclosed in the dark resin and is used to transfer signals using light. This type of optocoupler is not a sensor itself but is used for ensuring electrical isolation between two circuits;
- a **slotted optocoupler** has an open space between the light source and the sensor; external objects can obstruct light and thus can influence the sensor signal. It can be used to detect the presence of flat objects, measure rotation speed, vibrations or serve as a bounce-free switch;
- a **reflective pair configuration**, the light signal is perceived as a reflection from the object's surface. This configuration is used for proximity detection, surface colour detection and tachometer.

A symbol, sample optocoupler and its connection to the microcontroller are present in figures 188, 189 and 190.

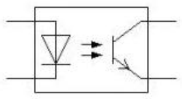


Figure 188: An optocoupler symbol

Figure 189: ELITR9909 reflective optocoupler sensor

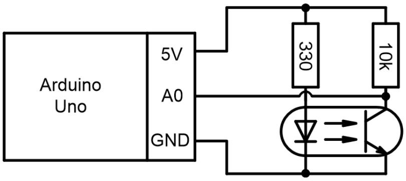


Figure 190: Arduino Uno and optocoupler schematics

An example code:

```
int optoPin = A0;    //Initialize an analogue A0 pin for optocoupler
int optoReading;     //The analogue value reading from the optocoupler

int objecttreshold = 1000; //Object threshold definition
int whitetreshold = 150;   //White colour threshold definition

void setup ()
{
  //Begin serial communication
  Serial.begin(9600);
  //Initialise the analogue pin of the optocoupler as an input
  pinMode(optoPin, INPUT);
}

void loop ()
{
  optoReading = analogRead(optoPin); //Read the value of the optocoupler
  Serial.print ("The reading of the optocoupler sensor is: ");
  Serial.println(optoReading);

  //When the reading value is lower than the object threshold
  if (optoReading < objecttreshold) {
    Serial.println ("There is an object in front of the sensor!");
    //When the reading value is lower than the white threshold
    if (optoReading < white threshold) {
      Serial.println ("Object is in white colour!");
    } else { //When the reading value is higher than the white threshold
      Serial.println ("Object is in dark colour!");
    }
  }
  else { //When the reading value is higher than the object threshold
    Serial.println ("There is no object in front of the sensor!");
  }
  delay(500); //Short delay
}
```

Colour Sensor

This type of sensor gives information about the colour of the light illuminating the sensor surface. Because computers often use RGB (red, green, blue) colour schemes, the sensor returns three values representing the intensity of three components. Colour sensors usually contain white LEDs to illuminate the surface, which colour should be distinguished by them. The colour sensor uses an SPI or TWI interface to send readings. Some models of colour sensors include an additional gesture detector which recognises simple gestures (up, down, left, right).
The sample device is present in figure 191 and the connection circuit in figure 192.

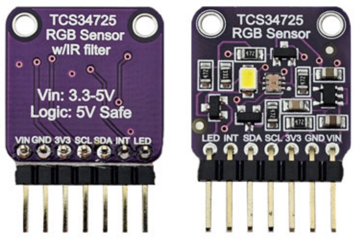


Figure 191: TCS34725 RGB colour sensor module

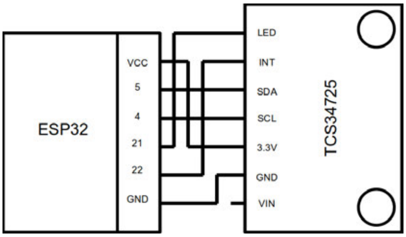


Figure 192: Connection of the TCS34725 and microcontroller

```
#include <Wire.h>
#include "Adafruit_TCS34725.h"

// Example code for the TCS34725 library by Adafruit

// Sensor class
Adafruit_TCS34725 rgb_sensor = Adafruit_TCS34725();

void setup(void) {
  Serial.begin(9600);
  Wire.begin(5,4);             //SCL SDA
  pinMode(21, OUTPUT);         //Pin 21 controls LED
  digitalWrite(21,LOW);        //Turn off onboard LED
  if (rgb_sensor.begin()) {   //Initialise RGB sensor
    Serial.println("RGB sensor present");
  } else {
    Serial.println("No TCS34725 found");
    while (1);
  }
}

void loop(void) {
  uint16_t r, g, b, unfiltered, lux;

  rgb_sensor.getRawData(&r, &g, &b, &unfiltered);
                             //read RGB and unfiltered light intensity
  lux = rgb_sensor.calculateLux(r, g, b);
                             //calculate illuminance in Lux

  Serial.print("Lux: ");     //print calculated Lux value
  Serial.print(lux, DEC);
  Serial.print(" - ");

  Serial.print("R: ");       //print red component value
  Serial.print(r, DEC);
  Serial.print(" ");

  Serial.print("G: ");       //print green component value
  Serial.print(g, DEC);
  Serial.print(" ");

  Serial.print("B: ");       //print blue component value
  Serial.print(b, DEC);
  Serial.print(" ");

  Serial.print("C: ");       //print unfiltered sensor value
  Serial.print(unfiltered, DEC);
  Serial.println(" ");

  delay(1000);
}
```

5.2.4. Electrical Characteristic Sensors

Electrical characteristic sensors measure the voltage and amperage of the electric current. When the voltage and current sensors are used concurrently, the device's consumed power can be determined. Electrical characteristic sensors can also determine whether the device's circuit is working correctly. Different sensor circuits must be used to measure direct current (DC) and alternating current (AC). For safety reasons, the parameters of the mains must be measured using transformers.

Potentiometer

A potentiometer is a type of resistor whose resistance can be adjusted using a mechanical lever. The device consists of three terminals. The resistor between the first and the third terminal has a fixed value, but the second terminal is connected to the lever. Whenever the lever is turned, a slider of the resistor is moved; it changes the resistance between the second terminal and side terminals. Variable resistance causes the change of the voltage, which can be measured to determine the position of the lever. Thus, the potentiometer output is an analogue value.
Potentiometers are commonly used as a control level, for example, a volume level for the sound and joystick position. They can also be used to determine the angle in feedback loops with motors, such as servo motors. The potentiometer symbol is present in figure 193, a device in figure 194 and a connection to the Arduino board in figure 195.

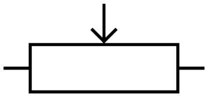


Figure 193: A symbol of a potentiometer

Figure 194: A potentiometer

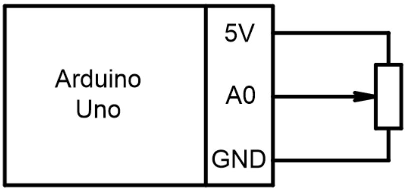


Figure 195: Arduino and potentiometer circuit

An example code:

```
//Potentiometer sensor output is connected to the analogue A0 pin
int potentioPin = A0;
//The analogue reading from the potentiometer output
int potentioReading;

void setup(void) {
  //Begin serial communication
  Serial.begin(9600);
  //Initialize the potentiometer analogue pin as an input
  pinMode(potentioPin, INPUT);
}

void loop(void) {
  //Read the analogue value of the potentiometer sensor
  potentioReading = analogRead(potentioPin);
  Serial.print("Potentiometer reading = "); //Print out
  Serial.println(potentioReading);
  delay(10);
}
```

Voltage Sensor

A voltage sensor is a device or circuit for voltage measurement. A simple DC (direct current) voltage sensor consists of a voltage divider circuit with an optional amplifier for a tiny voltage measure. For measuring the AC (alternating current), the input is connected to the rectifier diode or bridge to rectify AC to DC and a capacitor to flatten the voltage. The resulting voltage can be measured with an analogue digital converter of the

microcontroller. For safety, while measuring the mains voltage, an optoelectrical isolator should be added at the output, or a transformer should lower the voltage at the input.
A voltage sensor can detect a power failure and measure if the voltage is in the range required. IoT applications include monitoring appliances, power lines, and power supplies. Sample voltage sensor module is present in figure 196 and schematic connection to the Arduino Uno in figure 197.

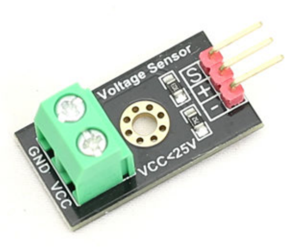


Figure 196: Voltage sensor module 0–25 V

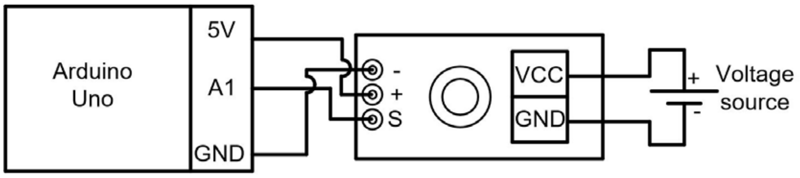


Figure 197: Arduino and voltage sensor schematics

The example code:

```
//Define an analogue A1 pin for voltage sensor
int voltagePin = A1;
//The result of the analogue reading from the voltage sensor
int voltageReading;

float vout = 0.0;
float vin = 0.0;
float R1 = 30000.0; //  30 kΩ resistor
float R2 = 7500.0; //  7.5 kΩ resistor

void setup()
{
    //Begin serial communication
    Serial.begin(9600);
    //Initialize the analogue pin as an input
    pinMode(voltagePin, INPUT);
}

void loop()
{
    //Read the value of the voltage sensor
    voltageReading = analogRead(voltagePin);
    vout = (voltageReading * 5.0) / 1024.0;
    vin = vout / (R2/(R1+R2));

    Serial.print("Voltage is: ");
    //Print out the value of the voltage to the serial monitor
    Serial.println(vin);
    delay(10); //Short delay
}
```

Current Sensor

A current sensor is a device or a circuit for current measurement. A simple DC sensor consists of a high-power resistor with low resistance. The current value is obtained by measuring the voltage on the resistor and applying a formula derived from Ohm's law. Other non-invasive measurement methods involve hall effect sensors for DC and AC and inductive coils (current transformer) for AC.
Current sensors determine the power consumption and detect whether the device is turned on or shorted.
Sample current sensor modules are present in figures 198 and 199, and schematic connection to the Arduino Uno in figure 200

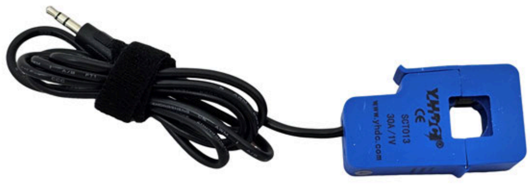


Figure 198: Current transformer module for AC

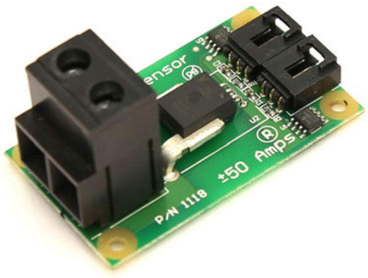


Figure 199: Analogue current meter module 0–50 A

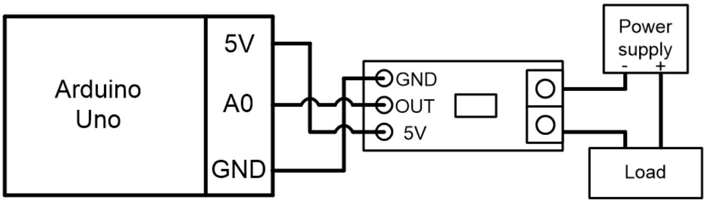


Figure 200: Arduino and current sensor module schematics

The example code:

```
//Define an analogue A0 pin for current sensor
const int currentPin = A0;
//Scale factor of the sensor use 100 for 20 A Module and 66 for 30 A Module
int mVperAmp = 185;
int currentReading;
int ACSoffset = 2500;
double voltage;
double current;

void setup(){
 Serial.begin(9600);
}

void loop(){

 currentReading = analogRead(currentPin);
 Voltage = (currentReading / 1024.0) * 5000; //Gets you mV
 Current = ((Voltage - ACSoffset) / mVperAmp); //Calculating current value

 Serial.print("Raw Value = " ); //Shows pre-scaled value
 Serial.print(currentReading);
 Serial.print("\t Current = "); //Shows the voltage measured
 //The '3' after current allows to display 3 digits after the decimal point
 Serial.println(Current,3);
 delay(1000); //Short delay
```

5.2.5. Proximity and Distance Sensors

Infrared Sensor

An infrared (IR) proximity sensor detects objects and measures distance without physical contact. IR sensor consists of an infrared emitter, a receiving sensor or array of sensors and a signal processing logic. The output of a sensor differs depending on the type – simple proximity detection sensor outputs *HIGH* or *LOW* level when an object is in its sensing range, but sensors which can measure distance output an analogue signal or use some communication protocol, like I2C to send sensor measuring results. IR sensors are used in robotics to detect obstacles located a few millimetres to several meters from the sensor and in mobile phones to help detect accidental screen touching.\\Sample IR sensor device is present in figure 201 and its connection to the microcontroller in figure 202.

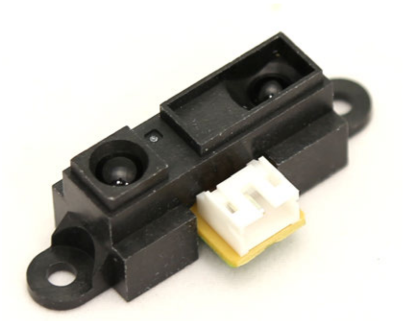


Figure 201: Distance Sensor GP2Y0A21YK0F

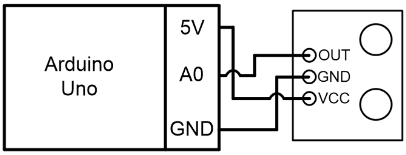


Figure 202: Arduino and IR proximity sensor circuit

An example code:

```
int irPin = A0;  //Define an analogue A0 pin for IR sensor
int irReading;  //The result of an analogue reading from the IR sensor

void setup()
{
    //Begin serial communication
    Serial.begin(9600);
    //Initialize the analogue pin of an IR sensor as an input
    pinMode(irPin, INPUT);
}

void loop()
{
    //Read the value of the IR sensor
    irReading = analogRead(irPin);
    //Print out the value of the IR sensor reading to the serial monitor
    Serial.println(irReading);
    delay(10); //Short delay
}
```

Ultrasonic Sensor

The ultrasonic sensor measures the distance to objects by emitting a short ultrasound sound pulse and measuring its returning time. The sensor consists of an ultrasonic emitter and receiver; sometimes, they are combined into a single device for emitting and receiving. Ultrasonic sensors can measure greater distances and cost less than infrared sensors but are more imprecise and interfere with each other's measurements if more than one is used. Simple sensors have a trigger pin and an echo pin; when the trigger pin is set high for a small amount of time, ultrasound is emitted, and on the echo pin, response time is measured. Ultrasonic sensors are used in car parking sensors and robots

for proximity detection. A sample ultrasonic sensor is present in figure 203 and a circuit in figure 204.
Examples of IoT applications are robotic obstacle detection and room layout scanning.

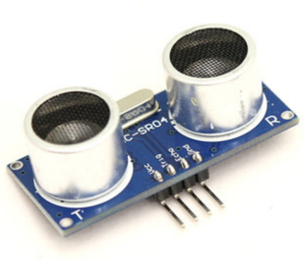


Figure 203: Ultrasonic proximity sensor HC-SR04

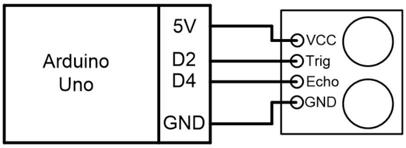


Figure 204: Arduino and ultrasound proximity sensor circuit

An example code:

```
int trigPin = 2;  //Define a trigger pin D2
int echoPin = 4;  //Define an echo pin D4

void setup()
{
    Serial.begin(9600); //Begin serial communication
    pinMode(trigPin, OUTPUT); //Set the trigPin as an Output
    pinMode(echoPin, INPUT); //Set the echoPin as an Input
}

void loop()
{
    digitalWrite(trigPin, LOW);  //Clear the trigPin
    delayMicroseconds(2);

    //Set the trigPin on HIGH state for 10 μs
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);

    //Read the echoPin, return the sound wave travel time in microseconds
    duration = pulseIn(echoPin, HIGH);
    //Calculating the distance
    distance = duration*0.034/2;

    //Printing the distance on the Serial Monitor
    Serial.print("Distance: ");
    Serial.println(distance);
}
```

Motion Detector

The motion detector is a sensor that detects moving objects, primarily people. Motion detectors use different technologies, like passive infrared sensors, microwaves and the Doppler effect, video cameras, and ultrasonic and IR sensors. Passive IR (PIR) sensors are the simplest motion detectors that sense people by detecting changes in IR radiation emitted through the skin. When the motion is detected, the output of a motion sensor is a digital *HIGH/LOW* signal.
Motion sensors are used in security alarm systems, automated lights and door control. As an example in IoT, the PIR motion sensor can detect motion in security systems in a house or any building.\\Sample PIR sensor is present in 205 and connection schematic in figure 206.

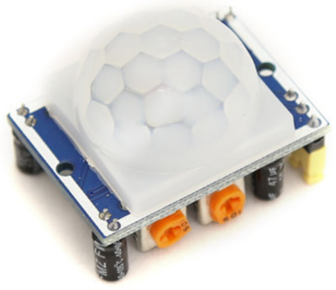


Figure 205: PIR motion sensor

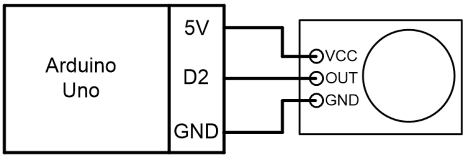


Figure 206: Arduino and PIR motion sensor circuit

An example code:

```
//Passive Infrared (PIR) sensor output is connected to the digital 2 pin
int pirPin = 2;
//The digital reading from the PIR output
int pirReading;

void setup(void) {
  //Begin serial communication
  Serial.begin(9600);
  //Initialize the PIR digital pin as an input
  pinMode(pirPin, INPUT);
}

void loop(void) {
  //Read the digital value of the PIR motion sensor
  pirReading = digitalRead(pirPin);
  //Print out
  Serial.print("Digital reading = ");
  Serial.println(pirReading);
  if(pirReading == HIGH) {  //Motion was detected
    Serial.println("Motion Detected");
  }

  delay(10);
}
```

Fluid Level Sensor

A level sensor detects the level of fluid or fluidised solid. Level sensors can be divided into two groups:

- **continuous level sensors** that can detect the exact position of the fluid. Proximity sensors, like ultrasonic or infrared, are usually used for level detection. Capacitive sensors can also be used by recording the changing capacitance value depending on the fluid level. The output can be either analogue or digital value;
- **point-level sensors** can detect whether a fluid is above or below the sensor. A membrane with air pressure or changes in conductivity or capacitance can be used for level detection as a floating or mechanical switch. The output is usually a digital value that indicates *HIGH* or *LOW* value.

Fluid level sensors can be used for smart waste management, measuring tank levels, diesel fuel gauging, liquid assets inventory, chemical manufacturing, high or low-level alarms, and irrigation control.
Sample level sensor is present in figure 207 and connection schematic in figure 208.

Figure 207: Liquid level sensor

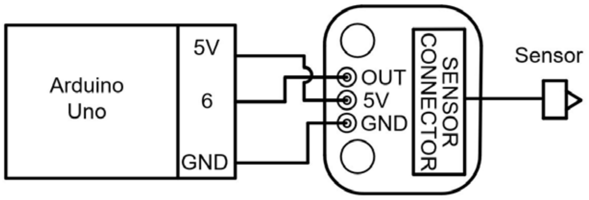


Figure 208: Arduino Uno and liquid level sensor schematics

An example code:

```
int levelPin = 6; //Liquid level sensor output is connected to the digital 6 pin
int levelReading; //Stores level sensor detection reading

void setup(void) {
  Serial.begin(9600);   //Begin serial communication
  pinMode(levelPin, INPUT); //Initialize the level sensor pin as an input
}
```

```
void loop(void) {
  levelReading = digitalRead(levelPin); //Read the digital value of the level sensor
  Serial.print("Level sensor value: "); //Print out
  Serial.println(levelReading);
  delay(10); //Short delay
}
```

5.2.6. Angle & Orientation Sensors

The Inertial Measurement Unit (IMU)

An IMU is an electronic device that consists of an accelerometer, gyroscope, and sometimes a magnetometer. The combination of these sensors returns the object's orientation in 3D space. IMU sensors can present the object's current position and movement, expressed with at most six values called the DOF (Degrees Of Freedom). Three values represent the linear movements that the accelerometer can measure:

- moving forward/backwards,
- moving left/right,
- moving up/down.

Another three values present the rotation in three axes that can be measured by gyroscope:

- roll side to side,
- pitch forward and backwards,
- yaw left and right.

A **gyroscope** is a sensor that measures the angular velocity. A microelectromechanical system (MEMS) technology integrates the sensor into the chip. The sensor output can be analogue or digital, using I2C or SPI interface. Gyroscope microchips can vary in the number of axes they can measure. The available number of axes is 1, 2 or 3 axes in the gyroscope. A gyroscope is commonly used with an accelerometer to determine the device's orientation, position and velocity precisely. Gyroscope sensors are used in aviation, navigation and motion control.

An **accelerometer** measures the acceleration of the object. The sensor uses microelectromechanical system (MEMS) technology, where capacitive plates are attached to springs. When acceleration force is applied to the plates, the capacitance is changed; thus, it can be measured. Accelerometers can have 1 to 3 axes. The 3-axis accelerometer can detect the device's orientation, shake, tap, double tap, fall, tilt, motion, positioning, shock or vibration. Outputs of the sensor are usually digital interfaces like I2C or SPI. The accelerometer is often used with a gyroscope to measure the object's movement and orientation in space precisely.
Accelerometers measure objects' vibrations, including cars, industrial devices, and buildings, and detect volcanic activity. IoT applications can also be used for accurate motion detection for medical and home appliances, portable navigation devices,

augmented reality, smartphones and tablets.

A **magnetometer** is a sensor that can measure the device's orientation to the Earth's magnetic field. A magnetometer is used as a compass in outdoor navigation for mobile devices, robots, and quadcopters.

Different elements allow measuring linear accelerations, angular accelerations, and magnetic fields in three axes. There exist elements that combine two (called 6-axis or 6-DOF) or all (9-axis, 9-DOF) measurement units. Popular integrated circuits are MPU6050 (3-axes gyro + 3-axes accelerometer, figure 209), MPU9250 (3-axes gyro + 3-axes accelerometer + 3-axes compass, figure 210), and BNO055 (3-axes gyro + 3-axes accelerometer + 3-axes magnetometer, figure 211). All of them can be programmed in an Arduino environment using dedicated libraries.
The latter automatically calculates additional information like gravity vector and absolute orientation expressed as an Euler vector or a quaternion. The sample connection circuit for the BNO055 sensor is present in figure 212. Note, figure 212 does not present pull-up resistors on the I2C bus as they are integrated into the development boards.

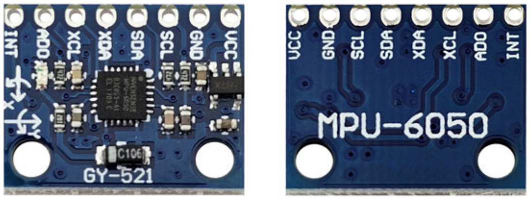


Figure 209: IMU MPU6050 module

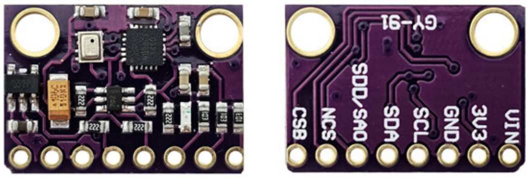


Figure 210: IMU MPU9250 module

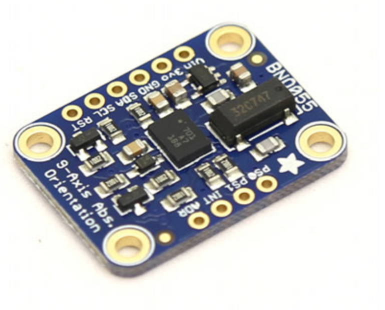


Figure 211: IMU BNO055 module

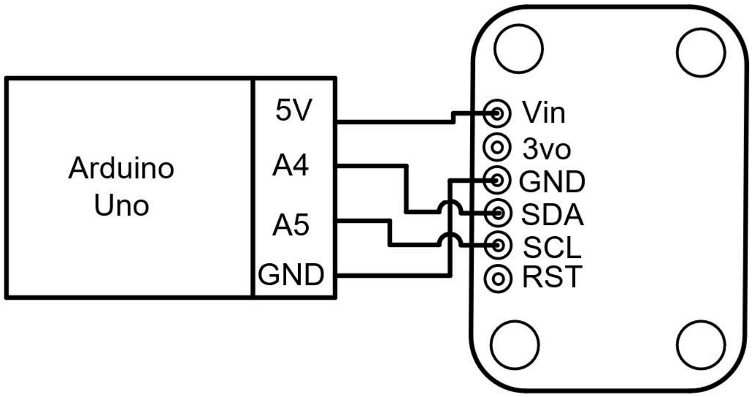


Figure 212: Arduino Uno and IMU BNO055 module schematics

The example code:

```
//Library for I2C communication
#include <Wire.h>
//Downloaded from https://github.com/adafruit/Adafruit_Sensor
#include <Adafruit_Sensor.h>
//Downloaded from https://github.com/adafruit/Adafruit_BNO055
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
Adafruit_BNO055 bno = Adafruit_BNO055(55);
void setup(void)
{
  bno.setExtCrystalUse(true);
}
void loop(void)
{
  //Read sensor data
  sensors_event_t event;
  bno.getEvent(&event);
  //Print X, Y And Z orientation
  Serial.print("X: ");
  Serial.print(event.orientation.x, 4);
  Serial.print("\tY: ");
  Serial.print(event.orientation.y, 4);
  Serial.print("\tZ: ");
  Serial.print(event.orientation.z, 4);
  Serial.println("");
  delay(100);
}
```

Most MEMS devices present built-in inaccuracy. For this reason, gyros and accelerometers should be calibrated before use to calculate their so-called offset, an average error they present (in each axis separately). Later, this error is used to calculate a correction factor applied during regular operation. Sample MPU6050 library along with calibration code can be found in the Github repository [144].

Similar problem is present in the case of the magnetometers: the surrounding environment can impact readings; thus, they require calibration that can be achieved by recording the minimum and maximum values during rotation in every axis. This process is common when using a drone in a new location.

5.2.7. Environment Sensors

B E

Temperature Sensor

A temperature sensor is a device used to determine the temperature of the surrounding environment. Most temperature sensors work on the principle that the material's resistance changes depending on its temperature. The most common temperature sensors are:

- **thermocouple** – consists of two junctions of dissimilar metals,
- **thermistor** – includes the temperature-dependent resistor,
- **resistive temperature detector** – is made of a pure metal coil.

The main difference between sensors is the measured temperature range, precision and response time. Temperature sensor usually outputs the analogue value, but some existing sensors have a digital interface [145]. The thermistor can have a positive (PTC) or negative (NTC) thermal coefficient. For PTC, resistance rises with rising temperature, while resistance decreases in higher temperatures for NTC. An analogue thermistor must calculate the value read if the result should be presented in known units. Digital temperature sensors usually express the result in Celsius degrees or other units.

Temperature sensors are most commonly used in environmental monitoring devices and thermoelectric switches. In IoT applications, the sensor can be used for greenhouse temperature monitoring, warehouse temperature monitoring to avoid freezing, fire suppression systems and tracking the temperature of the soil, water and plants. The sample temperature sensor is present in figure 213 and connection schematic in 214.

Figure 213: A thermistor

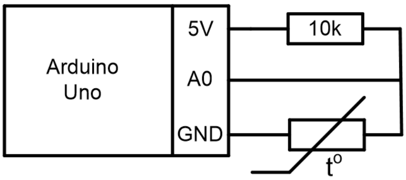


Figure 214: Arduino and thermistor circuit

An example code:

```
//Thermistor sensor output is connected to the analogue A0 pin
int thermoPin = 0;
//The analogue reading from the thermistor output
int thermoReading;

void setup(void) {
  //Begin serial communication
  Serial.begin(9600);
  //Initialize the thermistor analogue pin as an input
  pinMode(thermoPin, INPUT);
}

void loop(void) {
  //Read the analogue value of the thermistor sensor
  thermoReading = analogRead(thermoPin);
  Serial.print("Thermistor reading = "); //Print out
  Serial.println(thermoReading);
  delay(10);
}
```

Digital Temperature Sensor

Digital temperature sensors automatically convert the temperature reading into some known unit, e.g. Celsius, Fahrenheit or Kelvin Degrees. Digital thermometers use one of the popular communication links. An example of a digital thermometer is DS18B20 by Dallas Semiconductors (figures 215 and 216). It uses a 1-Wire communication protocol; a sample schematic is present in the figure 217.

Figure 215: DS18B20 temperature sensor

Figure 216: DS18B20 temperature sensor, waterproof version

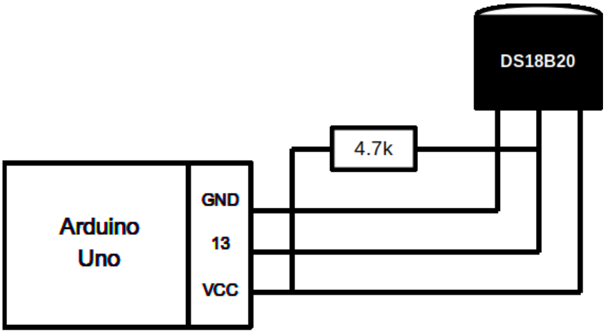


Figure 217: DS18B20 circuit (One Wire on pin 13)

```
#include <OneWire.h>           //library for 1-Wire protocol
#include <DallasTemperature.h> //library for DS18B20 digital thermometer

const int SENSOR_PIN = 13;   //DS18B20 pin

OneWire oneWire(SENSOR_PIN); //oneWire class
DallasTemperature tempSensor(&oneWire);
                             //connect oneWire to DallasTemperature library

float tempCelsius;           //temperature in Celsius degrees

void setup()
```

```
{
  Serial.begin(9600);         //initialize serial port
  tempSensor.begin();         //initialize DS18B20
}

void loop()
{
  tempSensor.requestTemperatures();
                              //command to read temperatures
  tempCelsius = tempSensor.getTempCByIndex(0);
                              //read temperature (in Celsius)

  Serial.print("Temp: ");
  Serial.print(tempCelsius); //print the temperature
  Serial.println(" C");

  delay(1000);
}
```

Digital temperature sensors using 1-Wire are handy when communication speed is not crucial. 1-Wire offers longer distance and less cabling compared, e.g. to I2C and SPI.

Humidity Sensor

A humidity sensor (hygrometer) is a sensor that detects the amount of water or water vapour in the environment. The most common principle of air humidity sensors is the change of capacitance or resistance of materials that absorb moisture from the atmosphere. Soil humidity sensors measure the resistance between the two electrodes. Soluble salts and water amounts influence the resistance between electrodes in the soil. The output of a humidity sensor is an analogue signal value or digital value sent with some popular protocols [146]. A DHT11 (temperature+humidity) sensor is present in figure 218 and its connection to the microcontroller in 219.
IoT applications include monitoring humidors, greenhouse humidity, agriculture, art galleries and museum environments.

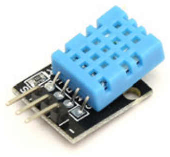


Figure 218: Temperature and humidity sensor module

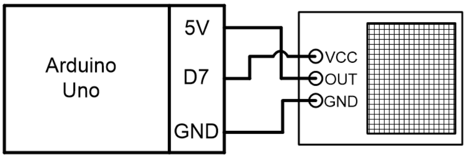


Figure 219: Arduino Uno and humidity sensor schematics

An example code [147]:

```
#include <dht.h>

dht DHT;

#define DHT_PIN 7

void setup(){
  Serial.begin(9600);
}

void loop()
{
  int chk = DHT.read11(DHT_PIN);
  Serial.print("Humidity = ");
  Serial.println(DHT.humidity);
  delay(1000);
}
```

DHT sensors use their own One Wire communication standard that is incompatible with standard 1-Wire, even if it uses a similar connection schema.

Sound Sensor

A sound sensor is a sensor that detects vibrations in a gas, liquid or solid environment. At first, the sound wave pressure makes mechanical vibrations, which transfer to changes in capacitance, electromagnetic induction, light modulation or piezoelectric generation to create an electric signal. The electrical signal is then amplified to the required output levels. Sound sensors can record sound and detect noise and its level.
Sound sensors are used in drone detection, gunshot alert, seismic detection and vault safety alarms.
Sample digital sound sensor is present in figure 220 and its application with Arduino in figure 221.

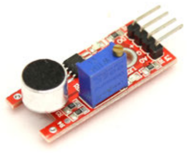


Figure 220: Digital sound detector sensor module

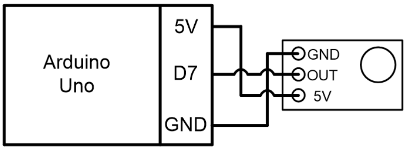


Figure 221: Arduino Uno and sound sensor schematics

An example code:

```
//Sound sensor output is connected to the digital 7 pin
int soundPin = 7;
//Stores sound sensor detection readings
int soundReading = HIGH;

void setup(void) {
  //Begin serial communication
  Serial.begin(9600);
  //Initialize the sound detector module pin as an input
  pinMode(soundPin, INPUT);
}

void loop(void) {
  //Read the digital value to determine whether the sound has been detected
  soundReading = digitalRead(soundPin);
  if (soundPin==LOW) { //When sound detector detected the sound
    Serial.println("Sound detected!"); //Print out
  } else { //When the sound is not detected
    Serial.println("Sound not detected!"); //Print out
  }
  delay(10);
}
```

Chemical and Gas Sensor

Gas sensors are a group that can detect and measure the concentration of certain gasses in the air. The working principle of electrochemical sensors is to absorb the gas and create current from an electrochemical reaction. For process acceleration, a heating element can be used. For each type of gas, different kind of sensor needs to be used. Multiple types of gas sensors can also be combined in a single device. The single gas sensor output is an analogue signal, but devices with various sensors have a digital interface. The smoke or air pollution sensors usually use LED or laser that emits light and a detector normally shaded from the light. If there are particles of smoke or polluted air inside the sensor, the light is reflected by them, which can be observed by the detector.
Gas sensors are used for safety devices, air quality control, and manufacturing equipment. IoT applications include air quality control management in smart buildings and smart cities or toxic gas detection in sewers and underground mines.\\MQ-7 Carbon

Monoxide detector is present in figure 222 and its connection using analogue signal in figure 223.

Figure 222: MQ-7 gas sensor

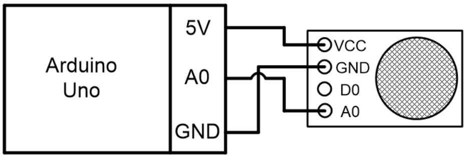


Figure 223: Arduino Uno and MQ2 gas sensor schematics

An example code:

```
int gasPin = A0; //Gas sensor output is connected to the analog A0 pin
int gasReading; //Stores gas sensor detection reading

void setup(void) {
  Serial.begin(9600);   //Begin serial communication
  pinMode(gasPin, INPUT); //Initialize the gas detector pin as an input
}

void loop(void) {
  gasReading = analogRead(gasPin); //Read the analog value of the gas sensor
  Serial.print("Gas detector value: "); //Print out
  Serial.println(gasReading);
  delay(10); //Short delay
}
```

Smoke and Air Pollution Sensors

The smoke sensors usually emit LED light, and a detector is typically shaded from the light. If there are particles of smoke present inside the sensor, the light is reflected by them, which can be observed by the detector.
Smoke detectors are used in fire alarm systems.
The air pollution sensors usually use a laser directed onto the detector. Between the laser and detector, the thin stream of air flows and pollution particles create shades on the detector. Thus, the detector can distinguish the sizes of particles and count the number of them.
Air pollution sensors are used in air purifiers and air quality measurement stations to monitor current air conditions, mainly in cities. Because the air pollution sensor generates more data, the serial connection is often used for reading measurement results. An example of an air pollution sensor that can count particles of PM1.0, PM2.5, and PM10 is PMS5003. PMS series sensors are controlled with a serial port and additional signalling GPIOs with 3.3V logic, but they require 5V to power on an internal fan that ensures correct airflow. A PMS5003 sensor is present in figures 224 and 225, and its connection in figure 226.

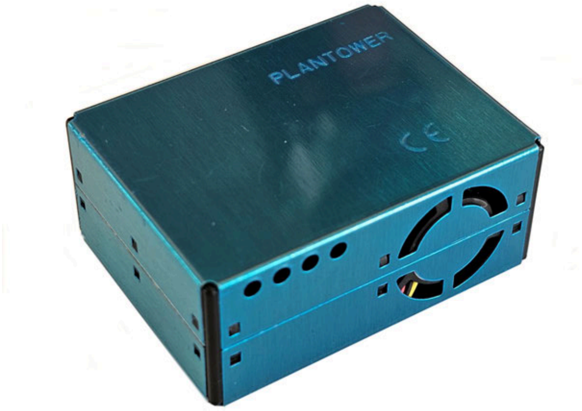


Figure 224: PMS5003 laser sensor for PM1.0, PM2.5 and PM10 - airduct fan side

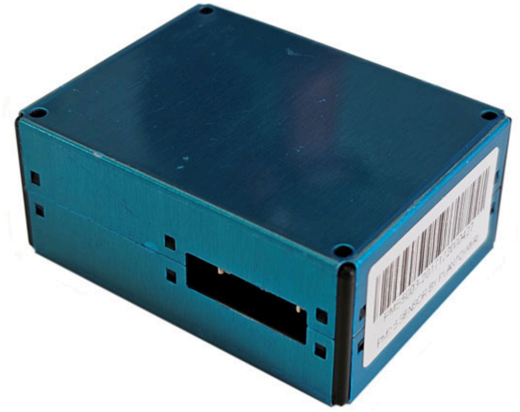


Figure 225: PMS5003 laser sensor for PM1.0, PM2.5 and PM10 - connector side

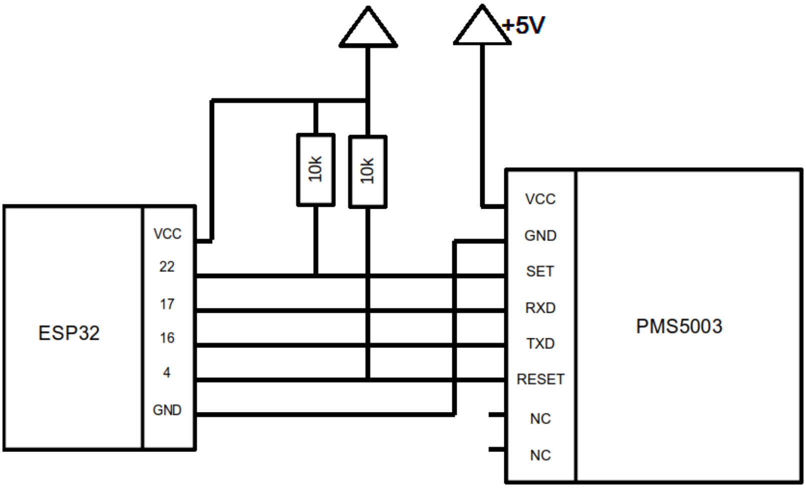


Figure 226: PMS5003 connection circuit for ESP32

An example code that uses the PMS5003 sensor:

```
#include <HardwareSerial.h>
#include <Arduino.h>

// Define the serial port for the PMS5003 sensor
HardwareSerial pmsSerial(1);
#define SET_PIN 22;
#define RESET_PIN 4;
#define RXD_PIN 16; //to TXD of the sensor
#define TDX_PIN 17; //to RXD of the sensor

bool verifyChecksum(uint8_t *data, int len);

void setup() {
  Serial.begin(9600);

  pinMode(SET_PIN, OUTPUT);      //controls sensor's low power mode
                                 //(LOW) -> turns fan down
  pinMode(RESET_PIN, OUTPUT);    //controls sensor's reset (LOW)
  digitalWrite(SET_PIN, HIGH);   //enable both
  digitalWrite(RESET_PIN, HIGH);

  pmsSerial.begin(9600, SERIAL_8N1, RXD_PIN, TXD_PIN);
}

void loop() {
  if (pmsSerial.available()) {
    if (pmsSerial.peek() == 0x42) {
      if (pmsSerial.available() >= 32) {
        uint8_t buffer[32];
        pmsSerial.readBytes(buffer, 32);


        if (verifyChecksum(buffer, 30)) {
          uint16_t pm25 = makeWord(buffer[10], buffer[11]);
          uint16_t pm10 = makeWord(buffer[12], buffer[13]);

          Serial.print("PM2.5: ");
          Serial.print(pm25);
          Serial.print(" ug/m3\t");
          Serial.print("PM10: ");
          Serial.print(pm10);
          Serial.println(" ug/m3");
        }
      }
    }
  }
}

// Function to verify the checksum
bool verifyChecksum(uint8_t *data, int len) {
  uint16_t checksum = 0;
  for (int i = 0; i < len - 2; i++) {
    checksum += data[i];
  }
  return (checksum == makeWord(data[len - 2], data[len - 1]));
}
```

Air Pressure Sensor

Air pressure sensors can measure the absolute pressure in the surrounding environment. Some popular sensors use a piezo-resistive sensing element, which is then connected to the amplifier and analogue digital converter. Frint-end uses the logic to interface the microcontroller. Usually, barometric sensor readings depend on the temperature, so they include the temperature sensor for temperature compensation of the pressure. Popular examples of barometric sensors are BME280 and BMP280. Both include barometric sensors and temperature sensors built in for compensation and possible measurement, while BME280 also consists of a humidity sensor. Communication with these sensors is done with an I2C or SPI bus.

Barometric sensors are commonly used in home automation appliances for heating, venting, air conditioning (HVAC), airflow measurement and weather stations. Because air pressure varies with altitude, they are often used in altimeters. Sample connection schematic is present in figure 228 and the module itself in figure 227.

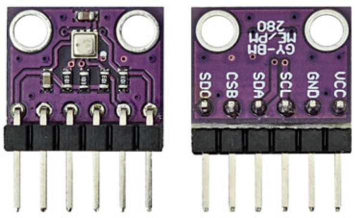


Figure 227: BME 280 air pressure sensor board

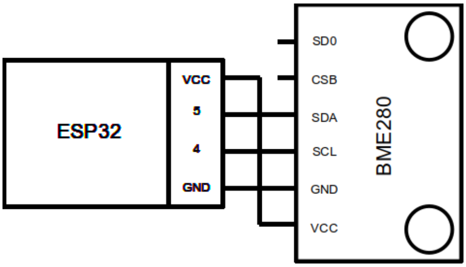


Figure 228: BME 280 connection circuit (I2C)

Opposite to the BMP280 (pressure only sensor), BME280 module boards usually do not contain voltage regulators and need to be powered with 3.3V (and so must be the signal logic).

An example code of BME280 use is below:

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

#define BME_ADDRESS 0x76

Adafruit_BME280 bme; // I2C

void setup() {
  Serial.begin(9600);
  bool status;
  Wire.begin(5,4); //SDA=GPIO5, SCL=GPIO4)
  status = bme.begin(BME_ADDRESS);
  if (!status) {
    Serial.println("Could not contact BME sensor");
    while (1);
  }
  delay(1000);
}


void loop() {

  Serial.print("Temperature=");
  Serial.print(bme.readTemperature());
  Serial.println("*C");

  Serial.print("Air pressure=");
  Serial.print(bme.readPressure() / 100.0F);
  Serial.println("hPa");

  Serial.print("Humidity=");
  Serial.print(bme.readHumidity());
  Serial.println("%rh");
  Serial.println();

  delay(1000);
}
```

5.2.8. Other Sensors

Hall sensor

A **Hall effect sensor** detects strong magnetic fields, their polarities and the relative strength of the field. In the Hall effect sensors, a magnetic force influences current flow through the semiconductor material and creates a measurable voltage on the sides of the semiconductor. Sensors with analogue output can measure the strength of the magnetic field, while digital sensors give *HIGH* or *LOW* output value, depending on the presence of the magnetic field.

Hall effect sensors are used in magnetic encoders for speed and rotation measurements. They can replace mechanical switches in keyboards and proximity switches because they do not require contact, which ensures high reliability. An example application can be sensing the position of rotary valves. Sample sensor is present in figure 229 and its connection to the Arduino board in figure 230.

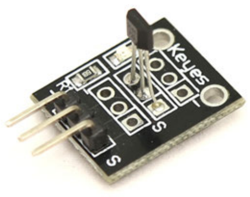


Figure 229: Hall-effect sensor module

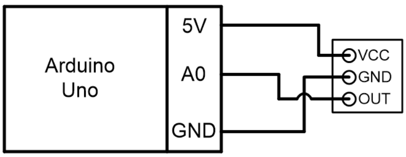


Figure 230: Arduino Uno and Hall sensor schematics

The example code:

```
int hallPin = A0; //Hall sensor output is connected to the analogue A0 pin
int hallReading;  //Stores Hall sensor reading

void setup(void) {
  Serial.begin(9600);      //Begin serial communication
  pinMode(hallPin, INPUT); //Initialize the Hall sensor pin as an input
}
```

```
void loop(void) {
  hallReading = analogRead(hallPin);   //Read the analogue value of the Hall sensor
  Serial.print("Hall sensor value: "); //Print out
  Serial.println(hallReading);
  delay(10); //Short delay
}
```

Global Positioning System

A GPS receiver is a device that can receive information from a global navigation satellite system and calculate its position on the Earth. A GPS receiver uses a constellation of satellites and ground stations to compute position and time almost anywhere on Earth. GPS receivers (figure 231) are used for navigation only in the outdoor area because they need to receive signals from the satellites, which is complicated inside the buildings. The GPS location's precision can vary depending on the number of visible satellites, weather conditions, and current satellites' placement. The GPS receiver is often connected to a microcontroller with a serial communication port and sends information according to the NMEA scheme (figure 232).

A GPS receiver is used for device location tracking. Real applications might be, e.g., pet, kid or personal belonging location tracking.

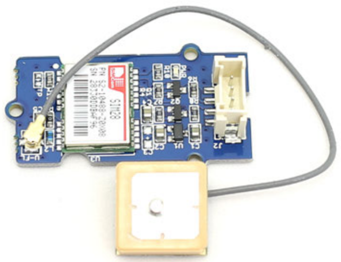


Figure 231: Grove GPS receiver module

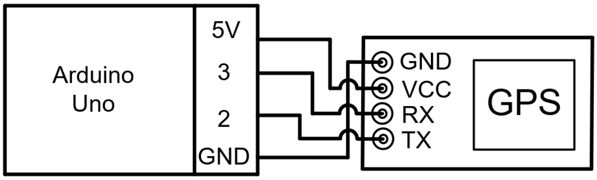


Figure 232: Arduino Uno and Grove GPS receiver schematics

The example code [148]:

```
#include <SoftwareSerial.h>
SoftwareSerial SoftSerial(2, 3);
unsigned char buffer[64];    //Buffer array for data receive over serial port
int count=0;                 //Counter for buffer array
void setup()
```

```
{
    SoftSerial.begin(9600);  //The SoftSerial baud rate
    Serial.begin(9600);      //The Serial port of Arduino baud rate.
}

void loop()
{
    if (SoftSerial.available())  //If data is coming from software serial port
                                 // ==> Data is coming from SoftSerial shield
    {
        while(SoftSerial.available())  //Reading data into char array
        {
            buffer[count++]=SoftSerial.read(); //Writing data into array
            if(count == 64)break;
        }
        Serial.write(buffer,count);    //If no data transmission ends,
                                       //Write buffer to hardware serial port
        clearBufferArray();            //Call clearBufferArray function to clear
                                       //The stored data from the array
        count = 0;                     //Set the counter of the while loop to zero
    }
    if (Serial.available())       //If data is available on hardware serial port
                                       // ==> Data is coming from a PC or notebook
    SoftSerial.write(Serial.read());   //Write it to the SoftSerial shield
}


void clearBufferArray()                //Function to clear buffer array
{
    for (int i=0; i<count;i++)
    {
        buffer[i]=NULL;
    }                           //Clear all content of an array with NULL
}
```

5.3. Actuators and Output Devices

An output device is a unit that changes an electrical signal coming from the microcontroller into the physical parameter. It can generate or modify light, sound, force, pressure and other physical values that influence other devices nearby or the surrounding environment. Some output elements can be connected directly to the microcontroller's pins, and some require higher voltage or current, so they need an additional electronic circuit called the driver. Output devices can be divided into groups based on the physical phenomenon they control. Popular output devices include LEDs, displays, motors (actuators), speakers, and buzzers.

5.3.1. Optical Output Devices

Light-Emitting Diode

Unlike the other diodes, the light-emitting diode, also called LED, is a particular type that emits light. LED has an entirely different body, which is made of transparent plastic that protects the diode and lets it emit light (figure 233). Like the other diodes, LED conducts the current in one way, so connecting it to the scheme is essential. There are two safe ways to determine the direction of the diode:

- the cathode's side of the diode housing is chipped,
- the anode's leg is usually longer than the cathode's leg.

Figure 233: 5 mm Red LED

The LED is one of the most efficient light sources. Unlike incandescent bulbs, LED transforms most of the power into light, not warmth; it is more durable, works for a more extended period and can be manufactured in a smaller size.
The semiconductor material determines the LED colour. Diodes are usually silicon, and LEDs are made from elements like gallium phosphate silicon carbide. Because the semiconductors used are different, the voltage needed for the LED to shine is also different.
When the LED is connected to the voltage and turned on, a considerable current starts to flow through it, and it can damage the diode. That is why all **LEDs have to be connected in series with a current-limiting resistor** (figure 234).

Current limiting resistors resistance is determined by three parameters:

- *I_D* – Current that can flow through the LED,

- *U_D* – Voltage that is needed to turn on the LED,
- *U* – Combined voltage for LED and resistor.

A short guide on calculating resistance for a diode is present below:

1. Find out the voltage needed for the diode to work *U_D*; you can find it in the diode parameters table.
2. Find out the amperage needed for the LED to shine *I_D*; it can be found in the LEDs datasheet, but if you can't find it, then 20 mA current is usually a correct and safe choice.
3. Find out the combined voltage for the LED and resistor; usually, it is the feeding voltage for the scheme.
4. Insert all the values into this equation: *R = (U – U_D) / I_D*.
5. You get the resistance for the resistor for the safe use of the LED.
6. Find a resistor with a nominal value that is the same or slightly bigger than the calculated resistance.

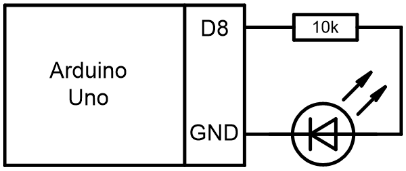


Figure 234: Arduino Uno and LED control schematic

An example of the blinking LED code:

```
int ledPin = 8;//Defining the pin of the LED

void setup()
{
    pinMode(ledPin,OUTPUT); //The LED pin is set to output
}

void loop()
{
    //Set pin output signal to HIGH – LED is working
    digitalWrite(ledPin,HIGH);
    //Belay of 1000 ms
    delay(1000);

    //Set pin output signal to LOW – LED is not working
    digitalWrite(ledPin,LOW);
    //Delay of 1000 ms
    delay(1000);
}
```

LED's brightness can be controlled easily with a PWM signal.
There exist LEDs with more than one light-emitting chip in one enclosure. They are made as two-coloured or RGB elements with coloured controlled separately. There are two internal configurations of such elements:

- common anode - anodes of all internal LEDs are connected (for sample MCU connection, look in figure 235),

- common cathode - cathodes of all internal LEDs are connected (for sample MCU connection, look in figure 236).

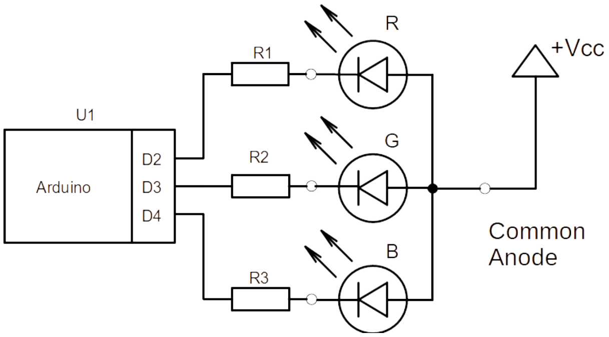


Figure 235: Connection of RGB common anode LED to Arduino

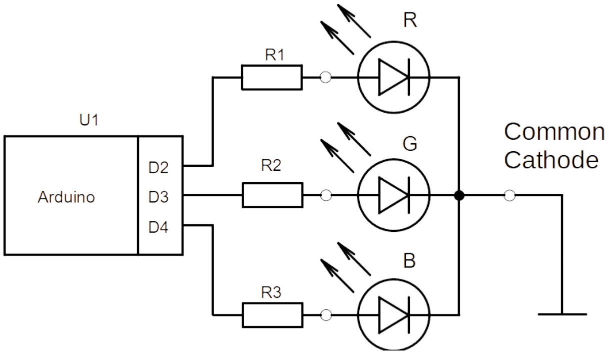


Figure 236: Connection of RGB common cathode LED to Arduino

Digital LED

Digital LED does not have anode or cathode connections available externally. They have power supply pins and two pins for data transmission, one for input and a second for output. The input accepts the digital signal from the microcontroller to set the brightness of all three internal LEDs. Output connects the input of another LED to form a series of LEDs. Digital LEDS are available as single elements but also as strips, rings or matrices that a microcontroller with one pin can control. Every LED can shine in different colours, creating interesting visual effects. An example of a popular digital LED is WS2812. A particular protocol is used to transmit data. One LED requires 24 bits (1 byte for red, 1 for green, and 1 for blue) to set the colour. After receiving its data, the LED resends any further byte to the following LEDs in the chain.
There are software libraries for Arduino and other platforms available to ease the handling of digital LEDs, including advanced visual effects for stripes, matrices and other shapes. Sample 8 LED WS2812 stripe is present in the figure 237 and its connection to the MCU in 238.

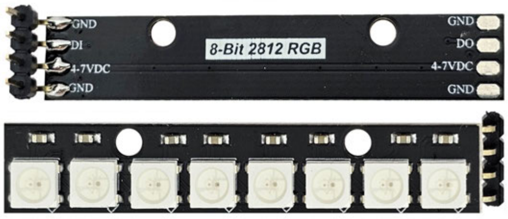


Figure 237: WS2812 8 smart LEDs stripe

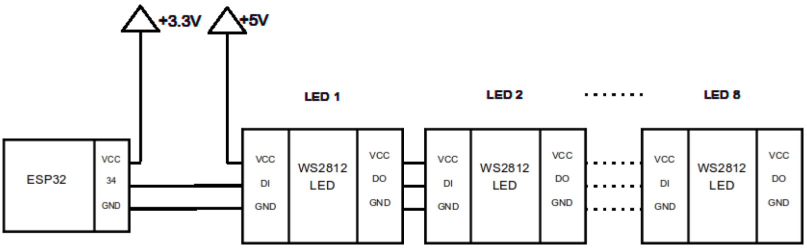


Figure 238: MCU control of the digital LEDs

The example code that uses the popular Adafruir NeoPixel library:

```
#include <Adafruit_NeoPixel.h>

#define PIN       34 //Define the pin connected to the digital LED data input
#define NUMPIXELS  8 //Define the number of LEDs in the strip

Adafruit_NeoPixel pixels = Adafruit_NeoPixel(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

void setColor(uint8_t red, uint8_t green, uint8_t blue) {
  for (int i = 0; i < pixels.numPixels(); i++) {
    pixels.setPixelColor(i, pixels.Color(red, green, blue));
  }
  pixels.show();
}

void setup() {
  pixels.begin();      // Initialize the NeoPixel library
}

void loop() {
  // Change the colour of the NeoPixel LED
  setColor(255, 0, 0); // Red color (R, G, B)
  delay(1000);         // Delay to make the colour change visible (in milliseconds)
  setColor(0, 255, 0); // Green color (R, G, B)
  delay(1000);         // Delay to make the colour change visible (in milliseconds)
  setColor(0, 0, 255); // Blue color (R, G, B)
  delay(1000);         // Delay to make the colour change visible (in milliseconds)
}
```

Displays

A display is a quick way to get feedback information from the device. There are many display technologies. For IoT solutions, low-power, easy-to-use displays are used:

- 7-segment LED display,
- LED matrix display,
- liquid-crystal display (LCD),
- organic light-emitting diode display (OLED),
- electronic ink display (E-ink).

7-segment LED display

The seven-segment LED display is built with seven LEDs forming the shape, making it possible to display symbols similar to digits and even some letters. Usually, the eighth LED is added as the decimal point. 7-segment displays can have similar internal connections as RGB LEDs, common anode or common cathode. If there is more than one digit in the element, all the same segments are also connected. Such displays need special controllers or the software part that displays separate digits in a sequence one by one. To avoid unnecessary blinking or differences in the brightness of digits, software for sequential displays is written using timers and interrupts. As for the RGB LEDs, 7-segment displays need a separate resistor for every segment. Sample 2-digit 7-segment module is present in the figure 239.

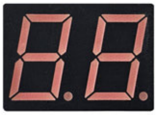


Figure 239: 7 segment LED display

LED matrix display

LED matrix displays offer the possibility of displaying not only digits and letters but also pictograms and symbols. The most popular versions have 8 rows and 8 columns (figure 240), or 7 rows and 5 columns, but it is possible to find other configurations. As for the 7-segment displays, there are common anode and common cathode configurations. All anodes in one row and all cathodes in one column are connected to a common anode. For a common cathode, all cathodes in one row and all anodes in one column are connected. Modern LED matrix displays have built-in controllers or are made with digital RGB LEDs, making it possible to display pictures and videos.

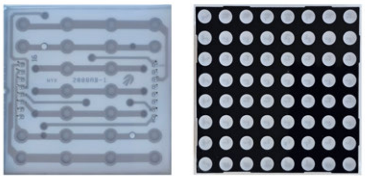


Figure 240: 8×8 LED matrix

Liquid-Crystal Display (LCD)

Monochrome LCD uses modulating properties of liquid crystal to block the passing-through light. Thus, when a voltage is applied to a pixel, it is dark. A display consists of layers of electrodes, polarising filters, liquid crystals and a reflector or backlight. Liquid crystals do not emit light directly but through reflection or backlight. Because of this reason, they are more energy efficient. Small, monochrome LCDs are widely used to show little numerical or textual information like temperature, time, device status, etc. The most popular LCD device is an alphanumerical 2×16 characters display based on the HD44780 controller (figure 241).
There also exist graphic monochrome and colour TFT displays that use LCD technology. LCD modules commonly come with an onboard control circuit and are controlled through parallel or serial interfaces. Sample circuit for 2×16 display is present in figure 242.

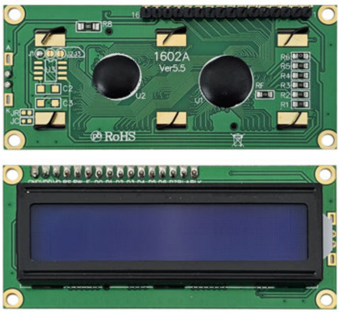


Figure 241: Blue 16 × 2 LCD display

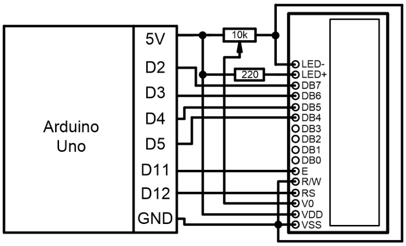


Figure 242: Arduino and LCD 2×16 connection schematics

The example code:

```
#include <LiquidCrystal.h> //include LCD library

//Define LCD pins
const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
//Create an LCD object with predefined pins
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

void setup() {
  lcd.begin(16, 2); //Set up the LCD's number of columns and rows
  lcd.print("hello, world!"); //Print a message to the LCD
}

void loop() {
  //Set the cursor to column 0, line 1 – line 1 is the second row
  //Since counting begins with 0
  lcd.setCursor(0, 1);
  //Print the number of seconds since the reset
  lcd.print(millis() / 1000);
}
```

Organic Light-Emitting Diode Display (OLED)
OLED display uses electroluminescent materials that emit light when the current passes through these materials. The display consists of two electrodes and a layer of an organic compound. OLED displays are thinner than LCDs, have higher contrast, and can be more energy efficient depending on usage (figure 243). OLED displays are commonly used in mobile devices like smartwatches and cell phones, replacing LCDs in other devices. OLED displays come as monochrome or RGB colour devices. Small OLED display modules usually have an onboard control circuit that uses digital interfaces like I2C (figure 244) or SPI.

Figure 243: OLED I2C display

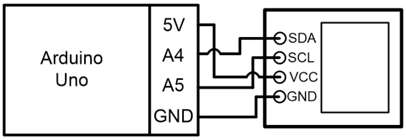


Figure 244: Arduino and OLED I2C schematics

```
//Add libraries to ensure the functioning of OLED
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#define OLED_RESET 4
Adafruit_SSD1306 display(OLED_RESET);

void setup() {
  //Setting up initial OLED parameters
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C, false);
  display.setTextSize(1); //Size of the text
  display.setTextColor(WHITE); //Colour of the text – white

void loop() {

  //Print out on display output sensor values
  display.setCursor(0, 0);
  display.clearDisplay();
  display.print("Test of the OLED"); //Print out the text on the OLED
  display.display();
  delay(100);
  display.clearDisplay();
}
```

Monochrome Electronic Ink Displays (E-Ink)
E-ink display uses charged particles to create a paper-like effect. The display comprises transparent microcapsules filled with oppositely charged white and black particles between electrodes. Charged particles change their location depending on the orientation of the electric field; thus, individual pixels can be either black or white (figure ##REF:eink0##). The image does not need power to persist on the screen; power is used only when the image is changed. Thus, the e-ink display is very energy efficient. It has a high contrast and viewing angle but a low refresh rate. E-ink displays are commonly used in e-readers, smartwatches, outdoor signs, and electronic shelf labels. Sample E-Ink module is present in figure 245. The majority of the e-Ink displays are controlled with an SPI interface. Sample connection is present in figure 247.

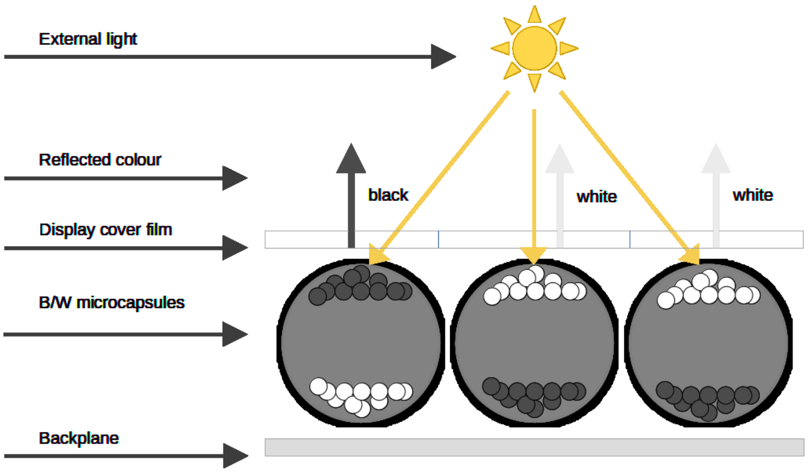


Figure 245: Monochrome E-Ink display working principle

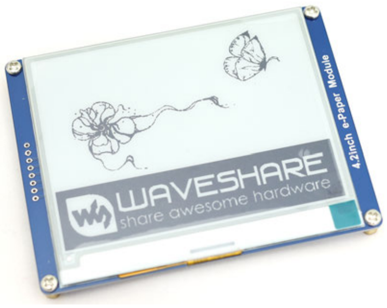


Figure 246: E-ink display module

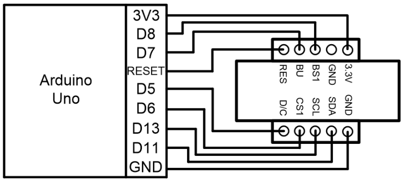


Figure 247: Arduino Uno and E-ink display module schematics

```
#include <SmartEink.h>
#include <SPI.h>

E_ink Eink;

void setup()
{
  //BS LOW for 4 line SPI
  pinMode(8,OUTPUT);
  digitalWrite(8, LOW);

  Eink.InitEink();
  Eink.ClearScreen();//Clear the screen
  Eink.EinkP8x16Str(14,8,"IoT e-ink example");
  Eink.EinkP8x16Str(10,8,"IoT-open.eu");
  Eink.EinkP8x16Str(6,8,"0123456789");
  Eink.EinkP8x16Str(2,8,"9876543210");
  Eink.RefreshScreen();
}
void loop()
{

}
```

Colourful e-Ink displays
Recent advances in E-Ink (E-Paper) technology present the ability to display coloured information. Various approaches are present in the engineering of colourful E-Ink displays, along with multiple technologies for the presentation of colours.

Tri-colour e-Ink displays with predefined colour areas are a development of the black-white ones where part of the capsules (usually the upper half), instead of containing black microcapsules, contain yellow or red. This enables the presence of the information in black or selected colour, but the colour depends on the location of the information on the display. This display was designed for shopping shelves (ESL-Electronic Shelf Label) to emphasize benefits or bargains.

Grayscale e-Ink displays benefit from the fact that microcapsules inside a pixel sphere do not travel simultaneously. As some capsules have more charge than others, it is possible to design and charge them the way that variable external charge can pull or push not all of them but just partially. In practice, it enables the presentation of grayscale in a single pixel as observed from a distance. A principle of operation is present in figure 248.

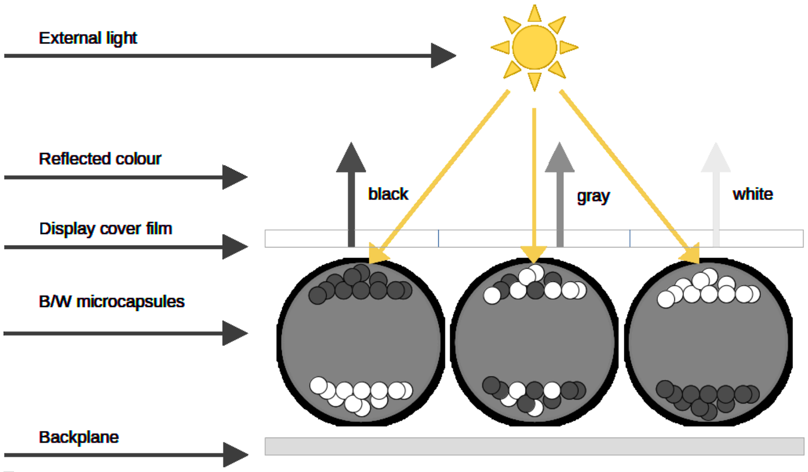


Figure 248: Grayscale E-Ink display operation principle

Opposite to the above, multicolour e-Ink displays provide a true selection of colours per pixel and are implemented in various technologies presented below.

Multicolour with filtering
In this construction, classical black-white capsules are covered with colour RGB filters on top of them. A single pixel is then composed, in fact, of 3 spheres, covered with red, green and blue and the final colour is observed as a mixture of those. Moreover, controlling a single sphere similarly to the grayscale displays enables an even bigger number of colours presented by a single pixel domain without using high resolution and dithering. This kind of display uses additive colour mixing (RGB). A principle of operation is present in figure 249.

Note, in RGB filtered displays, at least 3 spheres are needed to present a single colourful pixel, so the overall resolution is lower than in monochrome or grayscale E-Inks.

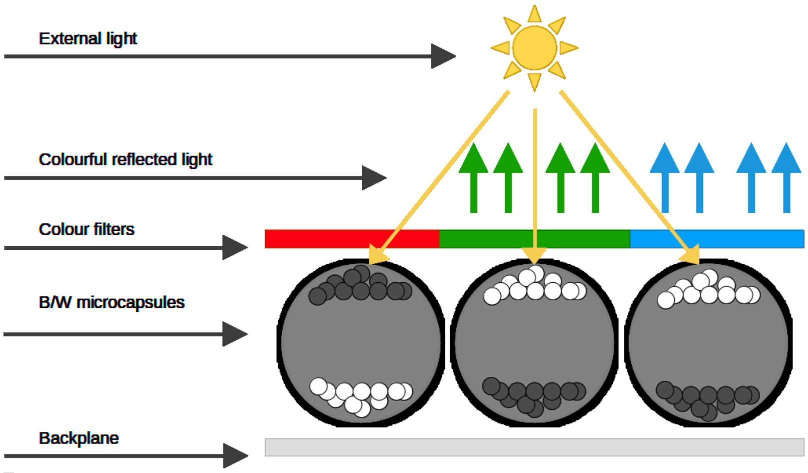


Figure 249: A construction of the E-Ink colourful display with RGB filtering

Multicoloured capsules in a single sphere (ACEP Advanced Colour ePaper)
In this approach, capsules in a single sphere are multicoloured rather than black-white. Microcapsules of different colours have slightly different charging, so a variating external electric field applied to the single sphere controls the colour of the capsules on the top of the sphere that is visible to the user. A single sphere can then present a wide range of colours. This kind of display uses subtractive colour mixing (CMY/CMYK). A principle of operation is present in figure ##REF:eink5##.

This solution provides quite good resolution, but controlling the microcapsules is tricky and requires complex electric field control.

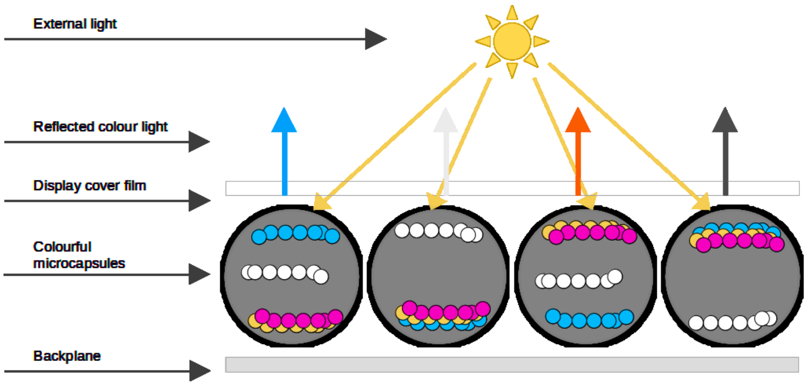


Figure 250: ACEP E-Ink display operation principle

Multicoloured capsules in separate spheres

This approach is theoretical as manufacturing such devices is inefficient because of the need to compose a matrix of spheres with different colours of microcapsules nearby. A domain of such spheres composes a single pixel. A principle of operation is present in figure 251.

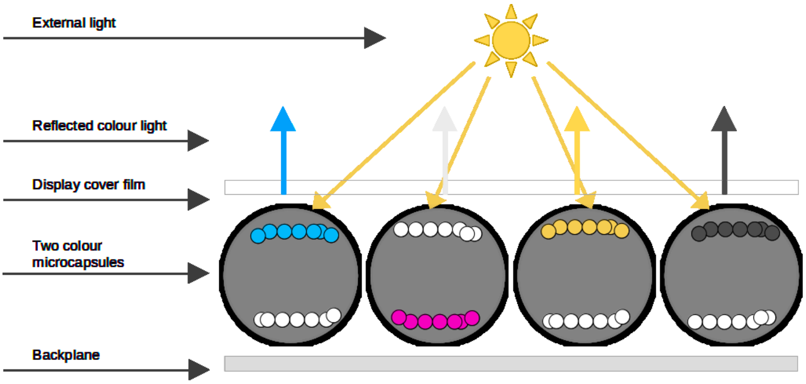


Figure 251: Multicoloured capsules E-Ink display operation principle

5.3.2. Electromechanical Devices

Relay

Relays are electromechanical devices that use electromagnets to connect or disconnect the plates of a switch. Relays are used to control high-power circuits with low-power circuits. Both circuits are electrically isolated; thus, the control logic is protected from high voltage, sometimes from the power mains. Relays are used in household appliance automation, lighting and climate control. Although the electromagnet's coil of the relay requires relatively low power compared to the power capability of the output circuit, it cannot be connected directly to the microcontroller's pin. Creating the transistor driver or using a relay module with the driver built-in is possible. The sample relay module is present in figure 252 and its connection to the Arduino development board in figure 253.

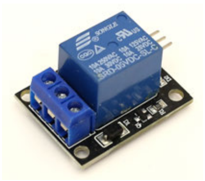


Figure 252: One-channel relay module

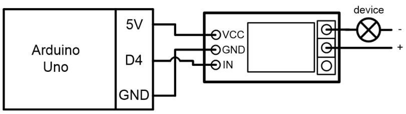


Figure 253: Arduino Uno and one-channel relay module schematics

The following example code should work properly for the relay module. It turns the relay on while there is a “1” state at the microcontroller's output and turns the relay off while there is a “0” state at the output.

```
#define relayPin  4 //Define the relay pin

void setup()
{
  Serial.begin(9600);
  pinMode(relayPin, OUTPUT); //Set relayPin to output

}

void loop()
{
   digitalWrite(relayPin,1);  //Turns relay on
   Serial.println("Relay ON"); //Output text
   delay(2000); // Wait 2 seconds

   digitalWrite(relayPin,0);  //Turns relay off
   Serial.println("Relay OFF");
   delay(2000);
}
```

Solenoid

Solenoids use electromagnets to pull or push iron or steel cores. They are used as linear actuators for locking mechanisms indoors, pneumatic and hydraulic valves and in-car starter systems.
Solenoids and relays use electromagnets, and connecting them to Arduino is very similar. Coils need much power and are usually attached to the circuit's power source using a transistor driver. Turning the coil's power off makes the electromagnetic field collapse and creates a very high voltage. A shunt diode channels the overvoltage for the semiconductor devices' protection. For extra safety, an optoisolator can be used. Sample solenoid is present in figure 254 and connection to the MCU in figure 255.

Figure 254: A solenoid

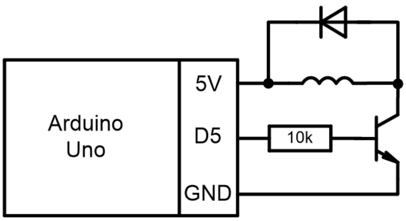


Figure 255: Arduino Uno and solenoid schematics

The example code to control solenoid is present below:

```
#define solenoidPin  4 //Define the solenoid pin

void setup()
{
  Serial.begin(9600);
  pinMode(solenoidPin, OUTPUT); //Set solenoidPin to output

}

void loop()
{
   digitalWrite(solenoidPin,1);  //Turns solenoid on
   Serial.println("Solenoid  ON"); //Output text
   delay(2000); //Wait 2 seconds

   digitalWrite(solenoidPin,0);  //Turns solenoid off
   Serial.println("Solenoid OFF");
   delay(2000);
}
```

5.3.3. Sound Output Devices

Speaker

Speakers are electroacoustic devices that convert electrical signals into sound waves. A speaker uses a permanent magnet and a coil attached to the membrane. Sound signal flowing through the coil creates an electromagnetic field with variable strength; the coil attracts a magnet according to the strength of the field, thus making a membrane vibrate and creating a sound wave.
Another widely used speaker technology, called piezo speaker, uses piezoelectric materials instead of magnets.
Speakers are used to create an audible sound for human perception and ultrasonic sound for sensors and measurement equipment. Some speakers are designed to generate a single, fixed-frequency acoustic tone. Such elements are called buzzers and have a built-in generator to emit sound if the voltage is on. Elements without built-in generators should be controlled with the frequency signal coming from the microcontroller. Sound-generating devices require more power than LED, so there is a need to check if the operating current is lower than the maximum current of the microcontroller's pin as specified in technical documentation. In the case of the overcurrent, an additional amplifying element is required, e.g., a transistor. Sample speaker is present in the figure 256 while connection of the piezoelectric one is present in the figure 257.

Figure 256: Electromagnetic speaker, 8 Ω 0.5 W

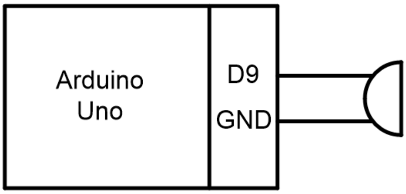


Figure 257: Arduino Uno and piezo speaker schematics

```
const int speakerPin = 9;      //Define the piezo speaker pin

void setup()
{
  pinMode(speakerPin, OUTPUT); //Set spekaer pin as an output
}

void loop()
{
  tone(speakerrPin, 1000);     //Send 1 kHz sound signal
  delay(1000);                 //For 1 s
  noTone(speakerPin);          //Stop sound
  delay(1000);                 //For 1 s
}
```

5.3.4. Actuators

Actuators are devices that can do a physical action to the surrounding world. Most actuators are based on one of the forms of electric motors, sometimes directly, sometimes using a gearbox and advanced control logic.
An electric motor is an electromechanical device which can turn electrical energy into mechanical energy. The motor turns because the electricity in its winding generates a magnetic field that inducts the mechanical force between the winding and the magnet. Electric motors are made in many variants, of which the simplest is the permanent-magnet DC motor.

DC Motor (One Direction)

DC motor is a device which converts direct current into mechanical rotation. DC motor consists of permanent magnets in the stator and coils in the rotor. Applying the current to coils creates an electromagnetic field, and the rotor tries to align itself to the magnetic field. Each coil is connected to a commutator, which supplies coils with current, thus ensuring continuous rotation. Some motors have a tachometer functionality as the loopback signal that generates a pulse train of frequency proportional to the rotation speed. Tacho signal can be connected to a digital or interrupt input of a microcontroller, allowing for determining actual rotation speed. DC motors are widely used in power tools, toys, electric cars, robots, etc. (figure 258). The connection schematic for a small DC motor is present in figure 259.

Figure 258: A DC motor with gearbox 50:1

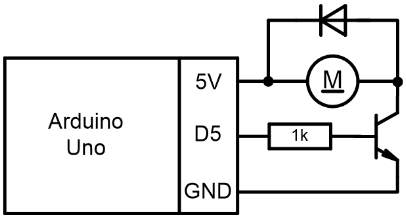


Figure 259: Arduino Uno and DC motor schematics

Sample code to control a DC motor using Arduino framework is present below:

```
void setup ()
{
  pinMode(5,OUTPUT); //Digital pin 5 is set to output
  //The function for turning on the motor is defined
  #define motON digitalWrite(5,HIGH)
  //The function for turning off the motor is defined
  #define motOFF digitalWrite(5,LOW)
}
void loop ()
{
  motON; //Turn on the motor
}
```

DC Motor With H-Bridge

The H-bridge has earned its name because it resembles the capital 'H' wherein all the corners are switches, and the electric motor is in the middle. This bridge is usually used for operating permanent-magnet DC motors, electromagnets and other similar elements because it allows working with significantly bigger current devices using a small, driving current. By switching the switches, it is possible to change the motor direction. It is important to remember that the switches must be turned on and off in pairs (figure 260).

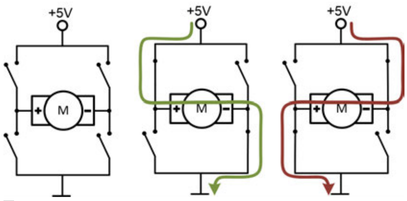


Figure 260: The flow of currents in the H-bridge

When all switches are turned off, the motor is in free movement. It is not always acceptable, so two solutions can be implemented. If both positive or negative switches are turned on at the top or the bottom, then the motor coil is shorted, not allowing it to have a free rotation – it is slowed down faster. The fastest option to stop the motor is to turn the H-bridge in the opposite direction for a while.

Neither of these braking mechanisms is good for the H-bridge or the power source because of excessive current appearance. This action is unacceptable without a particular reason because it can damage the switches or the power source.

The motor management can be reflected in the table 34.

Table 34: The Management of the H-Bridge Switches

Upper left	Upper right	Lower left	Lower right	Motor work mode
On	Off	Off	**On**	**Turns in one direction**
Off	**On**	**On**	Off	**Turns in another direction**
On	**On**	Off	Off	Braking
Off	Off	**On**	**On**	Braking

The complicated part is implementing and controlling the switches mentioned above, usually as relays or appropriate power transistors. The biggest drawback of relays is that they can only turn the engine on or off. Transistors must be used if the rotation speed needs to be regulated using the pulse width modulation. The MOSFET-type transistors should be used to ensure a large amount of power.
Nowadays, the stable operation of the bridge is ensured by adding extra elements. All elements can be encapsulated in a single integrated circuit, e.g. L293D (figure 261).

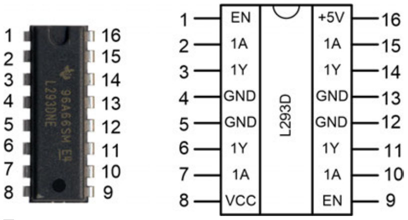


Figure 261: The L293D microchip and its representation in the circuit

The L293D microchip consists of two H-bridges and is made for managing two motors. It

has separate control pins for the left and right branches, avoiding the power short circuit if connected properly.
An example schematic diagram of connecting the chip to the Arduino Uno board can be seen in figure 262.

Using a PWM signal allows control of the rotation speed.

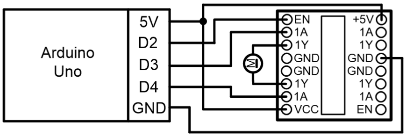


Figure 262: Arduino Uno and L293D H-bridge schematics

The example code to control the L293D chip is presented below:

```
int dirPin1 = 7;    //1st direction pin
int dirPin2 = 8;    //2nd direction pin
int speedPin = 5;   //Pin responsible for the motor speed

void setup ()
{
  pinMode (dirPin1,OUTPUT);  //1st direction pin is set to output
  pinMode (dirPin2,OUTPUT);  //2nd direction pin is set to output
  pinMode (speedPin,OUTPUT); //Speed pin is set to output
}

void loop ()
{
  analogWrite(speedPin, 100); //Setting motor speed
  //Speed value can be from 0 to 255

  int motDirection = 1; //Motor direction can be either 0 or 1

  if (motDirection) //Setting motor direction
  {//If 1
    digitalWrite(dirPin1,HIGH);
    digitalWrite(dirPin2,LOW);
  }
  else
  {//If 0
    digitalWrite(dirPin1,LOW);
    digitalWrite(dirPin2,HIGH);
  }
}
```

Linear actuator

A bidirectional DC motor, usually controlled with an H-bridge and equipped with thread gear, can be used to implement the linear actuators.
Linear actuators used to be equipped with end position detectors such as switches, or eventually, their end positions can be detected with overload detection. A simple linear actuator is present in figure 263.

Figure 263: Low voltage linear actuator

Stepper Motor

A certain angle or step can move stepper motors. The full rotation of the motor is divided into small, equal steps. Stepper motor has many individually controlled electromagnets; turning them on or off makes a motor shaft rotate by one step. Changing the switching speed or order can precisely control the rotation's angle, direction or speed. Because of their exact control ability, they are used in CNC machines, 3D printers, scanners, hard drives, etc.
A popular stepper motor is present in figure 264 and its controlling circuit in figure 265.
An example of use can be found in the source [149].

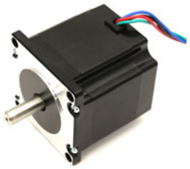


Figure 264: A stepper motor

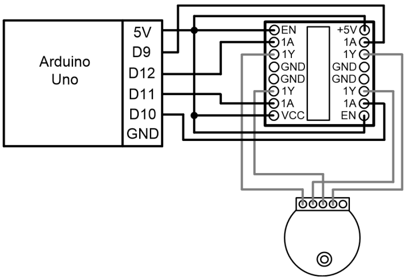


Figure 265: Arduino Uno and stepper motor schematics

The example code:

```
#include <Stepper.h> //Include library for stepper motor

int in1Pin = 12; //Defining stepper motor pins
int in2Pin = 11;
int in3Pin = 10;
int in4Pin = 9;

//Define a stepper motor object
Stepper motor(512, in1Pin, in2Pin, in3Pin, in4Pin);

void setup()
{
  pinMode(in1Pin, OUTPUT); //Set stepper motor control pins to output
  pinMode(in2Pin, OUTPUT);
  pinMode(in3Pin, OUTPUT);
  pinMode(in4Pin, OUTPUT);

  Serial.begin(9600);
  motor.setSpeed(20); //Set the speed of stepper motor object
}

void loop()
{
    motor.step(5); //Rotate 5 steps
}
```

Servomotor

The servomotor includes the internal closed-loop position feedback mechanism that precisely controls its position angle. To set the angle, the PWM technique is used. Additionally, it is possible to control the speed of angle change, acceleration and deceleration of the rotation.
Servomotors have limited rotation angles depending on their type, e.g. 90, 180 or 270 degrees. A typical servo is 180 degrees (usually a bit lower). Servo powering depends on size; micro servos are typically between 4.8V and 6V. Larger servos require higher voltage and more current to operate.
There are two standards for controlling servos, so-called “analogue” and “digital”. Analogue servos are controlled with a PWM signal of 50Hz (20ms period), while digital servos, even if backwards compatible with analogue, can be controlled with a PWM signal up to 250Hz. A duty cycle (a ratio between *HIGH* and *LOW*) controls servo position. We focus below on analogue servomotors, controlled with a 50Hz PWM signal, but the actuator is present in figure 263.

Digital servos used to have individual configurations of the control signals, and it is necessary to refer to the documentation of the particular model for correct timings.

From the figure 266, it can be seen that the length of the servomotor impulse cycle is 20 ms, but the impulse length itself is 1 ms or 2 ms. These signal characteristics are true for most enthusiast-level servomotors but should be verified for each module in the manufacturer specification, e.g. to obtain a full rotation of 180 degrees, it may be necessary to go beyond standard 1ms↔2ms duty cycle.
The servomotor management chain meets the impulse every 20 ms, but the pulse width shows the position the servomotor has to reach. For example, 1 ms corresponds to the 0° position but 2 ms – to the 180° position against the starting point. When entering the defined position, the servomotor will keep it and resist any outer forces trying to change the current position. The graphical representation of the control signal and its impact on the position of the servomotor is presented in image 266.

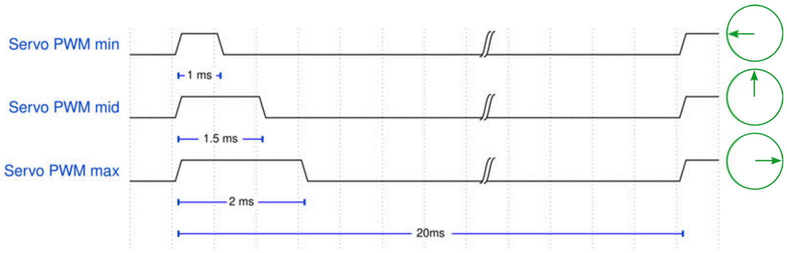


Figure 266: The pulse width modulated signal for different positions of servomotor

Just like other motors, servomotors have different parameters, where the most important one is the time of performance – the time necessary to change the position to the defined position. The best enthusiast-level servomotors do a 60° turn in 0.09 s. There are three types of servomotors:

- **Positional rotation servomotor** – most widely used type of servomotor. With the help of a management signal, it can determine the position of the rotation angle from its starting position.
- **Continuous rotation servomotor** – this type of motor allows setting the speed and direction of the rotation using the management signal. If the position is less than 90°, it turns in one direction, but if more than 90°, it turns in the opposite direction. The speed is determined by the difference in value from 90°; 0° or 180° will turn the motor at its maximum speed while 91° or 89° at its minimum rate.
- **Linear servomotor** – with the help of additional transfers, it allows moving forward or backwards; it doesn't rotate.

Unfortunately, using Arduino, the servomotor is not as easily manageable as the DC motor. For this purpose, a special servomotor management library, “Servo.h” has been created. Using PWM signal in other MCUs may involve the use of hardware or software timers and may impact other features as the number of hardware timers used to be limited. Thus, “Servo.h” implementation may vary between microcontrollers and SDKs.

Sample standard servo is present in figure 267 and connection in figure 268.

Figure 267: A standard servomotor

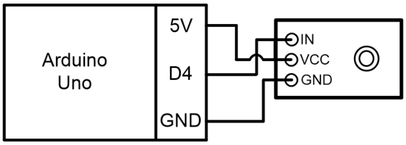


Figure 268: Arduino Uno and servomotor schematics

The example code to control a servo:

```
#include <Servo.h> //Include Servo library
Servo servo; //Define a Servo object

void setup ()
{
  servo.attach(6); //Connect servo object to pin D6
  servo.write(90); //Set position of servo to 90°
  Serial.begin(9600);
}

void loop ()
{
  servo.write(110); //Set position of servo to 110°
  delay(200); //wait for 200 ms
  servo.write(70);//Set position of servo to 70°
  delay(200); //Wait for 200 ms
}
```

5.4. Powering of the IoT Devices

IoT requires a constant and reliable power source to operate devices, sensors, and communication effectively. The choice of power source for IoT devices is from traditional batteries to cutting-edge energy harvesting technologies. The factors influencing the choice are:

- device size,
- working environment,
- intended operation,
- lifetime of the device,
- operation reliability and availability.

The majority of IoT devices use a DC power source. AC is usually converted into the DC, eventually used for powering high-power actuators. Most IoT devices rely on batteries as their energy source, which is common in edge devices. Fog devices are powered with a mixture of the sources: battery (DC) or socket/grid (AC). Green energy sources are introduced in both classes; the choice of technology depends on the IoT application domain and scenario.

Batteries (non-rechargeable or rechargeable) provide DC. A plain battery's voltage (e.g. 1.5V or 3.7V) is unsuitable for directly powering the devices. The raw battery changes its voltage over time as the discharge process occurs. For those reasons, converters and stabilisers are used. In the case of modern rechargeable batteries, a charging and discharging module (battery management) is also necessary to prevent overcharging and overdraining.
Using green energy sources requires conversion and energy storage as its nature is non-consistent in the time domain. Using green energy as a power source requires a different approach towards IoT device control algorithms because of the unpredictable nature of the green energy sources. It is common for devices to put themselves into the low power consumption mode on demand because of the sudden lack of energy.

The majority of IoT devices require one or two voltage standards to power microcontrollers and sensors:

- 5V - common of older microcontrollers (such as AVR) and for most peripherals,
- 3.3V - for recent microcontrollers, more and more peripherals are 3.3V powered.

Integration of the 3.3V and 5V components in one IoT device is not always straightforward: while controlling a 5V powered device with a 3.3V signal is usually non-problematic (GPIO-IN), the opposite requires signal conversion because most devices do not accept overvoltage on their inputs! Driving a 3.3V GPIO with 5V can damage a device!

A special note on powering IoT devices with inductive actuators: a separate DC powering rail should be used. Actuators using electromagnetic components, such as relays, motors and servos, should not share a powering bus with MCU. They introduce profound inference into the power rail (both rising and falling the nominal voltage), thus frequently causing instability or rebooting of the MCU due to the over / under voltages or even leading to permanent damage. Eventually, capacitor-based filtering can be introduced to reduce power rail inference if separation is impossible.

When dealing with high current, high voltage or high inference devices, it is expected to use physical separation of the signals with means of e.g. optocouplers.

During their operation, IoT devices commonly control their power sources regarding their condition (e.g., remaining energy, ageing symptoms). In the simplest case, the battery terminal can be connected to the A/D GPIO (usually via a voltage divider). It is also expected to measure battery drain, constantly monitoring both current and voltage, thus calculating energy consumed. Battery Management Systems can be a stand-alone module or can be a part of the IoT device. In the first case, using some communication protocol to monitor power source status is required (e.g. Serial or I2C). IoT devices can then inform the IoT ecosystem about, e.g., approaching running out of power. They can also limit their energy consumption by switching to sleep mode.

IoT Energy Sources

P B M E

A reliable energy source is required to keep an IoT device alive. An interruption is when the energy source shuts down the IoT device, increasing downtime and reducing the quality of service or the quality of experience the users feel. Therefore, choosing the energy source is very important when designing IoT systems. The following factors should be considered when selecting an energy source for an IoT device:

- Size: In some IoT applications, it is required that the size of the IoT device should be as small as possible. The chosen power source should be one whose size can be scaled down as much as possible.
- Mobility: In some IoT applications, mobility is an essential requirement (e.g., in wearable IoT devices), imposing both size and weight constraints on the IoT devices. The energy source chosen should not be location-dependent sho, should be able to provide energy to the IoT devices during motion, and should not obstruct the device's mobility.
- Reliability: The energy source chosen should be reliable. That one will supply energy to the IoT devices when necessary with minimal failures.
- Scalability: The energy source chosen should be easily accessible at affordable prices to ensure a continuous supply of energy to the growing number of IoT devices and infrastructure.
- Cost: The energy source should be cheap, and the cost of energy should also be reasonable.
- minimum maintenance requirement: It is recommended to choose energy sources to ensure that the devices' lifetime is reasonably long. That is a minimal energy-related maintenance requirement, especially in large IoT networks.
- Ease of deployment: The energy source should be simple to integrate into the IoT system.
- IoT application context: The choice of the energy source depends mainly on the application context.
- Power requirement: The choice of the energy source also depends on the energy requirement of the device. The energy sources required for IoT devices in agriculture differ from those in smart factories or smart health applications.
- Sustainability: The energy sources chosen should be green and sustainable. The energy source should produce minimal environmental pollution and be easily recyclable.

The choice of the energy source is critical in the IoT design process as it will influence the selection of the computing power, communication protocols and technologies, and the security mechanism and other subsystems of the IoT system. The three primary energy sources for IoT devices are:

- Main energy
- Energy storage systems
- Energy harvesting systems.

Main power

In IoT applications where the hardware devices do not need to be mobile and are energy-hungry (consume significant energy), they can be reliably powered using mains power sources. The grid's main power is AC power and should be converted to DC power and scaled down to meet the power requirement of sensing, actuating, computing, and networking nodes. The hardware devices are the networking or transport layer, and those at the application layer (fog/cloud computing nodes) are often power-hungry and supplied using grid energy.

A drawback of using the main power to supply an IoT infrastructure with many IoT devices that depend on the main power source is the complexity of connecting the devices to the power source using cables. In the case of hundreds or thousands of devices, supplying them using the main power is impractical. If the energy from the main source is generated using fossil fuels, then the carbon footprint from the IoT infrastructure increases as its energy demands increase.

Energy storage systems

Energy storage systems are systems that are used to store energy so that it can be consumed later. It is preferable to power IoT devices using energy storage systems. One scenario is to charge the energy storage system (e.g., battery or supercapacitor) to its full capacity and then deploy the IoT device with the energy storage system as its only energy source (figure 269). In this case, when all the energy stored in the system is depleted, the device is shut down, resulting in an undesirable downtime.

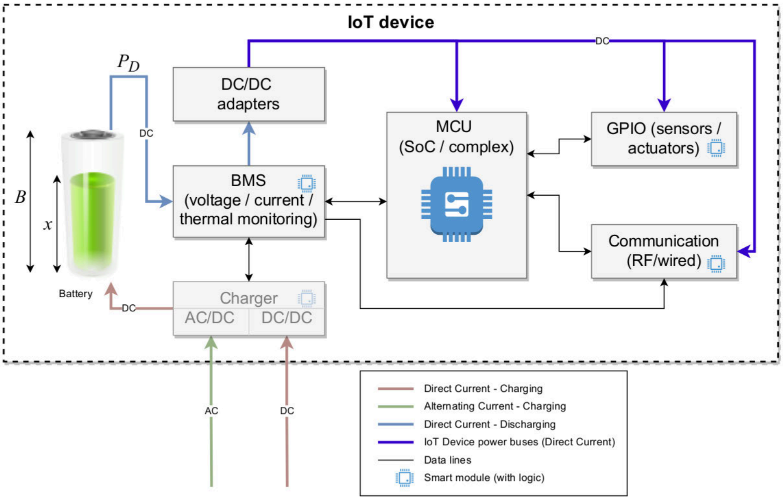

IoT device
DC
P_D
DC/DC adapters
MCU (SoC / complex)
GPIO (sensors / actuators)
DC
B
x
BMS (voltage / current / thermal monitoring)
Communication (RF/wired)
Battery
DC
Charger
AC/DC
DC/DC
AC
DC
Direct Current - Charging
Alternating Current - Charging
Direct Current - Discharging
IoT Device power buses (Direct Current)
Data lines
Smart module (with logic)

Figure 269: The architecture of an IoT device powered by a battery energy storage system [150]

The time from when the IoT device is deployed to the instant when all the energy stored in the energy storage system is depleted is called the device's lifetime. Among other factors such as mobility, scalability, and size, lifetime of the device and the energy density, energy capacity, and cycle life of the energy storage system are critical design parameters that should be considered when choosing an energy storage system to use as an energy source for an IoT device. To increase the lifetime of an IoT device and reduce the downtime resulting from the depletion of all the energy stored in energy storage systems of IoT devices, energy harvesting systems are sometimes incorporated into IoT devices.

Energy harvesting systems

Energy harvesting systems are also an alternative energy source for IoT devices. They capture energy from the environment and convert it to electrical energy to supply IoT devices. Suppose the energy captured is more than the power demand of the IoT device. In that case, the surplus can be stored in energy storage systems when the energy harvesting system cannot produce enough energy to supply the IoT device. A significant drawback of energy harvesting is that the amount of energy that can be harvested at any given time depends mainly on environmental conditions or the presence of external energy sources, resulting in a fluctuation in the amount of energy harvested over time. Hence, it is vital to carefully size the energy harvesting unit and the energy storage system in such a way as to maximise the lifetime of the IoT device. Sample architecture of a self-powered Green IoT device powered by a battery energy storage system and an energy harvesting system is present in figure 270.

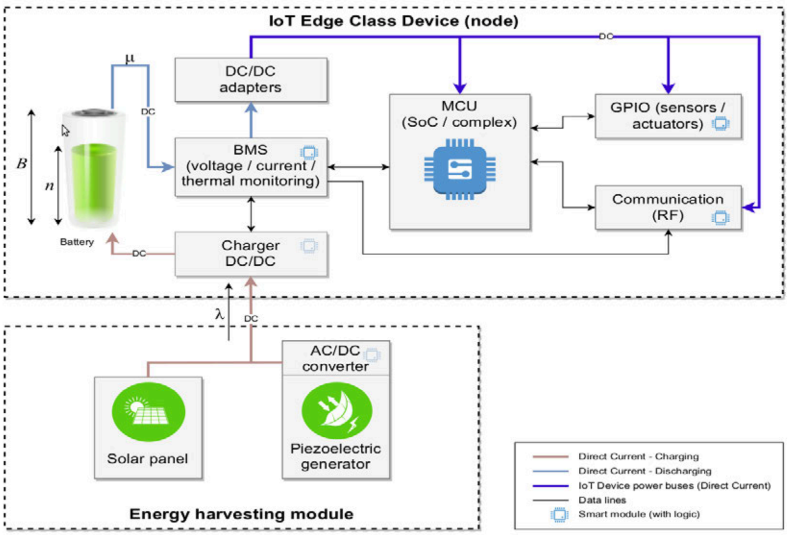

IoT Edge Class Device (node)
DC
μ
DC/DC adapters
DC
MCU (SoC / complex)
GPIO (sensors / actuators)
B
n
BMS (voltage / current / thermal monitoring)
Communication (RF)
Battery
DC
Charger DC/DC
λ
DC
AC/DC converter
Solar panel
Piezoelectric generator
Energy harvesting module
Direct Current - Charging
Direct Current - Discharging
IoT Device power buses (Direct Current)
Data lines
Smart module (with logic)

Figure 270: The architecture of a self-powered Green IoT device powered by a battery energy storage system and an energy harvesting system [151]

The kind of energy harvesting system to be deployed depends on the available energy sources (e.g., light, radio frequency, heat, vibration, wind, etc.). The amount of energy produced by most IoT energy harvesting systems is minimal compared to the energy needs of the devices. Some of the factors that influence the choice of the energy harvesting system are the availability of the energy sources, the size of the device, the energy needs of the device, and the energy density of the energy source. In a scenario where one energy source can produce sufficient energy, more than one energy harvesting system can be deployed (hybrid energy harvesting sources).

IoT Energy Storage Systems

Energy storage systems (ESS) are one of the energy sources for IoT devices. An energy storage system is a system that is designed to capture or receive energy from the energy source(s), convert it into a form that the system can conveniently store and then convert it into other usable forms of energy at a later time when the need arises. In the case of IoT devices, the electrical energy from energy harvesting systems incorporated in the IoT devices is converted to storable forms of energy (e.g., electrostatic, electrochemical, chemical, etc.) and later converted into electrical energy to power the IoT devices when needed.

In some deployment scenarios, energy is stored in an energy storage system (e.g., a battery) and then drawn to power the IoT devices. In this deployment type, when all the energy stored in the battery is depleted, the battery must be recharged or replaced; otherwise, the device will be shut down. The device's lifetime is the time from when the device is deployed to when all energy stored in its energy storage system is depleted. The energy storage system should be sized in such a way as to maximise the lifetime of the device to minimise the maintenance frequency and cost. Increasing the energy capacity of the device may result in an increase in size and price, which may be undesirable.

A possible way to increase the device's lifetime without a significant increase in cost and size is to incorporate energy harvesters to harvest energy, which is used to supply the IoT devices and store any surplus in the energy storage systems for later use. When the device is in sleep mode, most energy harvested is used to charge the energy storage system. In this case, the energy storage system can be a battery, a capacitor/ supercapacitor/ultracapacitor, or a hybrid (a combination of more than one energy storage system to exploit the benefits of each of them). In this case, the energy storage system is designed to ensure a rational balance between the energy harvesting, storage, and consumption processes and maximise the devices' lifetime.

One of the responsibilities of IoT system designers and developers is to choose appropriate energy storage systems. The choice of the energy storage system will depend on the design goals, technical constraints, and other business criteria. Some of the design requirements to be considered when designing an energy storage system for IoT include the following:

- Safety, convenience, durability (durable operations) - the energy storage system is safe (less likely to explode or become flammable). It should have a more prolonged health (should operate for long without requiring replacement).
- Energy density - Energy storage systems (ESS) with higher energy densities can store more energy per unit of mass or volume, reducing the cost, size, and weight of IoT devices, which also facilitate mobility.
- Charging speeds - Energy storage systems with fast charging speeds are preferable.
- Ability to charge the ESS with small currents since the energy harvested from IoT energy harvesters is minimal.
- Ability to deal with peak power demand - the ESS should handle peak load demand, especially during peak communication or computing load demand.
- Long-term storage - the ESS should be able to store the energy for long enough to ensure that it can power the device if the energy-generating source is absent for some time.
- Cycle life - the ESS should have a large charge/discharge cycle to ensure longer cycle life and less need to frequently maintain or replace the ESS like the case with batteries.
- Cost - ESS made from elements or minerals abundant in nature are preferable as they will be cheaper. Most batteries are made from lithium, a relatively expensive mineral compared to sodium, which is very abundant in nature. Efforts are being made to produce solid-state batteries from sodium, which may eventually lead to cheaper batteries when technologies to have these kinds of batteries mature.
- Mobility - The batteries should be lighter to facilitate mobility.

- Size - In some IoT applications, especially in smart health care, it is desirable to ensure that the device's size is as small as possible, and a small-sized ESS is required.
- Environmental sustainability - choosing the ESS in such a way as to maximise the cycle life minimises the frequency of replacing the ESS, which ensures environmental sustainability. The ESS could also be manufactured using materials that are easily disposed of.
- Scalability - choosing durable ESS ensure scalability of IoT deployments as the limitation to scalable IoT deployments is dealing with ESS-related maintenance issues.
- Little or no energy leakage - energy leakage is a significant problem, and the ESS chosen should not have high energy leakage.

Batteries

IoT devices can be powered with rechargeable and non-rechargeable batteries. The first r equires a c harger c ircuit (built-in o r a s a n e xternal d evice), w hile t he s econd is suitable for ultra-low-power devices that can operate on a single battery for a very long time. Devices with non-rechargeable batteries allow the user to replace the battery (mechanically); that is not always the case for IoT devices powered with rechargeable ones.

Non-rechargeable batteries are available in standard sizes such as AA, AAA, C, and D and coin-size ones such as LR44 or CR2032.

Rechargeable batteries include transient technologies such as Nickel-Cadmium batteries (NiCd) and Nickel-Metal Hydride batteries (NiMH), which were modern in the 1990s and the beginning of the 21st century. They are replaced with Lithium-ion (LiIon) and Lithium-Polymer (LiPo), which present higher reliability, lack of memory effect a nd h igher energy density. Still, they are also much more demanding on battery maintenance, including charging, discharging, operation temperature monitoring, and storage.
Lead-acid batteries are still common, but their application in IoT is limited due to their size and weight (low energy density), so they usually work as backups, e.g. in the context of green energy storage.

LiPo

Lithium polymer battery is a subtype of lithium ions. A single cell of the Lithium Polymer battery is usually in the form of a flat c uboid (figure 27 1). Th e si ngle ce ll's standard reference voltage is 3.7V. When fully charged, the cell reaches 4.2V and should never be charged over this limit. On the other hand, LiPo cells cannot be discharged below 3.3V (some to 3.0V). The discharge curve is predictable and common for both LiPo and LiIon. A sample discharge curve is present in the figure 274. Thanks to it, it is possible to estimate the remaining energy.
Single-cell voltage is low for most applications, so serial-connected cell stacks (battery packs) are used. Serial connections used to be referenced with S (capital letter s). So, e.g. 3S represents a battery composed of 3 cells, connected in serial. In the case of the serial connection, the battery's voltage is the sum of each cell. Thus, 3S represents a battery of 3×3.7V=11.1V (reference) or 12.6V when fully charged and 9.9V when fully discharged.
In the case of charging the serial-connected battery packs, charging requires a separate balancing of each cell and usually requires a so-called microprocessor charger. RAW battery packs composed of more than 1 cell have two terminals: main and auxiliary for load balancing (figure 272). The connection for charging the sample 5S battery is present in figure 273.

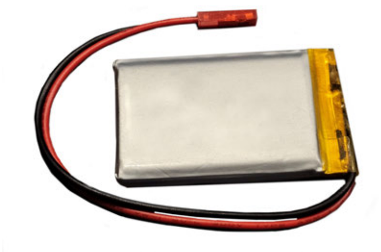


Figure 271: Sample 1S LiPo battery cell

Figure 272: Sample 850mAh 3S LiPo battery pack

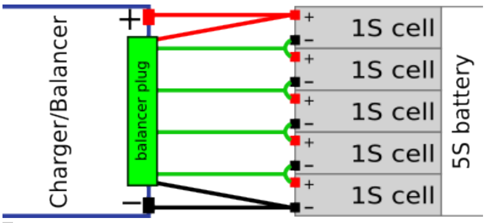


Figure 273: 5S charging connection schematics

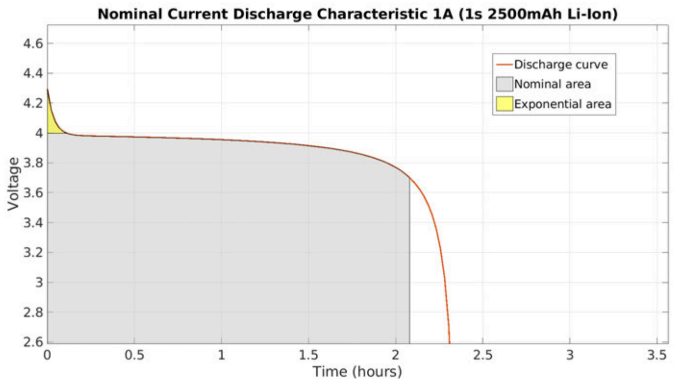


Figure 274: Discharging curve for 2.5Ah single cell (1S) LiPo cell

If the battery is broken, you can observe cracks, bends, or swollen; do not use it, discharge fully and recycle.

Never discharge LiPo battery below 3.0V on normal use.

LiPo batteries are very fragile, and overcharging usually finishes with fire and explosion.

Do not store LiPo batteries fully charged. They should be stored semi-charged with some 3.7-3.8V per cell.

LiIon
Lithium Ion batteries are widely used in electronic equipment nowadays. Their physical form is similar to LiPo ones, but there are also cylindrical units. Similarly to LiPo, LiIon single cell is nominal 3.6V or 3.7V. Charging and discharging require advanced control, and all warnings mentioned above regarding charging, discharging and maintenance of the LiPo cells also apply to LiIon.
The popular model for LiIon cell is the 18650 (figure 275) - the number comes from its dimension: a cylinder 18mm wide and 65mm high. Typical capacity is 2000-2500mAh per single 18650 cell.

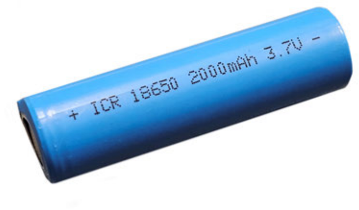


Figure 275: Sample 18650 cell

Besides the 18650, other sizes are available, such as 14500 (similar to AA size battery) with a capacity of hundreds of mAh or 26650 with a capacity exceeding 10000mAh, designated for high-rate applications such as actuators.

LiIon and LiPo batteries are very fragile to temperature changes. Charging those batteries when too cold or hot can cause battery and device damage, fire and related explosions.

Critical applications, such as use in electric cars, involve advanced cooling and heating systems to ensure optimal battery pack performance and charging conditions.

BMS

Most rechargeable batteries require a Battery Management System (BMS) that controls the charge and discharge of the RAW cells. It is essential, particularly in the case of the Lithium Polymer batteries and Lithium Ion ones.
BMS prevents overcharging and over-discharging and sometimes controls battery temperature, limiting charging current if the battery is overheating. Its general purpose is to keep the battery in good condition for a long time and prevent battery damage.

Overheating and over-charging may cause battery damage, fire and explosion!

Raw LiIon and LiPo cells are commonly available, and there are also protection and charging modules in the form of electronic PCBs for self-assembly, e.g., a dedicated module for an 18650 LiIon cell as in figure 276 and figure 277. BMS can also be integral to the IoT device's power module, e.g. figure 278. Those boards usually contain DC-DC converters, providing a stable voltage of 3.3V and/or 5V for powering IoT devices.

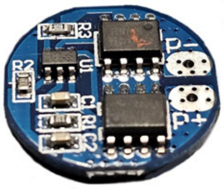


Figure 276: Protection module

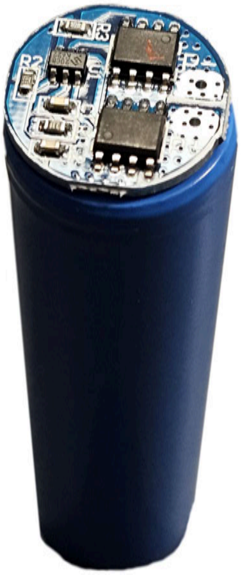


Figure 277: 18650 cell with protection module applied

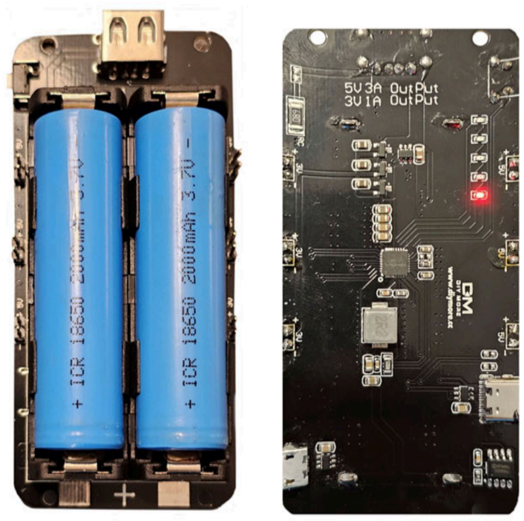


Figure 278: Integrated power module (BMS) with charger, discharging protection and voltage stabiliser for 3.3V and 5V rails. 2×18650 cells

Capacitors, supercapacitors, and ultracapacitors

Alternative energy storage systems that can be deployed to compensate for the limitations of batteries are capacitors, supercapacitors, or ultracapacitors. They can be deployed alongside batteries (hybrid deployment) and may eventually replace batteries in some IoT deployments. Some of the limitations of batteries that can be resolved with the use of capacitors, supercapacitors, or ultracapacitors are

- Limited cycle life - the limited cycle life requires that batteries should be replaced frequently, resulting in multiple challenges such as high and tedious maintenance costs (as it is difficult to service a vast number of IoT devices to replace or charge the batteries), degradation of the quality of service (as devices can be shut down when all the energy stored in batteries is depleted), and challenges in disposing of batteries (as vast amounts of batteries are required to be disposed of).
- Inability to handle peak power load demand - Small batteries are often not able to handle peak power load demands (which may result from peak communication or computing loads), which will require that the battery should be discharged at a higher C rate, which may be unhealthy or detrimental to the battery.
- Slow charging and discharging process — Batteries' charging and discharging speeds are relatively slow compared to those of capacitors, supercapacitors, and ultracapacitors.
- Charging and discharge inefficiencies - The magnitude of the energy harvested from

the ambient environment or external sources using the small energy harvesters in IoT devices is very small (in the order of a few hundred microwatts or milliwatts) to charge batteries but can effectively charge capacitors, supercapacitors, and ultracapacitors due to their high charging and discharging efficiencies.

- Sustainability challenges — Since batteries may be replaced regularly due to their short lifetime, there is a growing challenge of disposing of battery waste without causing significant environmental damage. Some materials used to make batteries are toxic to the environment, and frequent disposal of large amounts of batteries poses a potential environmental risk.

There is an increase in the adoption of capacitors, supercapacitors, or ultracapacitors as alternative energy storage systems in IoT devices due to their advantages as alternative energy storage systems. Some of their advantages include:

- Longer cycle life — The cycle life of capacitors, supercapacitors, or ultracapacitors is far greater than that of batteries. So, there is no need to frequently replace them, reducing maintenance costs and e-waste generated by frequently changing batteries. Supercapacitors can reach up to one million charge/discharge cycles, eliminating the limited cycle life problems often experienced when using batteries as energy storage systems.
- High power densities — The high power densities make it possible to charge them with small currents (since the amount of power produced by IoT energy harvesters is very small) and also handle peak power load demands (which require the delivery of relatively large power to the IoT devices).
- Sustainability - Since there is no need to change the energy storage systems frequently, the amount of waste produced is relatively small. The supercapacitors are also made from materials that can be easily recycled.
- Faster charging and discharging speeds - Capacitors, supercapacitors, and ultracapacitors can be charged relatively fast compared to batteries.

Although using capacitors, supercapacitors, and ultracapacitors has many advantages compared to batteries, they also have limitations.

- Inability to store energy for long — One limitation of this type of energy storage system is that it does not keep power for long, resulting in the short lifetime of the IoT devices (the time required to deplete all the energy stored in the capacitors, supercapacitors, and ultracapacitors).
- Size and cost limitations — One possible solution to the problem of short device lifetime resulting from the quick discharge of capacitors, supercapacitors, and ultracapacitors is to increase the energy storage capacity. However, this will increase the size and cost of the IoT devices, which is not desirable in most IoT applications, as devices are required to be as small as possible.
- Decrease in energy capacity - When a supercapacitor reaches the end of its life, its energy capacity may drop to about 70% of its original value, limiting its ability to meet the energy storage needs of IoT devices.
- Energy losses — They suffer from energy losses resulting from internal energy distribution and current leakage, which result in the wastage of the energy harvested and stored. Power leakage leads to low utilization of the harvested energy, and a portion of the harvested energy leaks away instead of powering the IoT devices.

Unlike batteries, supercapacitors have a lower energy density but do not suffer from cyclic degradation, similar to the case with battery cells. Ceramic capacitors are often used as an energy storage system to store energy harvested by energy harvesters incorporated into IoT devices because of their low degradation, low current leakage, and expected increase in energy densities. Thus, they are an excellent candidate to be adopted as energy storage systems deployed in industrial IoT devices [152] and IoT devices in other sectors.

Other energy storage systems

Although electrochemical energy storage systems (e.g., batteries) and electrostatic energy storage systems (e.g., capacitors, supercapacitors, and ultracapacitors) are the most popular energy storage systems used to store energy to be used to power IoT devices, other energy storage systems can be used, especially at the transport layer (internet access and core networks) and fog and cloud computing layer. Some of these other energy storage systems may not be convenient for IoT devices. Some of them include the following:

- Chemical energy storage systems - chemical energy storage systems convert the electrical energy delivered to them into chemical energy, which can then be converted into electrical power to supply the IoT systems later. One popular example of a chemical energy storage system is the hydrogen energy storage system. In a hydrogen energy storage system, electrical energy is converted into hydrogen, which is then stored. One of the approaches often used to produce hydrogen is water electrolysis, which produces hydrogen and oxygen. The hydrogen is then stored and later used as fuel in a fuel cell to generate electricity to power the IoT infrastructure (e.g., base stations and data centres). Compared to battery energy storage systems and supercapacitors, battery energy storage systems are inefficient as much energy is wasted. However, much research is being conducted by major energy and car companies and academic institutions to improve the efficiency of hydrogen energy storage systems, as it is expected that hydrogen energy storage systems should be among the top innovative technologies for future green economies.
- Mechanical energy storage system - mechanical energy storage systems can convert electrical energy into mechanical energy (potential or kinetic energy), which can then be converted into electrical energy to power IoT systems later. The most popular mechanical energy storage systems include pumped hydro, flywheels, and gravity energy storage systems. Mechanical energy storage systems are simple to design, as this technology has existed for hundreds of years. One of the limitations is that they have very low energy density and are also very inefficient.

Hybrid energy storage systems

The various energy storage systems that we have discussed above have their advantages and drawbacks. One possible way to exploit the advantage of some energy storage systems and eliminate the limitations imposed by some energy storage systems is to deploy more than one energy storage system. An energy storage system that consists of more than one energy storage system is called a hybrid energy storage system. The deployment of hybrid energy storage systems (more than one energy storage system) improves the overall performance of the energy storage system in terms of energy density, reliability, and the cycle life (or lifespan) of the energy storage system. It also reduces the overall cost of the energy storage system.

In IoT devices, batteries and supercapacitors can be deployed as a hybrid energy storage system. The advantage of supercapacitors is that they can be charged faster, even with small currents (typical of the currents delivered by IoT energy harvesting systems). The supercapacitor can also handle peak power loads and have a longer cycle life than batteries. The limitation of supercapacitors is that they cannot store energy for a long time, but batteries can store energy for a long time. Therefore, a battery and supercapacitor can be installed in an IoT device to provide a hybrid energy storage system that takes care of the limitations of both and exploits their advantages to offer improved performance.

Batteries are often used as an energy storage system in base stations and cloud data centre sites powered by renewable energy. Due to the limitation of cycle life and power density, a hybrid energy storage configuration consisting of a supercapacitor and battery can be considered. Another kind of configuration is a battery and a hydrogen energy storage system. When the battery is full, any additional energy harvested is lost (in some installations, it is passed through a dumb load to dissipate it).

In a battery-hydrogen hybrid energy storage system configuration, when the battery is full, the additional electrical energy harvested is used in water electrolysis to split water molecules to produce hydrogen and oxygen. The oxygen is then stored and later used as fuel in a fuel cell to generate electricity when necessary. This type of hybrid energy storage is beneficial for seasonal energy storage where during the season when the conditions for energy harvesting are favourable (e.g., during summer), a lot of energy is harvested and stored in the form of hydrogen, which is then used during winter to generate electricity when the energy harvesting is not enough.

Converters for IoT Powering

Power Conversion

Power sources tend to provide energy in a form that is not straightforwardly acceptable to IoT devices.
Wall sockets provide relatively high voltage alternating current (AC) that needs to be lowered and converted into DC, also stabilised as required by MCU, which is fragile for voltage variations. Common conversion flow for AC sources is present in figure 279. DC power sources (such as batteries) also require voltage conversion and stabilisation. The flow is present in figure 280.

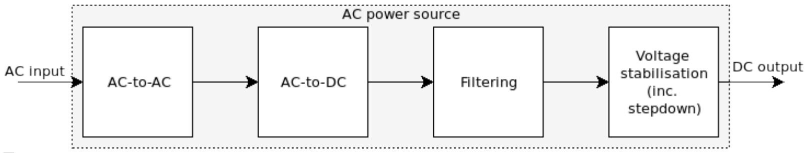


Figure 279: AC power source conversion flow

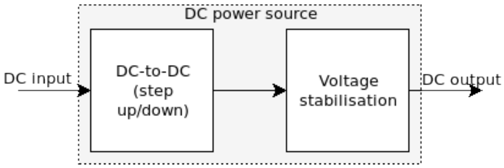


Figure 280: DC power source conversion flow

Common voltage conversions are:

- AC-to-AC conversion,
- AC-to-DC conversion,
- filtering,
- DC-DC conversion and voltage stabilisation.

Stabilisation usually is included as a part of DC-DC or AC-DC conversion.

AC-to-AC
AC-to-AC conversion is used whenever a high-voltage source is available and is required to lower it, typically somewhere between 12V and 5V.
Historically, AC-to-AC conversion was implemented using a transformer (symbol in figure 281). This technique has serious drawbacks:

- transformer-based converters are heavy as they use copper coils and steel cores (sample transformer in figure 282),
- the conversion rate is fixed, thus requiring different transformers in the countries with different socket voltages,
- require separate AC-to-DC conversion module (Graetz bridge at least).

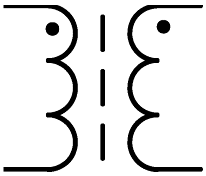


Figure 281: Transformer symbol

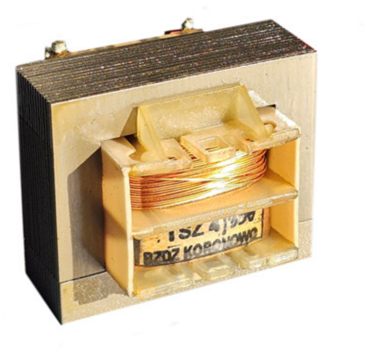


Figure 282: Sample high to low voltage transformer

Modern converters use a switching-mode power supply (SMPS) without a transformer, just a small coil. Those converters used to be much more complex, but nowadays, most of the circuit is implemented in a single, integrated chip. Currently, the cost of the SMPS is much lower than the transformer-based one. SMPSes are:

- much lighter and compact (do not involve the use of a transformer),
- usually integrate AC to DC conversion in a single circuit,
- accepting a wide range of input AC and DC voltages, thus can be used virtually worldwide and in various environments.

AC-to-DC
In general, IoT devices use DC to power MCUs and peripherals. A classical AC-to-DC conversion involves a Graetz's bridge with 4 diodes (schematic in figure 283), currently implemented commonly in a single enclosure as in figure 284.

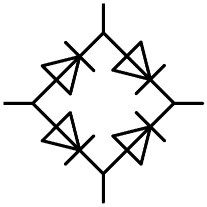


Figure 283: Graetz bridge

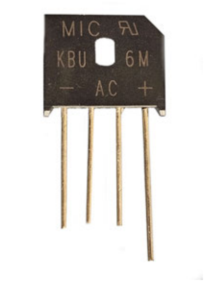


Figure 284: Graetz bridge in single enclosure

SMPS is used to integrate all necessary functions (including voltage stabiliser) in a single device, e.g. in figure 285.

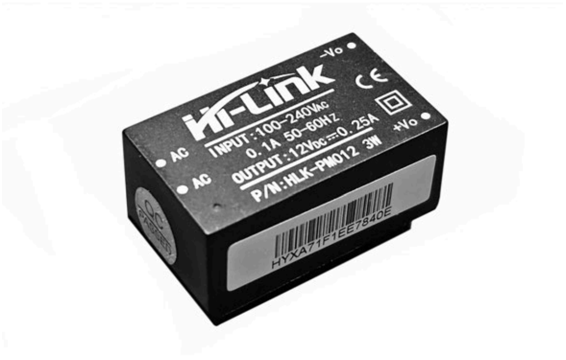


Figure 285: Sample AC to DC converter with voltage stabilised output

Filtering
Proper filtering of the current interferences is essential to ensure stable MCU operation. Even when using good quality power sources, nearby communication wires, power wires, and electromagnetic fields generated by actuators can cause severe interference voltage rise and drop. For this reason, the use of capacitors is essential. A rule of thumb is to add a large capacity (e.g. 1000uF) capacitor on the power bus and 100nF capacitors close to the IoT device's MCU and other sensitive components. It may be specific to the device so unstable work may require analysis of the power bus interferences regarding their frequency and amount.

DC-to-DC and voltage stabilisation
The DC-to-DC conversion is needed whenever the source voltage is unsuitable for an IoT device. It is also needed in the case of batteries as a second stage after AC-DC conversion. DC-DC converters involve voltage stabilisation.
Modern DC-to-DC converters are implemented with fixed output voltage or regulated to decrease (step-down) or increase (step-up) the voltage. Some circuits can implement both: step-up-down, where voltage is controlled with a regulator (usually a potentiometer).

Former solutions include linear voltage stabilisers (only step-down), e.g. popular and still used 78xx chips. Sample 5V stabiliser 7805 is presented in figure 286. Depending on their application and maximum current, linear stabilisers are available in various enclosures. They have several drawbacks, however:

- low efficiency,
- limited current,
- overheating - they require a radiator even for relatively low currents, e.g. over 1A,
- fixed voltage output (without the use of the external components).

Their advantage is that they are much easier to embed into the circuit as use requires only a few passive components. The sample application circuit is quite simple and present in figure 287.

Figure 286: Linear voltage regulator

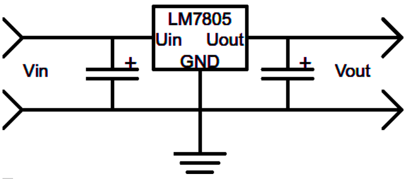


Figure 287: Linear voltage regulator application circuit

Modern DC-DC converters are of high efficiency, easily going over 90%. They are implemented as switching regulators rather than linear. The construction of the switching converters is quite complex. Sample device with fixed voltage regulation is present in figure 288 and the one with variable voltage, in figures 289 and 290, where the output voltage can be set using a potentiometer.

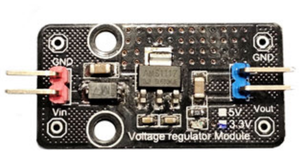


Figure 288: Fixed voltage step down converter module

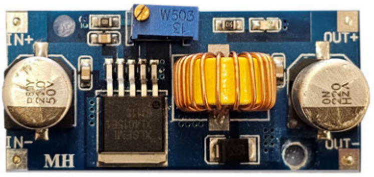


Figure 289: Variable voltage step-down converter module

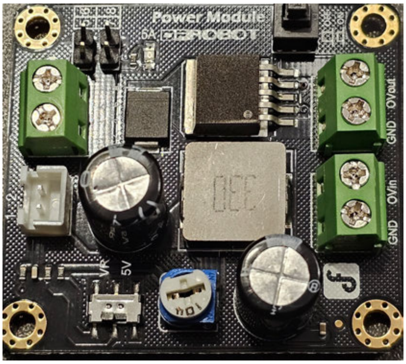


Figure 290: Variable voltage step-down converter module with additional 5V utility power source

Green Energy Sources in IoT

Powering IoT devices using energy storage systems (e.g., batteries or capacitors/ supercapacity/ultracapacitor) faces some challenges, such as the limited lifetime (the time from when an IoT device is deployed to when all the energy stored in its energy storage system is depleted or consumed), maintenance complexity, and scalability. In an IoT infrastructure with massive numbers of IoT devices (e.g., massive IoT), frequent change of IoT batteries results in maintenance complexities, high cost, and sustainability challenges. One of the solutions to these challenges is using energy harvesters to harvest energy from the ambient environment or external sources (e.g., vibrations or human body sources) to power the IoT devices. Energy harvesting is capturing energy from the ambient environment or external sources, which is then converted into electrical energy that can be used to power IoT devices or stored for later use.

IoT end node devices (Edge class) are usually powered with low current and voltage. This raises new capabilities to use green energy sources, which is essential in particular in distant and remote locations (e.g. earthquake sensors).

When selecting a renewable energy source, it is essential to consider:

- An energy budget - is the renewable energy source able to deliver enough energy for the duty cycle of the IoT device?
- Is there a need to provide a backup energy source to ensure continuous operation of the IoT device?
- How do ageing and time (daytime, season) affect the energy received from the green energy source?
- What is the cost of the green energy source compared to other powering opportunities?
- Is there an AC needed?

Answers to those questions drive a selection of the green energy source, which always regards a specific duty cycle and working conditions of the IoT device.
A short characteristic of selected green energy sources can help during powering design.

Energy harvesting from ambient sources

The energy can be harvested from ambient sources (environmental energy sources) such as solar and photovoltaic, Radio Frequency (RF), flow (wind and hydro energy sources), and thermal energy sources. Ambient energy harvesting is the process of capturing energy from the immediate environment of the device (ambient energy sources) and then converting it into electrical energy to power IoT devices. The ambient energy harvesting systems that can be used to harvest energy to power IoT devices, access points, fog nodes or cloud data centres include:

- Solar and photovoltaic energy harvesting: capturing natural light energy (in the case of light) or artificial light (in indoor deployments) and converting it into electrical energy to power IoT devices.
- Radio frequency (RF) energy harvesting: Capturing RF energy from the environment and converting it into electrical energy to power IoT devices.
- Flow energy harvesting: Converting the energy generated from airflow (e.g., wind energy harvesting) or water (e.g., hydro energy harvesting) into electrical energy to power IoT or other IT infrastructures.
- Thermal: Capturing the energy generated from temperature differences and converting it into electrical energy to power IoT systems and other IoT infrastructures.
- Acoustic noise: Capturing the energy from the pressure waves produced by a vibrating source and converting it into electrical energy to power IoT devices.

Solar and photovoltaic energy harvesting

So far, solar energy is the easiest and most widespread option to power remote IoT devices. It is available virtually worldwide, simple to implement and integrated with other energy resources.

Solar energy is grabbed using solar panels. Solar panels deliver DC. Solar panels work best in mid-temperature (overheating decreases efficiency) and when located perpendicular to the solar rays. As the sun changes its position during the day and during the season, it is important to ensure the correct angle to maximise the solar exposition possible when using a fixed mounting of the solar panels. Some active trackers can follow the sun's location in the sky and change the angle of the solar panel accordingly, but that requires an additional control system and extra energy. Tracking is usually unsuitable for small, low-powered IoT devices such as sensors.

Depending on the region, the weather (primarily clouds, snow and rainfalls) seriously impacts panels' efficiency and, thus, the amount of energy available. For this reason, it is common for solar panels to be oversized and equipped with backup energy storage, such as a battery pack. Moreover, in subpolar and polar regions, daylight is very short during the winter or even unavailable for latitudes beyond the polar circle.

Radio frequency (RF) energy Radiofrequency (RF) energy harvesting is among the most popular energy harvesting technologies developed to power self-powered IoT devices like IoT RFID tags and smart cards. The RF electromagnetic is captured and converted into electrical energy, which is then used to power the IoT devices or stored in a battery or capacitor/supercapacitor/ultracapacitor to be utilised later. Specialised antennas (including RF input filter and impedance matching network) are used to capture RF signals rectified by passing them through a rectifier, which rectifies the RF signals, converting the RF power into DC power. The DC power can then be used to power IoT devices or stored for later use.

The sources of RF signals could be from mobile cellular networks, radio and television wireless transmitters, and other Wireless access points (e.g., WiFi). The RF energy harvesting is influenced by the frequency of the signal, antenna gain, and the distance of the device from the source of the RF signal (e.g., the distance of the IoT device from a cellar base station, especially in situations where a cellular base station generates the RF signal). Although the amount of energy harvested from RF sources is relatively tiny, RF energy harvesting systems can easily be implemented. RF energy is readily available, making RF energy harvesting a cheaper and more convenient energy harvesting solution for power tiny and low-power (less energy-hungry) IoT devices. A significant possible drawback of RF energy harvesting is that when millions or tens of billions of RF-powered IoT devices are deployed in a given environment, RF energy harvesting may pose a health risk.

Flow energy

1. **Wind energy**
Wind energy is grabbed using a wind turbine, which converts rotation into a magnetic field that generates electric energy. Raw turbine delivers AC, which must be converted into the DC suitable for IoT devices. This conversion drops efficiency.

Wind energy is weather-dependent, so it is usually not a single energy source but works in parallel with other sources and frequently requires backup energy storage. Winds too strong for a turbine may cause damage; thus, the turbine has to be switched off in such cases. When the wind is too low, it cannot push the propellers, so energy is not generated.

Wind turbines tend to be big, and as they contain complex mechanics (blades, rotor, gear, generator), they require inspection and maintenance. Thus, they are not suitable for the “set and forget” IoT applications.

2. **Hydro energy**
Water energy is considered a stable energy source, eventually depending on the season. Its advantage is the ability to generate energy for a whole day, regardless of the day and night. Water energy is complex in use, however, because it uses additional infrastructure (e.g. pipes that deliver water).

Water turbines work with similar principles to wind turbines but use water to push the propellers instead of wind. Water turbines generate AC, so that needs to be converted to DC. Because of their size and the need for maintenance, they share a similar development area as wind turbines.

Water can also be considered as a backup energy battery regarding gravity: during the energy overhead, it can be pumped with that energy up, and then, thanks to the gravity and use of water turbines, this energy can be re-used when there is a lack of other energy resources. This process is used on a large scale and is known as a “pumped storage power plant”. However, principles remain scalable; they involve complex infrastructure and other (usually green) energy sources such as wind or solar.

Due to complexity, water turbines are not the first choice to power small IoT devices but rather to set up a local medium-scale energy source or support the grid.

Water has recently been considered a medium to generate hydrogen using external energy sources (such as solar panels or wind). When energy is available, hydrogen is generated and stored in tanks; later, it is used for energy generation using fuel cells and converted back from hydrogen and oxygen into water. Similar to the aforementioned pumped storage power plant, this solution delivers clean energy storage but also requires complex and extensive infrastructure. Hydrogen is also an explosive gas.

Thermal energy harvesting

Thermal energy harvesting is the capture of thermal energy and conversion into electrical energy to power IoT devices or store it for later use. Thermal energy is readily available in the environment (at home, in factories, and in regions with high temperatures). Some heat sources include car engines, geothermal heat from the ground, and heart from industrial operations. With the use of thermoelectric generators, thermal energy is captured and converted into electrical energy to power IoT systems or to store for future use.

Geothermal energy is considered to be very constant but of low availability. Its application is based on steam and hot water conversion to electrical energy, usually via high and low-pressure turbines. Due to the complex processing involving dealing with high temperatures (e.g. overheated steam of >200C)[153], it is suitable for mass-scale energy production for a grid rather than as a small energy source to power a single IoT device.

Energy harvesting from external sources

Below is a short list of energy harvesting characteristics from non-ambient, in general, external sources.

Energy harvesting from mechanical sources

- Vibration energy harvesting - harvesting the energy created by vibrations (e.g., due to car movements, operations of machines, etc.) and converting it into valuable electrical energy, which can be used to power IoT devices or stored in the battery for later use.

- Pressure energy harvesting - harvesting the energy from pressure sources and converting it into useful electrical energy.
- Stress-strain energy harvesting - harvesting energy from mechanical vibrations by exploiting the property of some materials (e.g., piezoelectric materials) that, when subject to mechanical strain, produce an electrical charge proportional to the stress applied to it.

Energy harvesting from human body sources

Human body energy harvesting is harvesting energy from the human body and then converting it to electrical energy. It is used to power wearable IoT devices, especially IoT devices designed for smart health applications. The energy source could be the vibration or deformations created by human activity (mechanical energy). The energy source could be from human temperature differences or gradients (thermal energy) or human physiology (chemical energy).

- Human activity energy harvesting - capturing the biomechanical energy resulting from human activities (walking, cycling, running, and other exercises) and then converting it into useful electrical energy that can be used to power the IoT devices or stored for later use.
- Human physiological energy harvesting - capturing the biochemical energy resulting from human physiological processes and then converting it into electrical energy that can be used to power IoT devices, especially medical implantable IoT devices.

6. Introduction to the IoT Communication and Networking

Mind, there is “I” in IoT!

There is no doubt that IoT is network-oriented: even the name IoT naturally relates to the Internet network. Communication is an essential part of IoT ideas. Every IoT device must somehow communicate, even the simplest, passive RFID tag – it responds with some data to the excitation.
Communication is always performed with some rules known for both communicating parties. People have different languages to use, and devices have protocols. Communication protocol describes how to address the information to the remote device, encode the data, and check the incoming message's correctness. The physical layer of the protocol description also tells how to transmit every bit of data, the frequency of radio waves, how fast we can send the data and the maximum range of the transmission. Those duties are pretty challenging to address in the context of the IoT, constrained devices.

Communication in IoT devices can be wired or wireless:

- End node (edge) devices mainly use wireless transmission.
- Fog-class devices use both, particularly as gateways or routers.
- The cloud segment of the IoT ecosystem extensively uses wired copper and optical (fibre).

IoT networking differs significantly from typical, multilayered, stack-oriented TCP/IP or similar communication models we know while using our PCs, MACs, servers or smartphones. Indeed, constrained IoT devices are usually unable to operate regularly – full time on ISO/OSI layered stack because of constrained resources. In detail, it primarily means IoT devices are limited by processor power, RAM and storage sizes, mainly because of limited power resources. IoT devices are expected to be energy efficient and thus low-powered, which usually excludes typical wireless connection standards, e.g., WiFi. On the other hand, IoT devices are expected to communicate over long distances – some couple or a dozen kilometres – where wired infrastructure like Ethernet cables and related infrastructure is non-existent and most of the wired technologies, copper-based, are out of range.

Also, the daily life-cycle of IoT devices is much different from that of PC life-cycles. We as humans used to switch on the notebook, work extensively on the web, then turn it to low power or off, making the machine sleep, hibernate or just shut it down. And we wake it up when needed. It barely makes network operation during sleep. IoT devices are expected to be sleeping, providing low power mode whenever possible, and on the other hand, they're supposed to be fully operable when only needed. Most performed IoT tasks related to sensing have a cyclical nature, e.g. measuring gases as a sensor-network node. In contrast, the period can be between seconds and months or even longer. They're usually expected to trigger themselves to be awake from sleep, perform some operation and connect to the network. Meanwhile, the network grid needs to be aware that those devices are still in their place, in good condition and able to awake (e.g. the battery is not being drained).

Because of the existence of different IoT devices, including those very constrained from 8-bit processors with some kB of the RAM to 32-bit multicore machines well-replacing PCs, IoT networking is very competitive on protocols, approaches and solutions. Standardisation organisations like IEEE indeed introduce some networking standards, yet they are competed by large manufacturers forcing their complex solutions, including dedicated hardware, software and protocols. The third force driving this market is open solutions and enthusiasts, usually working with cheap equipment, providing de-facto standards for many hobbyists and industries.

An interesting survey made by RS components [154] shows 11 wireless protocols used in IoT. Some of them you can use for free without having any license to purchase, while others are proprietary, and some need a subscription plan.

The following chapters explain some of the most popular concepts about organising networks, fulfilling the above constraints on communication between IoT devices (Machine-2-Machine) and how to let them communicate with the Internet, including hardware, software and human users. We focus on the de-facto standards existing on the web, usually as open-source libraries and somewhat low-cost devices.

6.1. Communication Stack

IoT devices are not separated from the global networking environment that nowadays is highly integrated, connecting various wired and wireless transmission standards into one network called the Internet. Indeed, some networks are separated because of security and safety reasons and regulations, yet they usually share the same standards as global Internet networks.

The XXI century brought wide acceptance of wireless connections. They became prevalent even in so-called pico-networks, like your PAN (Personal Area Network)[155], implemented using, e.g. Bluetooth Connection, where your smartphone in the left pocket of your jeans is hosting a server. There are many wireless devices connected to it: your wireless headset/audio device that you're listening to music, wireless HID controller, your Smartwatch, AR glasses, etc. All those devices constitute a personal (PAN-Personal Area Network) - a wireless network cell, usually also routed via NAT (Network Adress Translation) [156] to the Internet, through some other wireless connection gateway or router, using WiFi or mobile data (3G/4G/LTE).
Many IoT devices share standard wireless protocols, models and ideas, but some are not powerful enough to integrate with the Internet. On the other hand, wireless connections are natural to IoT devices, where they are expected to operate in remote destinations, usually without any wired infrastructure. Because of it, some Internet standards were adapted to their constraints or networks of constrained devices are separated and gatewayed through more powerful devices, where protocol translation occurs. Those models are discussed below in the following chapters.

In a similar way to regular Internet networking, IoT networking is implemented using a (usually simplified) layered stack, similar to the regular ISO/OSI 7-layer networking stack well known to all IT students [157], where the lowest 3 levels constitute so-called Media layers of the stack (recommendation X.200).

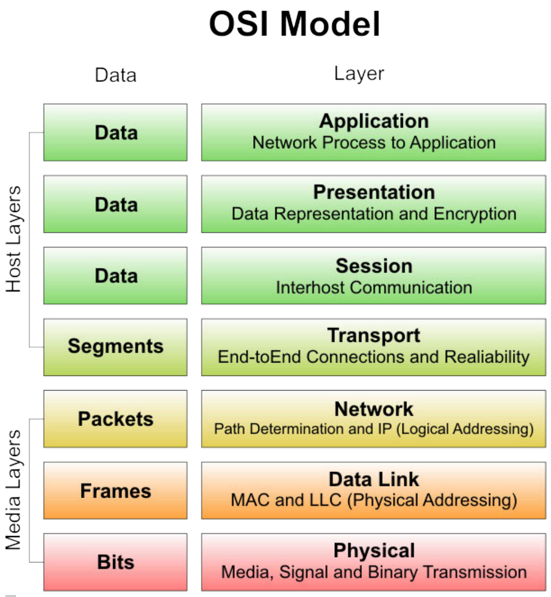

OSI Model
Data
Layer
Host Layers
Media Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-toEnd Connections and Realiability
Packets
Network
Path Determination and IP (Logical Addressing)
Frames
Data Link
MAC and LLC (Physical Addressing)
Bits
Physical
Media, Signal and Binary Transmission

Figure 291: ISO/OSI multi-layer Internet networking protocol stack

Level 1 is a Physical Layer (PHY). On top of it, level 2 is the Data Link Layer with Media Access Control and Logical Link Control (MAC/LLC). Level 3 is the Networking layer (NET) where packets are formed and routed as presented in figure 291. ISO-OSI model was designed and implemented for dedicated network controller chips and powerful processors; thus, not all IoT devices can fully implement this model, mainly because of constrained RAM and storage memory sizes. Also, this model requires an instant connection to the remote node (PHY dependent), so it strongly impacts the battery drain in the case of constrained-power devices.

A quick overview of popular communication technologies for the Internet is presented in figure 292. Note wide distance range between nodes of the Wireless devices regarding protocol used (primarily because of the PHY nature) where it varies from some meters in case of piconets up to some 180–2000 km when considering LEO (Low Earth Orbit) satellites, e.g. communicating through Iridium network and even up to about 35 786 km in case of the use of the geostationary satellites [158].

Another factor is the communication bandwidth. Fortunately, IoT devices usually do not require high bandwidth – a couple of kbps is enough; thus, almost all protocols apply here.

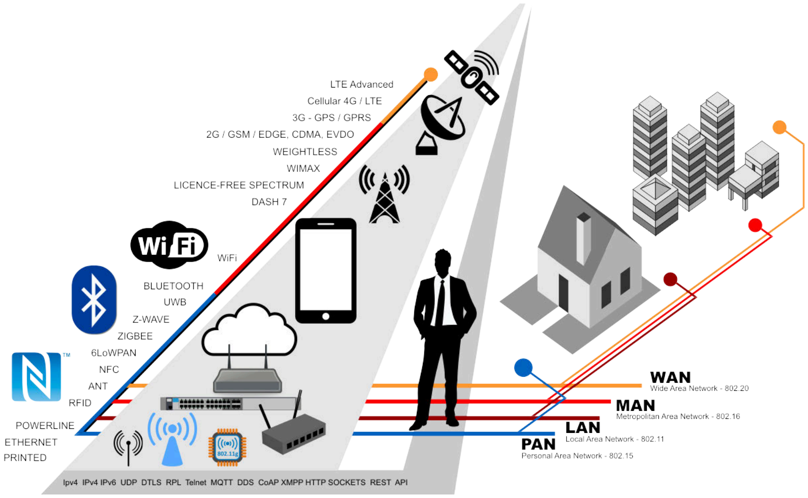
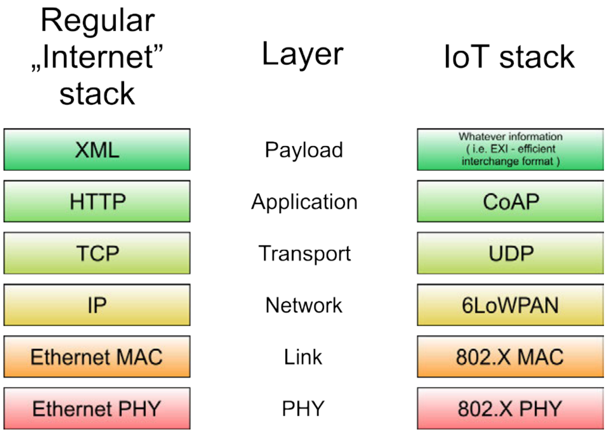

LTE Advanced
Cellular 4G / LTE
3G - GPS / GPRS
2G / GSM / EDGE, CDMA, EVDO
WEIGHTLESS
WIMAX
LICENCE-FREE SPECTRUM
DASH 7
WiFi
BLUETOOTH
UWB
Z-WAVE
ZIGBEE
6LoWPAN
NFC
ANT
RFID
POWERLINE
ETHERNET
PRINTED
WAN
Wide Area Network - 802.20
MAN
Metropolitan Area Network - 802.16
LAN
Local Area Network - 802.11
PAN
Personal Area Network - 802.15
Ipv4 IPv4 IPv6 UDP DTLS RPL Telnet MQTT DDS CoAP XMPP HTTP SOCKETS REST API

Figure 292: Popular wireless networking standards

In many cases, IoT remote, distant nodes do not need constant communication, e.g. weather sensing would better communicate on a datagram communication model (UDP rather than TCP) [159]. In such cases, IoT devices utilize a simplified IoT stack, as presented in figure 293.



Regular „Internet" stack	Layer	IoT stack
XML	Payload	Whatever information (i.e. EXI - efficient interchange format)
HTTP	Application	CoAP
TCP	Transport	UDP
IP	Network	6LoWPAN
Ethernet MAC	Link	802.X MAC
Ethernet PHY	PHY	802.X PHY

Figure 293: Simplified, IoT-oriented implementation of the protocol stack (using UDP)

6.2. Communication Models

IoT Devices can be classified regarding their ability to implement full protocol stacks of the typical Internet protocols like IPv4, IPv6, HTTP, etc.

- Devices unable to implement full protocol stack without external support, like, e.g. Arduino Uno (R3) with 32 kb of flash memory, 16 MHz single core processor and 2 kB of static ram, battery-powered, consuming some couple of mW while operating.
- Devices that can implement full protocol stack yet are still limited by their resources, e.g. ESP8266 and ESP32 chips, battery-powered, consuming some dozen or hundred of mW while operating.
- Devices that can offer various advanced network services, capable of efficiently implementing protocol stack yet not servers, routers or gateways, e.g. Raspberry Pi and its clones. Usually, DC powered with power consumption far above 1–2 W, usually up to 10–15 W.
- Dedicated solutions for gateways and routers, usually with embedded, hardware-based implementations of the switching logic, utilising some 10-50W of power.
- Universal IoT computers (e.g. Intel IoT), using PC-grade processors (x86, but sometimes ARM), using some about up to 100 W of power.

Some IoT networks are also constrained by the number of IP addresses available regarding the number of IoT devices one needs to connect, so their topology is *a priori* prepared as NAT (Network Adress Translation) solution [160] thus it requires automatically use of routers.

IoT devices are usually expected to deliver their data to some cloud for storage and processing while the cloud can send back commands to the actuators/outputs.

Finally, security concerns usually put the IoT devices in some separate sub-network and guarded by a firewall.

The abovementioned limitations bring 3 communication models available regarding specific IoT ecosystem requirements.

Device to Device and Industry 4.0 Revolution

The device-to-device communication model, sometimes referenced as M2M (machine-to-machine communication model), used to be implemented between the homogeneous class of IoT devices. Nowadays, there is a need to enable heterogeneous systems to collaborate and talk one-to-another. In a device to a device model, communication is usually held simple, sometimes with niche, proprietary protocols, i.e. ANT/ANT+ [161], sometimes employs heavy protocols like XML, so there is a need to provide standard communication ontologies and semantics. Devices participating in such networks usually act as multimode, constituting self-organising networks, capable of exchanging the data through routing and forwarding as it appears in 6LowPAN networks where nodes may not only act as data producers/consumers but are also expected to act as message forwarders/routers.

The device-to-device model is highly utilised in Industrial Automation Control systems and recently very popular in developing Industry 4.0 (I4.0) solutions, where manufacturing devices, i.e. robots and other Cyber-Physical systems (CPS), communicate to set operation sequences for optimal manufacturing process (so-called Industry 4.0) thus providing elastic working zones along with manufacturing flexibility and self-adaptation of the processes. It happens because of the presence of various IoT devices (here, sometimes referenced as Industrial IoT) and advanced data processing, including Big and Small data. Such device-to-device networks frequently mimic popular P2P (peer-to-peer) networks, where one device can virtually contact any other to ask for information or deliver one. Compared to the classical, tree-like topology, device-to-device communication constitutes a graph of relations rather than a hierarchised tree. Figure 294 presents comparison between pre-I4.0 (Industry 3.0) and I4.0 flow. Along with physical (real) devices participating in the manufacturing process, there usually goes their virtual representation (“digital twin”) to enable cognitive manufacturing based on data science. The detailed description of the data analysis and its use in I4.0 is out of the scope of this book, however.

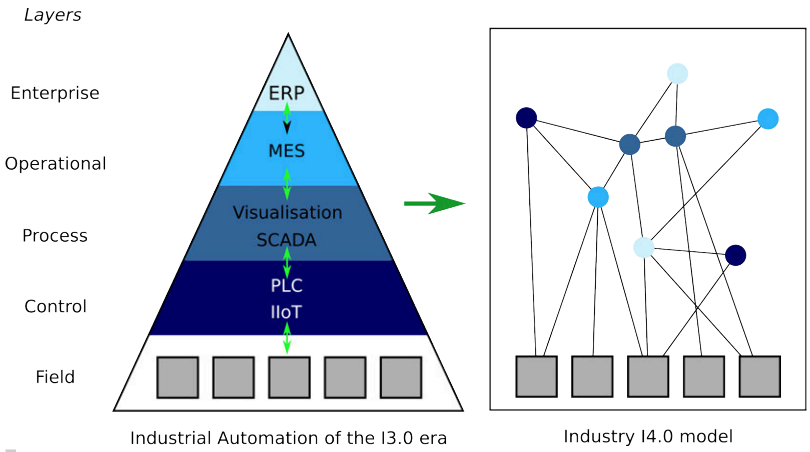


Figure 294: Industry 3.0 vs Industry 4.0 communication topology

The device-to-device communication assumes participating devices are smart enough to talk to one another without the need for translation or advanced data processing, even if their nature is different (e.g. your intelligent door can inform your smart IoT kettle to start boiling water for warm tea, once they get informed about poor weather condition by the Internet weather monitoring service when you're back home after a long day of work). Devices constituting mesh or scatter networks communicate virtually with one another in a similar way people do. Figure 295 briefly presents the data flow idea

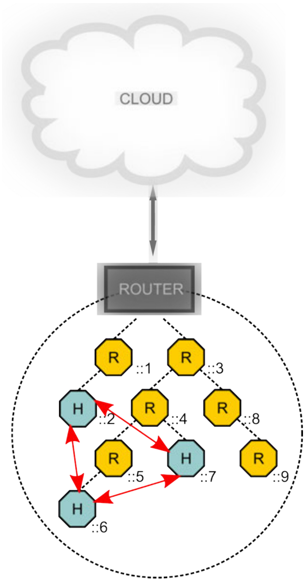


Figure 295: Device to device communication model

Device to Gateway

Device-to-gateway communication occurs when there is a need to provide the translated information between different networks, e.g., some Zigbee [162] network devices need to send data to the Internet to monitor the smart home remotely and manage. This model also appears when there is a need to transfer messages between the IoT network implemented with constrained devices, so using some simplified protocol (e.g. LoRA, 6LowPAN) and the Internet network, using the full implementation of the protocols (e.g. IP6). In this case, the gateway device (sometimes named here as Edge Router) needs to know constrained devices constituting the IoT network, and it usually supplies some missing information instead of them, e.g. enriching message headers or addresses when passing packets from IoT-constrained network to the Internet, but also translating Internet packets (e.g. by removing full address), when acting opposite, e.g. forwarding actuator requests to the IoT devices.

Gatewaying and protocol translation can also occur on the 6th and 7th level of the ISO/OSI model when the implementation of high-level protocols overwhelms even more advanced IoT devices, e.g. simple MQTT texting can be converted to the XML, heavy messages or exposed as XHTML. Those solutions are mostly software-based, e.g. Node-RED [163]. Figure 296 briefly presents the data flow. Please note the protocol change: arrows of the different colours reflect it.

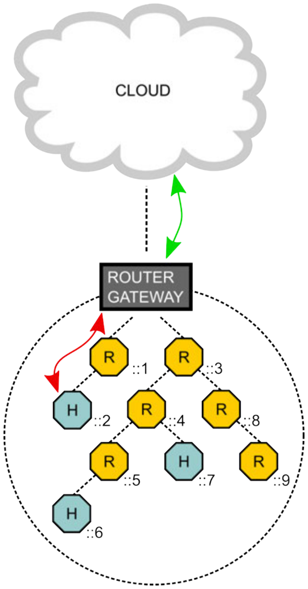


Figure 296: Device to gateway communication model

Device to Cloud

As IoT devices usually cannot constitute an efficient computation structure (as a single IoT node or even their federation), most data is forwarded to the server, often a cloud-based solution, where it is stored and processed. This data processing in the cloud varies, depending on the type of information, their goal, etc. In any case, we usually face the problem of visualisation and data analytics (statistics, AI, data mining, knowledge discovery, big data processing). Those tasks are resource-consuming and require substantial processing capabilities; thus, cloud solutions are usually a good choice.
In this context, “cloud” means public clouds like Amazon, Google or Microsoft and dedicated solutions hidden somewhere in the separated manufacturing networks. Eventually, there is a need to send back some actuation requests to the devices from the cloud. Cloud services are usually PC-based solutions; thus, they extensively use rich protocols, providing their APIs via REST [164], SoAP [165], HTTP GET/POST methods [166], etc. It requires IoT devices interfacing with the cloud to implement full communication stacks. Some IoT devices can interface with cloud services directly. Still, some of them cannot do so due to the constraints, so it is necessary to use gateways, as mentioned in the previous chapter.

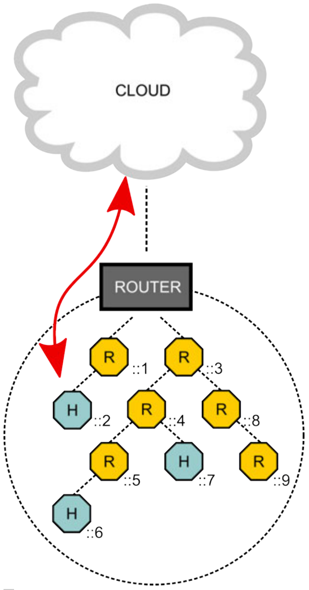


Figure 297: Device to cloud communication model

Recent advances in hardware development and energy-efficient design allowed even the edge-class constrained devices to implement some simple AI models. Still, AI model training, updating and using reinforced learning models typically involves a cloud-based solution.

6.3. Media Layers - Wired Network Protocols

B M E

While the IoT ecosystem is usually considered to be composed of wireless devices, it is still possible to connect IoT solutions using a wired connection.

When wireless-enabled SoCs were about to be delivered to the market (e.g. ESP8266), extension devices were already available for popular embedded systems, like Ethernet Shield for Arduino boards (figure 298).

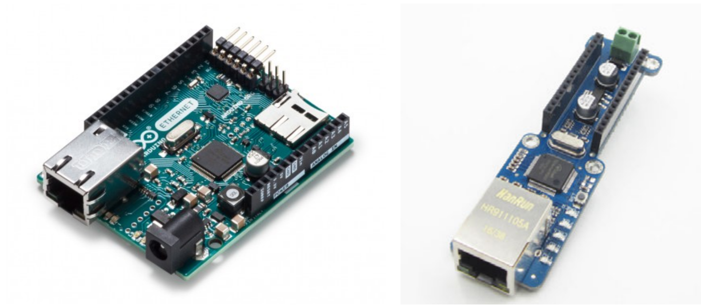


Figure 298: Ethernet shields for Arduino boards

Copper-based wired networks also bring an extra feature to the IoT designers – an ability to power the device via a wired connection, e.g. PoE (Power over Ethernet) – 802.3af, 802.3at, 802.3bt [167]. Long-distance connections may be implemented using optic-based, fibre connections, but those require physical medium converters that are usually quite complex, pretty expensive and power consuming; thus, they apply only to the niche IoT solutions.

The mentioned optical connections do not cover so-called LiFi, as those are considered to be of a wireless nature[168].

A non-exhaustive list of some present and former wired networking solutions is presented in table 35.

Table 35: A Short Review of the Most Popular Wired Networking Standards

Name	Communication medium	Max speed	Topology	Max range (single segment, passive)
Ethernet	Twisted pair: 10BaseT Coaxial: 10Base2/ 10Base5 Fibre: 10BaseF	10 Mbps	Bus, Star, Mixed (Tree)	10Base2: 0.5–200 m (185 m) 10Base5: 500 m 10BaseT: 100 m (150 m) 10BaseF: 2 km (multimode fibre)
Fast Ethernet	Twisted pair: 100BaseTx Fibre: 100BaseFx	100 Mbps	Star	100BaseTx: 100 m (Cat 5) 100BaseFx: 2 km
Gigabit Ethernet	Twisted pair: 1000BaseT	1000BaseT: 1 Gbps 1000BaseX: 4.268 Gbps	Star	1000BaseT: 100 m (Cat 5) 1000BaseLX: 5 km

Name	Communication medium	Max speed	Topology	Max range (single segment, passive)
	Fibre: 1000BaseX (LX/CX/SX)			
Local Talk (Apple)	Twisted pair	0.23 Mbps	Bus, Star (PhoneNet)	1000 ft
Token ring	Twisted pair	16 Mbps	Star wired ring	22.5 m / 100 m (cable dependent)
FDDI	Fibre	100 Mbps (200 Mbps on two rings, but no redundancy)	Dual ring	2 km

The most popular wired networks are 10/100/1000 BaseT – twisted pair with Cat 5, 5e and 6 cables. They require the IoT system to implement a full TCP/IP stack to operate seamlessly with conventional Internet/Intranet/Extranet networks. Because it is usually out of the scope of standard Arduino Uno processor capabilities to implement a full TCP stack, there are typically dedicated processors on the network interfaces that assist the central processor or even handle all networking tasks themselves.

6.4. Media Layers - Wireless Network Protocols

Wireless connections define core communication for IoT devices. A vast and growing amount of protocols, their variations and the dynamic IoT networking market all present a non-solid situation where old “adult” Internet protocols coexist along with new ideas, and IoT hardware and software platforms are more and more capable with every new generation; thus new concepts appear almost daily. Currently, many IoT networking protocols are defined for various layers of the protocol implementation stack, some compatible while others are concurring. Figure 299 presents some selected protocols existing for IoT. This covers only the most popular ones and gives a non-exhaustive view. We discuss them in more detail below.

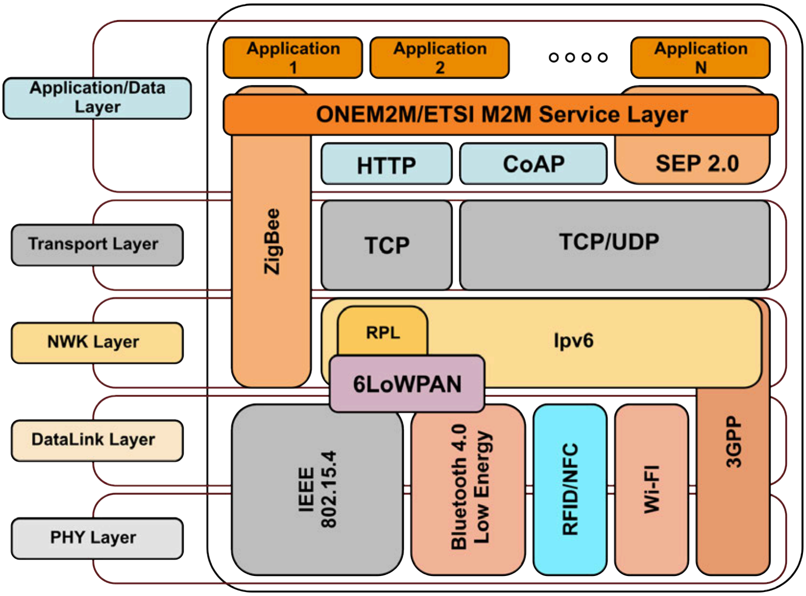


Figure 299: IoT protocols

PHY + MAC + LLC Layers

Below is a list of the most popular wireless protocols for the lower ISO/OSI layers 1–2 (Physical and Media Access Control); some also implement layer 3 – Networking in a single component).

WiFi

WiFi is the set of standards for wireless communication using the 2.4 GHz or 5 GHz band, slightly different spectrum in different countries. The core specification of the 2.4 GHz contains 14 channels with 20 MHz (currently 40 MHz) bandwidth. While there is no centralised physical layer controller, collisions frequently occur even more with a growing number of devices sharing the band. The collision is handled using CSMA-CA with a random binary exponential increase of repeating time.

With the high transmission speed and range usually not exceeding 100 m, it is widely used as the direct replacement of wired Ethernet in local area networks. It is an excellent choice when the amount of data to be transferred is larger, for example, video streams or assembled IoT streams delivered by gateways.
It can also be used in direct connectivity for smart sensors and other IoT elements, but the protocol is not designed to transmit small data packets. It is too energy-consuming for many IoT applications, especially battery-powered devices.
Moreover, WiFi itself offers only 1-to-1 (figure 300 or star-like, 1-to-many (figure 301 topologies of connections, where the central point is a WiFi Access Point. It does not provide mechanisms, e.g. self-reorganised, failure-tolerant mesh networks.

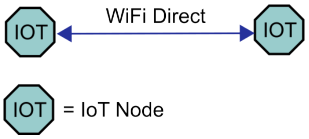


Figure 300: WiFi 1-to-1

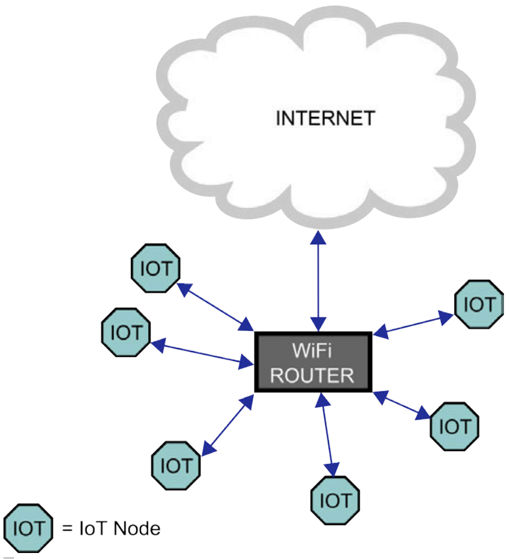


Figure 301: WiFi Star Topology

WiFi has become a more and more popular choice for not-so-constrained IoT devices because they need to implement a full TCP/IP stack, and those devices that are also not so constrained with power resources. A list of WiFi standards and related transmission speeds is present in table 36.

Table 36: WiFi Standards Summary

802.11 standard	Frequency	Channel width	Transmission speed (maximum)
802.11b	2.4 GHz	20 MHz	11 Mbps
802.11a	5 GHz	20 MHz	54 Mbps

802.11 standard	Frequency	Channel width	Transmission speed (maximum)
802.11g	2.4 GHz	20 MHz	54 Mbps
802.11n	2.4 GHz and 5 GHz	20 & 40 MHz	450 Mbps, single-user MIMO
802.11ac	5 GHz	20, 40, 80 MHz (for 802.11ac wave1) 20, 40, 80, 160 MHz (for 802.11ac wave2)	866.7 Mbps, single-user MIMO (for 802.11ac wave1) 1.73 Gbps, multi-user MIMO (for 802.11ac wave2)
802.11ax	2.4 and 5 GHz	20,40, 80, 160 MHz	2.4 Gbps, multi-user MIMO

Bluetooth

Bluetooth is a prevalent method of connecting various devices at short distances. Almost every computer and smartphone has a Bluetooth module built in. Standard has been defined by Bluetooth SIG (Special Interest Group), founded in 1998. Bluetooth operates in the 2.4 GHz band with 79 channels with automatic channel switching when interference occurs (frequency hopping). The single channel offers up to about 1Mbps (where around 700kbps is available for the user) bandwidth, and it provides communication within the range from up to 1 m (class 3, 1 mW) to up to 100 m (class 1, 100 mW). The most prevalent version is class 2, with a 10 m range (2.5 mW).

Every Bluetooth device has a unique 48-bit MAC address.

Bluetooth offers various “profiles” for multimedia, serial ports, packet transmission encapsulation (PAN), etc. The PAN (Personal Area Network) Profile and SPP (Serial Port) are the most useful for IoT devices.

Now Bluetooth covers two branches: BR/EDR (Basic Rate/Enhanced Data Rate) for high-speed audio and file transfer connections and LE (Low Energy) for short burst connections [169].

Classical (before BLE and 4.0) Bluetooth networks can create ad-hoc, so-called WPAN (Wireless Personal Area Networks), sometimes referenced as Piconets. Bluetooth Piconet can handle up to 7 + 1 devices, where 1 device acts as Master and can contact up to 7 Slave devices. Only the Master device can initiate a communication. Fortunately for the IoT approach, much Bluetooth hardware can act as Slave and Master simultaneously, constituting a kind of router; thus, devices can include a tree-like structure named a scatternet as presented in figure 302.

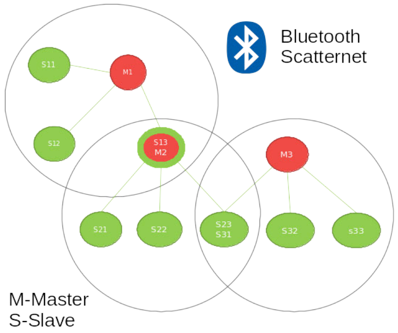


Figure 302: Bluetooth Scatternet

Bluetooth Low Energy (BLE) uses a simplified state machine implementation and thus is more constrained-devices friendly. It offers a limited range and is designed to expose the state rather than transmit streamed data. However, it provides a speed reaching up to about 1.4 Mbps (2 Mbps aerial throughput) if needed. It uses a 2.4 GHz band but is designed to avoid interference with WiFi AP and clients. Communication is organised into three advertising channels (located "between" WiFi) and 37 communication channels.

Latest Bluetooth implementations (protocol version 5.0 and newer, implemented in mid-2017) offer a Bluetooth mesh network extending ubiquitous connectivity via a many-to-many communication model dedicated to IoT devices, lighting, Industry 4.0, etc. The Bluetooth mesh is layer-organised, and since there is no longer a Master-Slave model used, but messages are relayed through the mesh, it is considered to be no longer the Scatternet because of its flat structure [170]. Sample Bluetooth Mesh Network idea is presented in figure 303 and a review of the Bluetooth protocols in table 37.

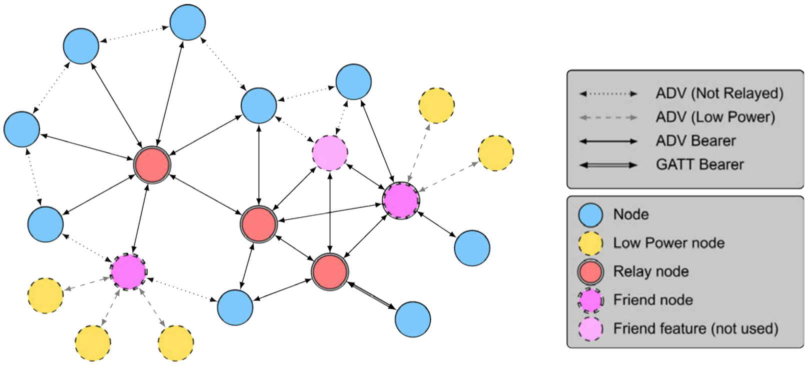


Figure 303: Example Topology of the Bluetooth 5 Mesh Network

Improvements introduced in the 5.1, 5.2, and 5.3 versions focused on localising neighbouring nodes better, on audio use, and improving power efficiency. Version 5.4 introduces a new feature that allows nodes to send encrypted data in the advertising frames (Encrypted Advertising Data). Another improvement is also focused on advertising frames, making it possible to respond to such a frame (Periodic Advertising with Responses). Both extensions are beneficial in the IoT world, allowing nodes to send small, encrypted packets using an advertising mechanism. Bluetooth 5.4 enables connectionless, bidirectional, secure communication with many low-power end nodes in the star topology.

Table 37: Bluetooth Standard Summary

Bluetooth	Transmission speed	Remarks
1.0	21 kbps	Few implementations
1.1	124 kbps	
1.2	328 kbps	First popular version
2.0 + EDR	3 Mbps	Extended Data Rate
3.0 + HS	24 Mbps	High Speed
3.1 + HS	40 Mbps	
4.0 + LE	1 Mbps	Low Energy
4.1	Designed for IoT	

Bluetooth	Transmission speed	Remarks
5.0	50 Mbps	One standard for all purposes
5.1		Better accuracy of node localization
5.2		Defined for Audio
5.3		Improved power efficiency
5.4		Improved security

Cellular

Cellular (mobile/GSM) networks are viable options for IoT communication because of their omnipresence and long-range communication capabilities. Those networks use orthogonality in frequency and time spaces. Cellular networks are presented by the subsequent generations (G) – currently up to 5G on the market and 6G in the experimental phase (introduced in years 2025-2029, country-dependent). Typical GSM network technology, sometimes referenced as an era, runs out within about 10–15 years. It is pretty close but still less than expected end-of-life for classes of IoT devices (15-25 years). GSM hardware was backwards compatible, enabling users to access older, even before 2G GSM networks with the latest chips. Still, the presence of old-generation networks becomes sparse, particularly in metropolitan areas, where recent generations provide better coverage and capacity.

Figure 304 presents GSM network evolution over time and generations. Cellular networks use different frequencies in different countries, yet available radio implementations nowadays can usually handle all of them.

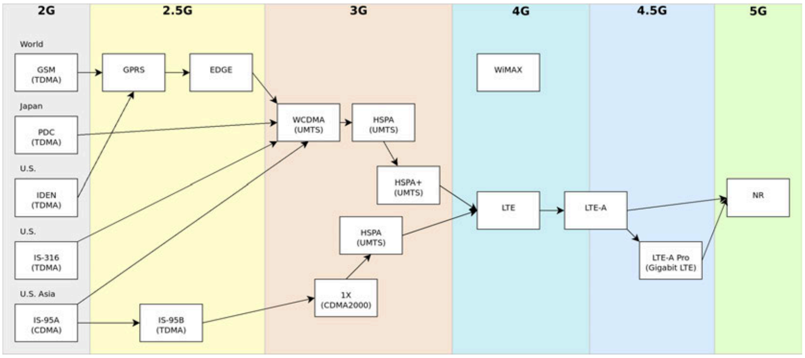


Figure 304: GSM network evolution and generations

Figure 305 presents sample GSM hardware (separate module and ready shield for the Arduino platform).

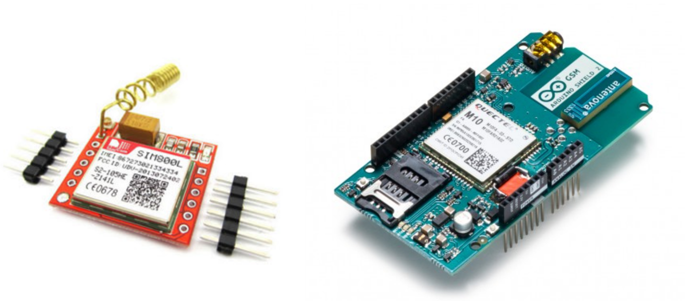


Figure 305: Sample GSM hardware for IoT prototyping

GSM protocols are proprietary, complex (including advanced ciphering) and require dedicated hardware. Documentation and standards are not publicly available because of security considerations (e.g., voice transmission ciphering details).
On the one hand, the GSM network seems to be a good solution for extended distant IoT networks. They have many disadvantages, however: they require operators' infrastructure, as GSM bands are not free, and GSM signalling requires quite decent energy.

Professional operation requires licencing, and connecting existing infrastructure involves a purchase of a unique identifier (phone ID and a number given by the SIM card, physical or virtual) and a service fee. With the 5G network, GSM offers dedicated IoT services such as network slicing and better energy efficiency.

Besides limited access constraints, one more particularly important exists: GSM modems use quite a significant amount of energy when establishing a connection because they need to broadcast their existence as far as possible to gain a link with a possibly distant-located base station. It requires tremendous power and drains the battery (up to 10 W peak); thus, cellular solutions are unsuitable for IoT devices that use frequent data communication and are constrained on energy resources.

ZigBee

ZigBee protocol is prevalent in Smart House but also in Industry appliances. Zigbee is a wireless technology developed as an open standard to address the needs of low-cost, low-power wireless machine-to-machine networks. However, it is more popular in the industry because of the relatively higher equipment cost than WiFi, Bluetooth or other RF modules.
The Zigbee standard operates on the radio bands 2.4 GHz for smart home applications, 915 MHz in the US and Australia, 868 MHz in Europe and 784 MHz in China. The advantage of ZigBee is the possibility of forming mesh networks where nodes are interconnected with others, so there are multiple paths connecting each pair of nodes. Connections are dynamically updated, so when one node turns off, the path going through that node will be automatically rerouted via another route.
Transmission speed is up to 250 kbps, with a theoretical range of up to 100 m but usually

to some 10–30 m.
ZigBee does not provide direct, unique IP-addressing on the Networking layer like 6LowPAN or Thread do. A single ZigBee network can handle up to 65000 devices.

Z-Wave

Z-Wave is a protocol similar in principle to the ZigBee, but hardware is cheaper; thus, it is more towards inexpensive home automation systems. Like in ZigBee, Z-Wave operates on different frequencies depending on the world region, usually between 865 MHz and 926 MHz. The transmission speed is up to 200 kbps, and the range is 100m. A single Z-Wave network is pretty limited in the number of concurrent devices in one network, that is, only 232 devices. Each Z-Wave network has a unique ID, and each node (device) in a network has a unique 8-bit identifier.

Thread

Another standard [171] works using the same 802.15.4 radio and is based on IPv6. There are some differences in the protocol, like address allocation. Like the Z-Wave mentioned above and Zigbee, Thread uses mesh network topology. It incorporates encryption, authentication, and secure key management to protect communication between devices on the network. It is also energy efficient, allowing devices constituting a mesh network to fall asleep and awake when only needed for communication. Those mechanisms cover (among others) asynchronous communication, scheduled sleep, routing concerning the devices' energy resources, adaptive data rates, wake-on-radio, and paging mechanisms (waking up only a selected group of devices).

NFC

NFC (Near Field Communication) is a technology that enables two-way interactions between electronic devices. One device mustn't have to be equipped with the power source – the receiving radio signal powers it. That's why NFC is used in contactless card technology, enabling devices to exchange data at a distance of less than 4 cm. Transmission speed varies between 100–420 kbps, the range between active devices is up to 10 cm, and the operating frequency is 13.56 MHz.

Sigfox

Sigfox [172] is the idea to connect objects with sub 1 GHz radio frequency. It uses the 900 MHz frequency range from the ISM band. The range is about 30–50 km (open space) and 3–10 km (urban environments). This standard uses a technology called Ultra Narrow Band (UNB). It has been designed to transmit data with deficient speed – from 10 to 1000 bps. Thanks to small data packets, it consumes only 50 mW of power. It is intended to create public networks only, so using Sigfox requires a subscription plan. The Sigfox network covers many (but not all) European countries.

LoRa and LoRaWAN

LoRa (Long Range) is the technology for data transmission with a relatively low speed (20 bps do 41 kbps) and a range of about 2 km (new transceivers can transmit data up to 15 km). It uses CSS (Chirp Spread Spectrum) modulation in the 433 MHz or 868 ISM radio band.

A chirp signal is characterized by a continuous frequency sweep over time. This means that the frequency of the transmitted signal starts at some lower frequency and continuously increases throughout the transmission of a single symbol. In LoRa the starting frequency differs depending on the symbol encoded, and while the modulated signal achieves the maximal value of the frequency starts from the minimal one. It means that each chirp uses the whole available bandwidth. Chirp Spread Spectrum modulation makes LoRa signals less susceptible to interference and noise and allows LoRa to achieve long-range communication. LoRa modulation is characterized by two parameters:

- **Spreading Factor** determines the speed of the signal frequency change over time. Higher spreading factors result in a longer communication range but lower data rates. It also defines the number of bits encoded by one chirp.
- The **Bandwidth** of the LoRa signal determines the amount of spectrum occupied by the transmitted signal. It can be 125, 250 or 500 kHz. It also specifies the sampling frequency of the signal in the receiver.

Having these parameters it is possible to calculate the efficient data rate (in bps). Because the range of LoRa communication is long, transmitters can interfere, so some rules for the maximum time of occupation of the channel were introduced. In the European Union, the maximum percentage of transmission time known as the Duty Cycle is 1%. This gives a maximum transmission time of 864 seconds per day. Transmission should be as short as possible, and the delay between following transmissions should last a few minutes. The duty cycle together with bandwidth and spreading factor makes it possible to calculate the maximum payload of the frame and the bitrate. Some online calculators help set LoRa parameters to fulfil the local regulations [173].

The cell topology is the star, with the gateway at the central point. End devices use one-hop communication with the gateway. A LoRaWAN gateway is usually connected to the standard IP network with a central network server. The LoRa technology is supported as LoRaWAN by LoRa Alliance [174] designed as Sigfox for public networks. Still, it can also be used in private networks that do not require a subscription. LoRaWAN uses simplified messaging, where collisions are solved at the server level.
The major assumption for the LoRaWAN network is each end-node device is within a range of at least one LoRaWAN gateway.
There are 3 classes of devices in LoRa:

- Class A: where downlink is active only after the device uses uplink in a particular time window (twice). It has the greatest energy efficiency among other classes. Downlink opportunity appears asynchronously, so this class is for scenarios where low latency is not a critical requirement.
- Class B: with scheduled receive window, where the downlink is synchronised; thus, the LoRa device listens to the downlink periodically. This causes increased energy use, however.
- Class C: is a class where the device listens to the downlink communication almost continuously. This brings the lowest latency in communication and the highest energy demand compared to the other classes.

NET (NWY) Layer

Traditionally, we use IP addressing (usually masked by DNS to be more user-friendly) when accessing Internet resources. IoT devices may also benefit from this approach. However, constrained devices require special “editions” of the conventional protocols, which are lightweight. The networking layer implements the basic communication mechanisms on the packet level, like routing, delivery, proxying, etc. Many IoT, lightweight implementations of the protocols presented below benefit or at least inherit ideas from regular “adult” implementations. Please note that some protocols implement more than one layer, as illustrated in image 299. We also provide a short reference of the IPv4 and IPv6 to show advantages and drawbacks.

IPv4

Internet Protocol v4 (1981) is perhaps the most widespread networking protocol. The predecessor of the IPv4 protocol, originally called IP, was introduced in 1974 and supported up to 2^8 hosts, organised in 2^4 subnetworks (RFC 675).

In IPv4 (RFC 760/RFC791), the logical addressing space was extended to 2^{32} devices, which seemed to be quite much in 1981, but now we struggle with a lack of free addressing space. This number is less because some addresses are reserved, e.g. for broadcasting and due to the existence of different classes of addresses and their pools [175]. Sample IPv4 address is, for example, 157.158.56.1.

Some relief to the suffocating Internet was brought as an ad-hoc solution with the NAT (Network Address Translation) introduction. NAT-enabled subnetworks are those where one public address represents a set of devices hidden behind the router. However, that limits usability because of the lack of direct access and unique identification in the global network level of the devices sharing private address spaces. Even so, more than 29 billion IoT devices are expected to be connected to the Internet by the end of 2030, according to Statista forecast [176]. They all need to be uniquely addressed!

IPv6

IPv6 is the next generation of the IPv4 protocol. It is supposed to replace IPv4. The transition process is not as quick as expected because many Internet and intranet services implement IPv4 only and would become inoperable if IPv4 were unavailable.
IPv6 brings addressing space large enough to cover all existing and future needs as it is possible to forecast. The number of possible addresses is 2^{128}. Addresses are presented by 8 groups of 4 hexadecimal values, e.g. 2001:0db8:0000:0042:0000:8a2e:0370:7334.

This brings the capability to uniquely identify any device connected to the Internet using its IPv6 address. Regarding IoT, implementations have many drawbacks (IPv4 also has them). IPv6 network is star-like, whereas IoT networks can benefit from the mesh model. IPv6 network requires a controller providing free addresses (a DHCP server) – devices must contact it to obtain the address. Every IoT device needs to keep a list of devices it corresponds with (ARP) to resolve their physical address. Moreover, full IPv6 stack implementation requires large RAM when used.

6LoWPAN

The name is the abbreviation of “IPv6 over Low-Power Wireless Personal Area Networks” [177] and, as it says, it is the IP-based network.
This protocol was introduced as a lightweight version of full IPv6, IoT-oriented.
This feature allows connecting 6LoWPAN networks with other networks using a so-called Edge Router. Thus, every node can be visible on the Internet and uniquely addressable, as stated in the IoT principles. This standard has been developed to operate on the radio channel defined in 802.15.4 (as ZigBee, Z-Wave). It creates the adaptation layer that allows the use of IPv6 over the 802.15.4 link. 6LoWPAN has been adopted in Bluetooth

Smart 4.2 standard as well.

6LoWPAN supports two addressing models: 64-bit and 16-bit. The former limits the number of devices connected to one network to 64000 nodes. The primary frame size is just 127 bytes (compared to full IPv6, where it is 1280 bytes at least). 6LoWPAN supports unicast and broadcast. It also supports IP routing and link-layer mesh (802.15.5) that introduces the fail-safe redundant, self-organising networks because the link-layer mesh can have more than one Edge Router. 6LoWPAN uses autoconfiguration for neighbour device discovery, so it does not require a DHCP server. It also supports ciphered transportation using AES 128 (and AES 64 for constrained devices).

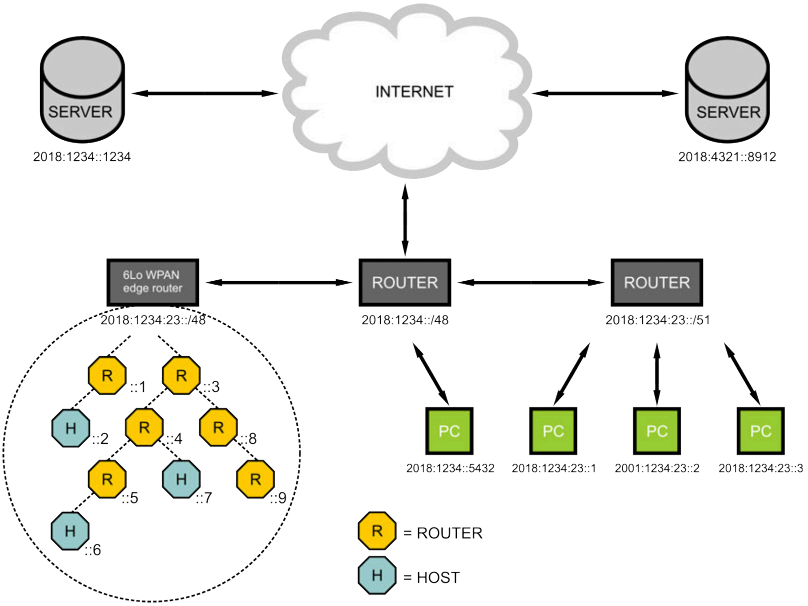


Figure 306: Sample 6LoWPAN and Internet Integration

6LoWPAN devices can be just nodes (Hosts) or nodes with routing capability (Routers) as presented in figure 306.

The Edge Router implements a gateway between 6LoWPAN and the regular IPv6 (IPv4) network. It aims to translate “compressed” IPv6 addresses to ensure bi-directional communication between the Internet and 6LoWPAN nodes. Note – the network structure of the 6LoWPAN is logically flat (star/mesh with single addressing space), and devices have unique MAC addresses to be recognisable by the Edge Router device.

6.5. Transport Layers

The transport layer in the standard Internet ISO/OSI stack is between the Media and Application layers. Sometimes, it is a part of the so-called host layer protocols, composed of the top 4 layers: transport, session, presentation, and application.

In IoT protocols, it is commonly a part of the other layers, commonly networking. Besides pure IoT protocols, this book also contains regular internet ones (such as WiFi), so we briefly introduce the transport layer focused on the IP.

The transport layer is responsible for end-to-end communication and provides segmentation and aggregation mechanisms, error detection and correction, flow control, port-based addressing and quality of services. IP-based protocols use two kinds of connections:

- **TCP**: connection-oriented mode, where the connection between endpoints is kept open for the whole transmission time, and it is possible to detect connection breaks and interferences. TCP ensures delivery, and this mechanism is built into the Transportation layer.
  One can imagine a TCP connection as making a phone call: while connected, you can hear the other side's speech, breathing, and background noises to tell the connection is not dropped. A dropped phone connection can be sensed, e.g., by the termination signal or simply when there is silence and no replies from the other party. Similarly, in TCP, when there is no acknowledgement for a specific timeout or after some retries, the connection is considered dropped. This way, both endpoints will be aware of the state of the connection and can ensure the quality of service.\\Examples of services using this connection on the regular Internet are Web (HTTP), FTP, and SSH.
  TCP is more complex than UDP in terms of both transmission and implementation.
- **UDP**: connectionless mode, where the sender sends the data and “hopes” to let it be delivered to the receiver. There is no warranty for the message to have been delivered. Moreover, because of the routed nature of the connections, consecutively sent messages may be delivered partially, unordered, duplicated or not delivered at all. For this reason, error checking and correction and quality of service in UDP are implemented in higher layers (if any).
  UDP is lightweight and, therefore, is typical for video streaming, e.g. video conferencing, in Internet applications. It is suitable when it is acceptable to lose some messages, e.g. in the multimedia transmission.
  One can imagine UDP protocol as sending an SMS message: you simply “send and forget”, and your friend may reply to it, to acknowledge, or may not. Delivery time is not precisely known and can only be estimated.
  UDP's implementation and use require fewer resources than TCP, so it is a better choice for constrained IoT devices and common in the networks connecting end node (edge class) devices.

A choice between UDP and TCP is driven by other network layers, e.g. some IoT network stacks do not provide TCP-like connections. Higher-level protocols also drive the selection of the transport layer connection standards, e.g. MQTT generally requires TCP while CoAP uses UDP; see the following chapters for details.

Popular IoT protocols may have a transportation layer embedded and merged with other layers, e.g. Zigbee protocol. In contrast, others follow regular ISO/OSI stack division into the 7 layers, as presented in figure 307.

Most IoT protocols use UDP, as it is easier to implement and more energy efficient regarding communication energy consumption, which is the primary factor to consider, e.g., in sensor networks.

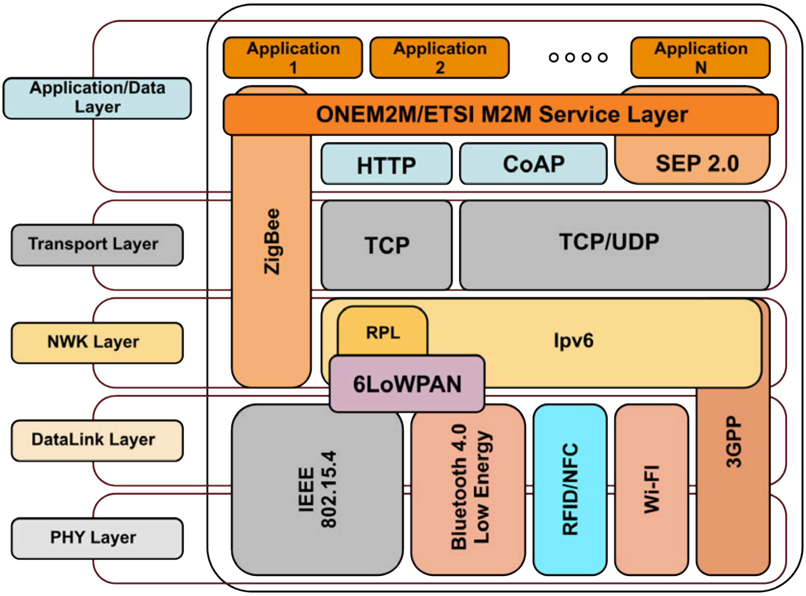


Figure 307: IoT Communication Protocols

6.6. Application Protocols

The host layers protocols include session (SES), presentation (PRES) and application (APP) levels. In particular, the APP (application) layer in regular Internet communication is dominated by the HTTP protocol and XML-related derivatives, e.g. SoAP. Also, the FTP protocol for file transfer is ubiquitous; it has existed since the beginning of the Internet. Most of them are somehow related to the textual presentation of the information. They're referenced as “WEB” protocols.
Although advanced and more powerful IoT devices frequently use these protocols, this is problematic to implement in the constrained IoT devices world: even the simplest HTTP header occupies at least 24 + 8 + 8 + 31 bytes without payload!
There is also a problem of crossing firewall boundaries when communication between subnetworks of IoT devices is expected to occur.

Some IoT-designed protocols are reviewed below.

MQTT

MQTT protocol [178] was invented especially for constrained IoT devices and low bandwidth networks. It is located in APP layer 7 of the ISO/OSI stack but covers all layers 5–7. It is a text-based protocol yet very compact and efficient. Protocol stack implementation requires about 10 kB of RAM/flash only.

MQTT uses a TCP connection, so it requires an open connection channel (this is opposite to UDP connections, where communications work in a way: “send and forget”). It is considered a drawback of the original MQTT protocol, but MQTT variations exist for non-TCP networks, e.g. MQTT-SN. Protocol definition provides reliability and delivery ensure mechanisms.

The standard MQTT Message header is composed of just two bytes only (table 38)! There are 16 MQTT message types. Some message types require variable-length headers.

Table 38: MQTT Standard Message Header

bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP flag	Qos level		RETAIN
byte 2	Remaining length							

MQTT requires a centralised MQTT Broker located outside firewalls and NATs, where all clients connect, send and receive messages via the **publish/subscribe** model. The client can act as publisher and subscriber simultaneously. Figure 308 presents a publish-subscribe model idea.

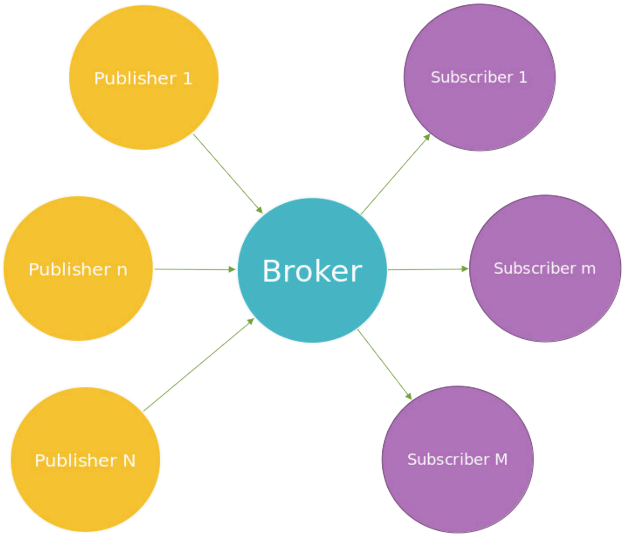


Figure 308: MQTT Broker, Publishers and Subscribers

MQTT Message

MQTT is a text-based protocol and is data-agnostic. A message comprises a Topic (text) and a Payload (data). The Topic is a directory-like string with a slash ("/“) delimiter. Thus, all Topics constitute (or may represent) a tree-like folder structure similar to the file system's. The subscriber can subscribe to a specific, single Topic or a variety of Topics using wildcards, where:

- # stands for the entire branch,
- + stands for the single level.

Example Scenario

Publishers deliver some air quality information data in separate MQTT messages and for various rooms at the department Inf of the Universities (SUT, RTU, ITMO) to the Broker:

Topic (publishers):
SUT/Inf/Room1/Sensor/Temperature
SUT/Inf/Corridor/Sensor/Temperature
SUT/Inf/Auditorium1/Sensor/Temperature
RTU/Inf/Room1/Sensor/Temperature
ITMO/Inf/Room1/Sensor/Temperature
RTU/Inf/Room1/Sensor/Humidity
SUT/Inf/Room3/Sensor/Temperature
RTU/Inf/Room1/Window/NorthSide/State

Subscriber 1 is willing to get all sensor data for SUT University, Inf (informatics) department only, for any space:

Topic (subscription):
SUT/Inf/+/Sensor/#

Subscriber 2 is willing to get only Temperature data virtually from any sensor and in any location in TalTech University:

Topic (subscription):
TalTech/#/Temperature

Subscriber 3 is willing to get any information from the sensors, but only for the RTU

Topic (subscription):
RTU/#

The message's payload (data) is also text, so if binary data is to be sent, it must be encoded (e.g., using Base64 encoding).

MQTT Broker
MQTT Broker is a server for both publishers and subscribers. The connection is initiated from the client to the Broker, so assuming the Broker is located outside a firewall, it breaks its boundaries.
The Broker provides QoS (Quality of Service) and can retain message payload. MQTT Broker QoS (supplied at the message level) has three levels.

- Unacknowledged service: Ensures that MQTT message is delivered at most once to each subscriber.
- Acknowledged service: Ensures message delivery to every subscriber at least once. The Broker expects acknowledgement to be sent from the subscriber. Otherwise, it retransmits data.
- Assured service: This two-step message delivery ensures the transmission is delivered exactly once to every subscriber.

Providing unique packet IDs in the MQTT frame is vital for Acknowledged and Assured services.

The DUP flag (byte 1, bit 3) represents information sent by the publisher, indicating whether the message is a “first try” (0) or a retransmitted one (1). This flag is primarily for internal purposes and is never propagated to the subscribers.

MQTT offers a limited set of features (options):

- clean session flag for durable connections:
  - if set *TRUE*, the Broker removes all of the client subscriptions on client disconnect,
  - otherwise, the Broker collects messages (QoS depending) and delivers them on client reconnecting; thus, connections remain idle.

- MQTT “will” - on connection loss, the Broker will automatically “simulate” publishing of the predefined MQTT message (Topic and payload). All clients subscribing to this message (whether directly or via a wildcard) will be notified immediately. It is an excellent feature for failure/disaster discovery.

- message retaining: it is a feature for regular messages. Any message can be set as retaining; in such case, the Broker will keep the last one. Once a new client subscribes to a topic, they will receive a retained message immediately, even if the publisher is not publishing any message. This feature is **last known good value**. It is good to present the publisher's state, e.g. the publisher sends a retained message meaning “I'm going offline” and then disconnects. Any client connecting will be notified immediately about the device (client) state.

Interestingly, MQTT is a protocol used by Facebook Messenger [179]. It is also implemented natively in Microsoft Azure and Amazon Web Services (among many others).

MQTT security is relatively weak. The MQTT Broker can offer user and password verification sent in plain text. However, all communication between the client and Broker may be encapsulated in an SSL-encrypted stream.

A short comparison of MQTT and HTTP protocols is presented in table 39.

Table 39: MQTT vs HTTP

	MQTT	HTTP
Design	Data-centric	Document centric
Pattern	Publish/Subscribe	Request/response
Complexity	Simple	Complex
Message size	Small, with 2 byte binary header	Larger with text-based status
Service levels	3 QoS	None
Implementation	C/C++: 10–30 kB Java ~100 kB	Depends on application but hits > MB
Data distribution models	1-to-1 1-to-N	1-to-1

CoAP

CoAP protocol (RFC7252) originates from the REST (Representational State Transfer). CoAP does not use a centralised server as MQTT does, but every single device “hosts” a server on its own to provide available resources to the clients asking for service offering distributed resources. CoAP uses UDP (compared to MQTT, which uses TCP) and is stateless; thus, it does not require memory to track the state. The CoAP implementation assumes every IoT device has a unique ID, and things can have multiple representations. It is intended to link “things” together using existing standard methods. It is resource-oriented (not document-oriented like HTTP/HTML) and designed for slow IoT networks with high packet loss. It also supports devices that are periodically offline.
CoAP uses URIs to address services:

- coap://<host>[:<port>]/<path>[?<query>] to access a service/resource,
- a secure, encrypted version uses “coaps” instead of “coap”.

It supports various content types, can work with proxy and can be cached.
The protocol is designed to be compact and straightforward to implement. The stack implementation takes only about 10 kB of RAM and 100 kB of storage. The header is only 4 bytes.

CoAP protocol has a binary header to decrease overhead, but the payload depends on the content type. The initial, non-exclusive list of the payload types includes:

- text/plain (charset=utf-8) (ID=0, RFC2046, RFC3676, RFC5147),
- application/link-format (ID=40, RFC6690),
- application/xml (ID=41 RFC3023),
- application/octet-stream (ID=42, RFC2045, RFC2046),
- application/json (ID=50, RFC7159).

CoAP endpoint services are identified by unique IP and port number. However, they operate on the UDP instead of TCP (like, e.g. HTML does). The transfer in CoAP is made using a non-reliable UDP network, so a message can appear duplicated, disappear, or be delivered in another order than initially sent. Because of the nature of datagram communication, messages are exchanged asynchronously between two endpoints, transporting Requests and Responses. CoAP messages can be (non-exhaustive list):

- CON - Confirmable, those requiring ACK Acknowledge,
- NON - Non-Confirmable, those that do not need ACK,
- ACK - an acknowledgement message,
- RESET - sent if CON or NON was received, but the receiver cannot understand the context, e.g. part of the communication is missing because of device restart, messages memory loss, etc.

Empty RESET messages can be used to “ping” the device.

Because of the UDP network characteristics, CoAP provides an efficient yet straightforward reliability mechanism to ensure successful delivery of messages:

- stop and wait for retransmission with exponential back-off for CON messages,
- duplicate message detection for CON and NON-messages.

The request-response pair is identified by a unique “Token”. Sample request-response scenarios are presented in the images below. Sample CoAP message exchange scenarios between client and server are presented (two per image) in figure 309 and figure 310.

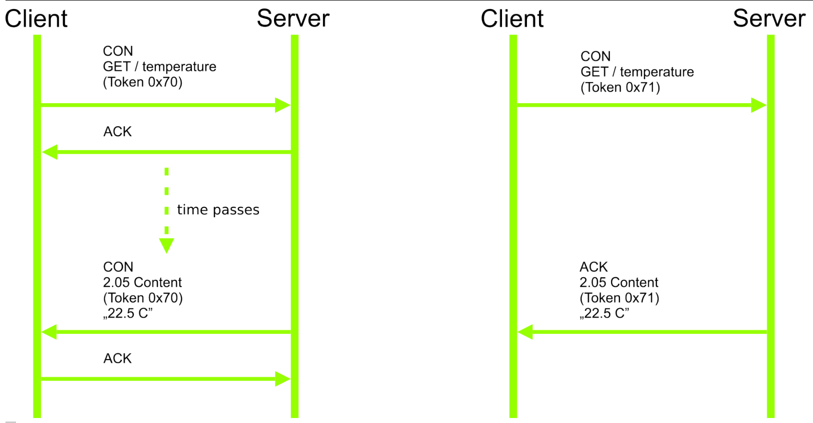


Figure 309: CoAP scenario 1: confirmable with time delay payload answer (0 × 70) and immediate payload answer (0 × 71)

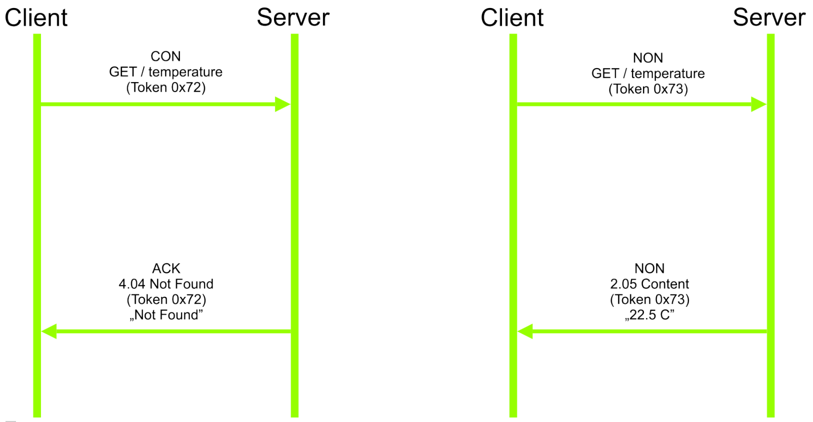


Figure 310: CoAP scenario 2: unrecognized request (0 × 72) and non-confirmable request (0 × 73)

The scenario in figure 309 (left, with token 0 × 70) is executed in a situation when a CoAP server device (a node) needs some time to prepare data and cannot deliver information right away. The scenario in figure 309 (right, with token 0 × 71) is used when a CoAP server can provide information to the client immediately. The scenario in figure 310 (left, with token 0 × 72) appears when a CoAP server cannot understand the request. The scenario in figure 310 (right, with token 0 × 73) presents the situation where the request to the CoAP server was made with a non-confirmable request.

AMQP

In its principles, the AMQP (Advanced Message Queuing Protocol) somehow recalls MQTT: it is message-oriented and uses a central broker. There are data publishers and consumers (that, in the case of the MQTT, are called subscribers). Messages are routed from publishers to the Broker, where they hit so-called exchanges, and then they are copied to the queues (0, 1 or more) from which the consumer can later read. A diagram of the message's flow is present in the figure 311.

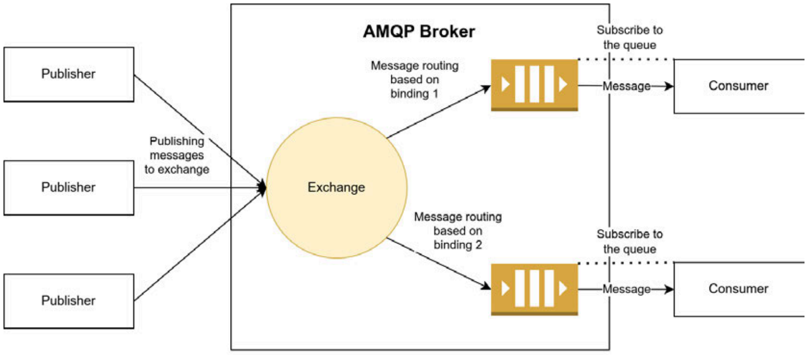


Figure 311: AMQP protocol messages flow

AMQP uses TCP/IP. AMQP is intended to work in non-reliable networks; thus, the protocol has a message acknowledgement mechanism to ensure delivery. A message is removed from the queue only if it has been acknowledged. Besides acknowledged delivery, it is also possible to use an unacknowledged one that does not involve acknowledgements. If a message cannot be routed (for any reason), it can be returned to the publisher, dropped or placed in the “dead letter queue”. The behaviour is defined along with a message. Opposite to MQTT, in AMQP protocol, the connection status is unknown; thus, there is no mechanism to let other devices know that some node has disconnected, such as the last will in MQTT.

Queues
AMQP is a programmable protocol, so bindings are not defined by the Broker but rather by the publisher. Queues are also created on-demand via external actors (mostly consumers). Routing via bindings is provided with a message, and the Broker analyses it to provide correct message handling and delivery.
Consumers can subscribe to the exchange and define a queue. Bindings then act as filters so they receive only selected messages. A single queue is intended to handle one consumer, but there is a possibility of letting many consumers use a single queue in the round-robin model. As in the protocol version 0.9, queues have the following properties:

- Name,
- Durable flag - the queue and its contents are persistent across broker restarts,
- Exclusive flag - used exclusively by one consumer only,
- Auto-Delete flag - the queue is removed if the last consumer unsubscribed,
- Arguments - optional.

A queue name can be selected explicitly, or a broker may deliver one on demand. A queue has to be expressly defined. An existing queue can be silently redeclared with an exact attributes set. A declaration of the existing queue with different attributes set throws an exception code `406 (PRECONDITION_FAILED)`.

Exchanges
Specification 0.9 of the AMQP protocol creates 4 exchanges (exchange types):

- Direct Exchange (its name is empty string or `amq.direct`); The default, Direct Exchange, has a special feature that automatically creates and binds new queues, where the queue's name is the same as a routing key; thus, it is ideal for unicast communication. Assuming the queue's name is K (there can be more than one) and there comes a message with routing key N, the message is routed only to those queues, where K=N (figure 312).

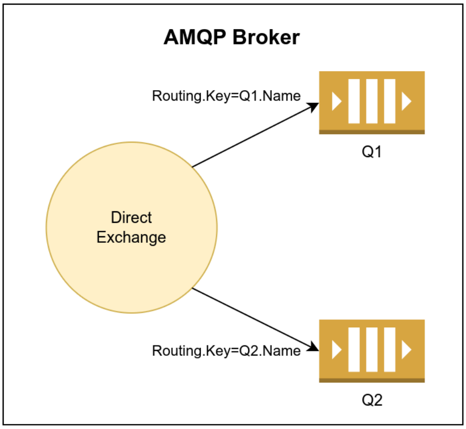


Figure 312: Direct Exchange working principles

- Fanout Exchange (`amq.fanout`); In the Fanout Exchange, all messages are routed to all queues bound to the Fanout Exchange, regardless of their routing key. They help broadcast the information (figure 313).

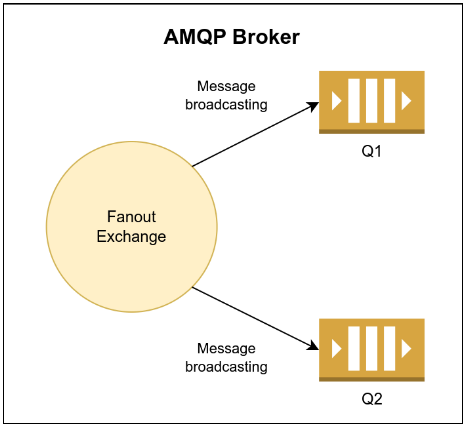


Figure 313: Fanout Exchange working principles

- Topic Exchange (`amq.topic`); Topic Exchange works similarly to MQTT topic subscriptions: an AMQP queue bound to the Topic Exchange defines a pattern rather than the fixed name, and messages with the matching routing key are copied to the queue (314). It is great for multicasting.

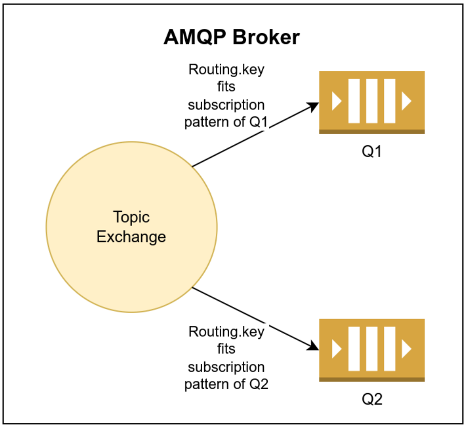

AMQP Broker
Routing.key fits subscription pattern of Q1
Q1
Topic Exchange
Routing.key fits subscription pattern of Q2
Q2

Figure 314: Topic Exchange working principles

- Headers Exchange (`amq.match`); This exchange uses message headers for routing instead of routing keys. The routing key is ignored, and it is possible to bind a queue to the Headers Exchange using more than one header. It is also possible to specify whether it is enough to find a single matching among many conditions or all must be satisfied. Sample routing is present in the figure 315: Q1 requires one of the conditions to be satisfied, while Q2 requires all of the conditions to be satisfied to execute a binding and route a message.

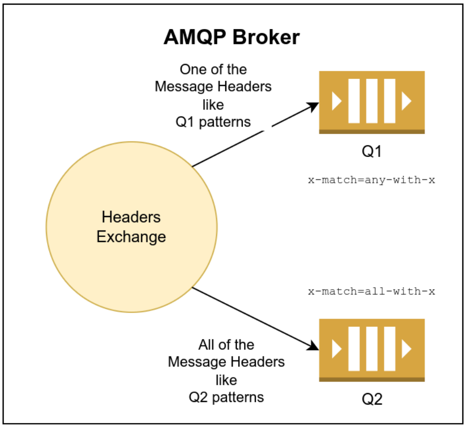

AMQP Broker
One of the Message Headers like Q1 patterns
Q1
x-match=any-with-x
Headers Exchange
x-match=all-with-x
All of the Message Headers like Q2 patterns
Q2

Figure 315: Headers Exchange working principles

Bindings
Bindings are rules defining how to route from exchange to queues. Depending on the exchange type the publisher interacts with, the messages are routed using algorithms that consider both exchange type, arguments and bindings. The publisher is responsible for providing information on which queue will receive the messages from the exchange.

Consumers
Consumers subscribe to the queue to “consume” messages. There are two ways to let consumers receive them: push API and pull API. For performance reasons, pull API should be avoided. Once subscribed to the queue, the consumer receives a unique “consumer tag” that is a string (text) and is later necessary to unsubscribe.

Messages
The essential, immutable AMQP frame size is 8 bytes, and the payload is up to 2GB. Besides the header frame, messages have several virtually freely definable attributes, but the Broker uses some predefined ones. Common attributes cover:

- content type,
- content encoding,
- routing key,
- persistence flag,
- message priority,
- message publishing timestamp,
- content expiration period,
- publisher app id.

Header attributes (used, e.g. by the Headers Exchanges) are optional and similar to X-Headers in HTTP [180].
A payload in the AMQP message is a byte array. Broker does not process or review the content, and it can be even zero length.

Exchanges, bindings and queues are named AMQP entities. To draw an analogy to the real world:

- a queue is like your home,
- an exchange is a nearby train station,
- bindings are routes from the train station to your home, and usually there are one, many, or eventually there are none.

The address of the Broker is referenced with URI, similar to the CoAP (e.g.):

- amqp://<user>:<password>@<host>[:<port>]/<path> for raw connections,

- “amqps” for TLS/SSL secure connections.

Starting AMQP version 1.0, broker control is no longer in the specification; thus, it is expected to be defined in the higher-level protocols. Thus, the AMQP protocol definition seems inconsistent in the development over time.

7. Programming for IoT Networking

The Internet of Things has revolutionised how we interact with the physical world mainly because of the ease of data exchange, which we can make almost everywhere. IoT relies on exchanging information between a myriad of devices and sensors, all of which need to communicate with each other and often with the cloud or other servers. The most exciting Internet of Things features are the possibility of using data transmission between nodes, between nodes and servers, and the ability to read measurements and control the behaviour of devices remotely. All these features use networking functionality. IoT networking is the backbone of the IoT ecosystem, enabling devices to collect and transmit data, receive instructions, and interact with other devices, even from other parts of the world.

Network programming is specific to IoT devices. It uses hardware built into the microcontroller or as an external communication coprocessor connected to the main MCU using one of the popular embedded systems protocols. Many IoT MCUs include a variety of programmable communication radios. Wired communication is possible with external modules (if at all). On the low ISO/OSI layers, the most popular implementations include 802.11 (a variety of WiFi standards) and 802.15 (Bluetooth, Thread, Zigbee and so on). Most modern radios implement many standards and use SDR (Software Defined Radio), but due to constrained resources, use of all standards in parallel may not be possible.
Programming details are related to the specific hardware, programming language and manufacturer, but standard templates and scenarios exist. Standardisation occurs mainly in the context of higher-level protocols, such as CoAP, MQTT, HTTP, and similar - many parts of the code are interchangeable between MCUs thanks to the HAL (Hardware Abstraction Layer) libraries.

Fog class devices use OS-level networking for low-level communication. Thus, there is no need to explicitly write a code that connects, e.g. to the WiFi AP, set up one or makes Bluetooth pairing: it is configured and handled on the OS level.
That is not the case in edge-class devices, where developers must implement and control the full networking stack. For this reason, parallel and asynchronous programming techniques are extensively used (such as multitasking, asynchronous and even multicore programming) because various communication tasks need to be addressed in the background while the device's main logic is running.

In the previous chapters, some examples were presented for sensors, actuators and other interesting elements, but without computer networks. This chapter presents some elementary programming examples for networking using Espressif SoCs as the Edge-class devices and Raspberry PI as the Fog-class device. Espressif SoCs can be used as WiFi network controllers connected to other microcontrollers (e.g. Arduino Uno) and programmed with “AT” commands. They can also be stand-alone microcontrollers with on-board network capabilities.
ESP8266 and ESP32 can use WiFi connectivity, while many versions of ESP32 can also use Bluetooth. In WiFi networks, the Espressif chips can operate as the network client (like a regular computer connected to the Access Point or Router), as the network provider (Access Point), or in both modes simultaneously (as the Repeater).
Currently, none of the STM32 devices supports WiFi, but the Wireless Series of the STM32 family have built-in Bluetooth radio and can use other 802.15.4 IoT protocols.
Nordic Semiconductor nRF52 family of SoCs supports Bluetooth, Bluetooth Low Energy,

and 802.15.4 protocols. These chips cannot connect to the WiFi network directly.

In the first part of this chapter, the emphasis will be put on the ESP8266 SoC, and all the examples will present software for this chip. The second part of the chapter presents some Scripting examples in Python (and Micropython) for the Fog and Edge classes of IoT devices.

7.1. Networking for Espressif

Expressif Networking Modes Explained

Espressif SoC devices ESP32 and ESP8266 can use a few WiFi network modes (applies to layers 1–3 in ISO-OSI model). ESP8266 or ESP32 SoC can act as an Access Point (AP) - a device to connect to like connecting a notebook to the Internet router, and as a client - ESP then behaves like any WiFi-enabled device, e.g. tablet or mobile phone, connecting to the Internet infrastructure. Interestingly, Espressif SoCs can act simultaneously in both modes simultaneously, even if they have only one WiFi interface! Espressif SoCs can operate in the following modes:

- as WiFi client connected to WiFi router (figure 316),

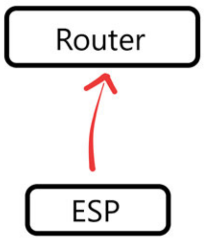


Figure 316: ESP client mode

- as independent WiFi access point (figure 317),

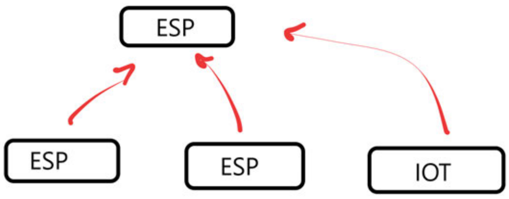


Figure 317: ESP AP mode

- as a repeater with devices connected to ESP and ESP connected to an external router (figure 318),

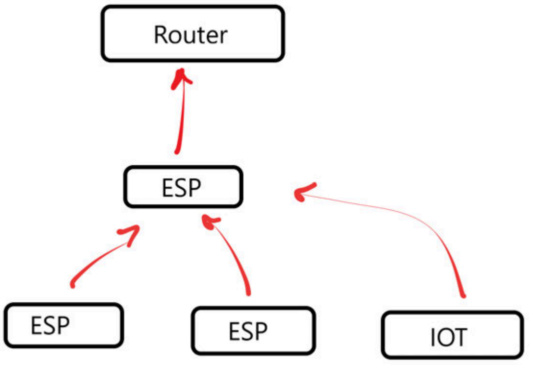


Figure 318: ESP dual mode

- as client and server in mesh network (figure 319).

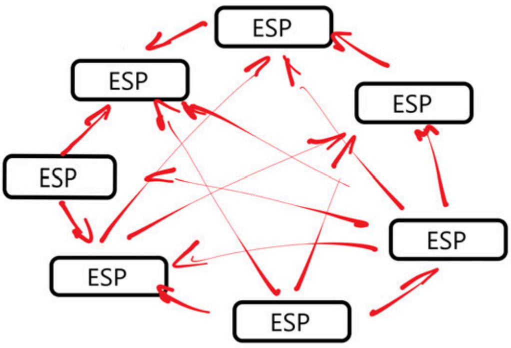


Figure 319: ESP mesh networking

7.1.1. ESP AT Networking

ESP8266 SoC can work as the WiFi communication module for other microcontrollers. To use the ESP8266 chip as a modem (figure 320), the module must be flashed with the appropriate AT-command firmware. Espressif and other developers prepared the ready-to-use firmware with the AT-command interpreter. This firmware can be downloaded from the web and flashed into ESP8266 memory with a flash tool.

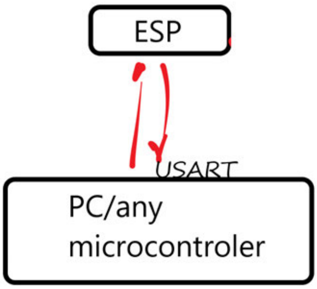


Figure 320: ESP8266 as a modem

AT commands were developed to control telephony modems. They are often used to control modules connected via a serial port, including GPS receivers, GSM/LTE modems, network modules, and others.

Preparing an ESP8266 chip with AT commands firmware

Downloading Software

- Download the latest ESP Flash Download Tool (v3.9.5 at the time of writing) from [181].

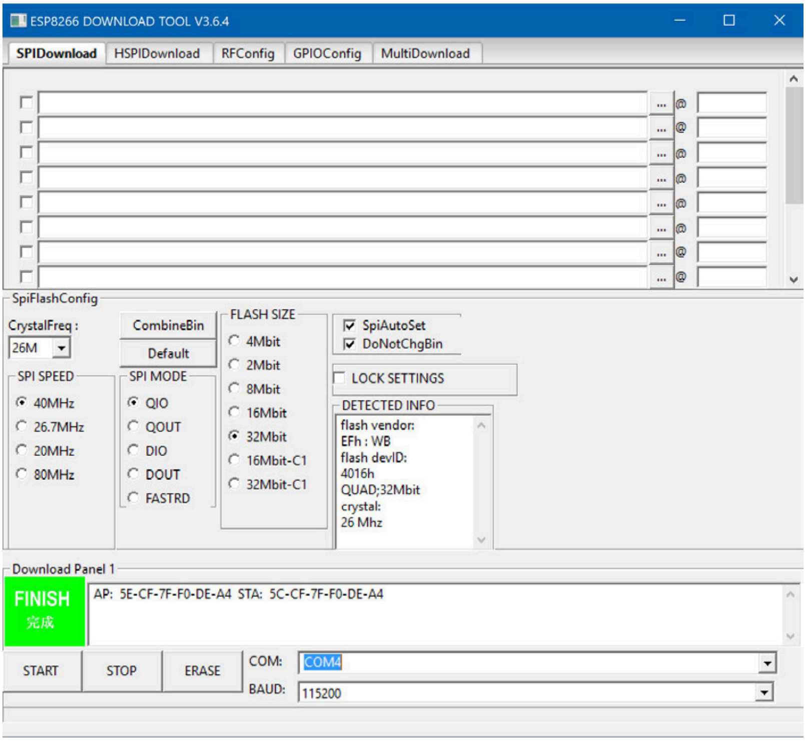
Other flashing tools like NodeMcu Flasher [182] exist. While using a single binary file, other flashing tools can be used like esp8266 flasher [183], Tasmotizer [184] or others.

- Download the latest AT release from [185]

The newest version of Espressif firmware is not compatible with ESP8266 SoCs. In the case of using ESP8266-based boards, download older AiThinker firmware available on GitHub [186]. The firmware can come in different versions. It can be a set of binary files that must be uploaded to specific memory addresses or as a combined single binary file. Note that a single file is prepared for a particular flash memory size.

Flashing Procedure with single binary file

- Detect ESP8266 module parameters. Start the ESP Flash Download Tool (“ESPFlashDownloadTool_v3.9.5”), set the COM port corresponding to your programmer, and then click the START button to detect the board's specs. After detection, one should see something like this (figure 321):



SpiFlashConfig

CrystalFreq : 26M

SPI SPEED: (● 40MHz) 26.7MHz 20MHz 80MHz

SPI MODE: (● QIO) QOUT DIO DOUT FASTRD

FLASH SIZE: 4Mbit 2Mbit 8Mbit 16Mbit (● 32Mbit) 16Mbit-C1 32Mbit-C1

☑ SpiAutoSet
☑ DoNotChgBin
☐ LOCK SETTINGS

DETECTED INFO
flash vendor:
EFh : WB
flash devID:
4016h
QUAD;32Mbit
crystal:
26 Mhz

Download Panel 1

FINISH 完成

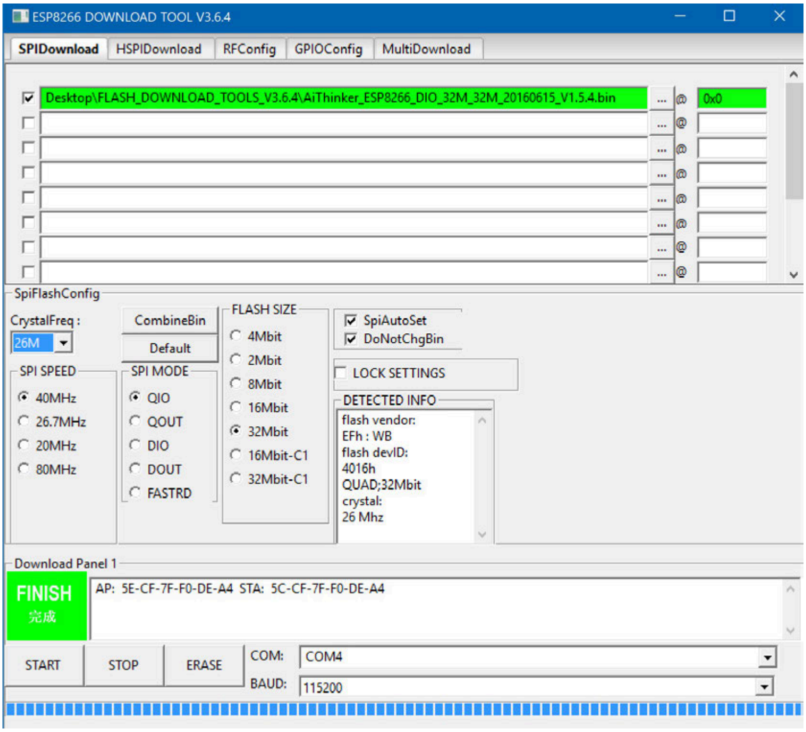
AP: 5E-CF-7F-F0-DE-A4 STA: 5C-CF-7F-F0-DE-A4

COM: COM4

BAUD: 115200

Figure 321: Programming ESP8266 - detected parameters

- Gather information. Make a note of the flash memory size. In this example, we have a 32 Mbit flash.
- Load the correct size of the combined AT binary firmware file (”.bin“) and set the offset as 0×0; one should see something like the view present in figure 322.



[✓] Desktop\FLASH_DOWNLOAD_TOOLS_V3.6.4\AiThinker_ESP8266_DIO_32M_32M_20160615_V1.5.4.bin @ 0x0

CrystalFreq : 26M

SPI SPEED: [✓ 40MHz] 26.7MHz 20MHz 80MHz

SPI MODE: [✓ QIO] QOUT DIO DOUT FASTRD

FLASH SIZE: 4Mbit 2Mbit 8Mbit 16Mbit [✓ 32Mbit] 16Mbit-C1 32Mbit-C1

[✓] SpiAutoSet
[✓] DoNotChgBin

DETECTED INFO
flash vendor:
EFh : WB
flash devID:
4016h
QUAD;32Mbit
crystal:
26 Mhz

AP: 5E-CF-7F-F0-DE-A4 STA: 5C-CF-7F-F0-DE-A4

COM: COM4
BAUD: 115200

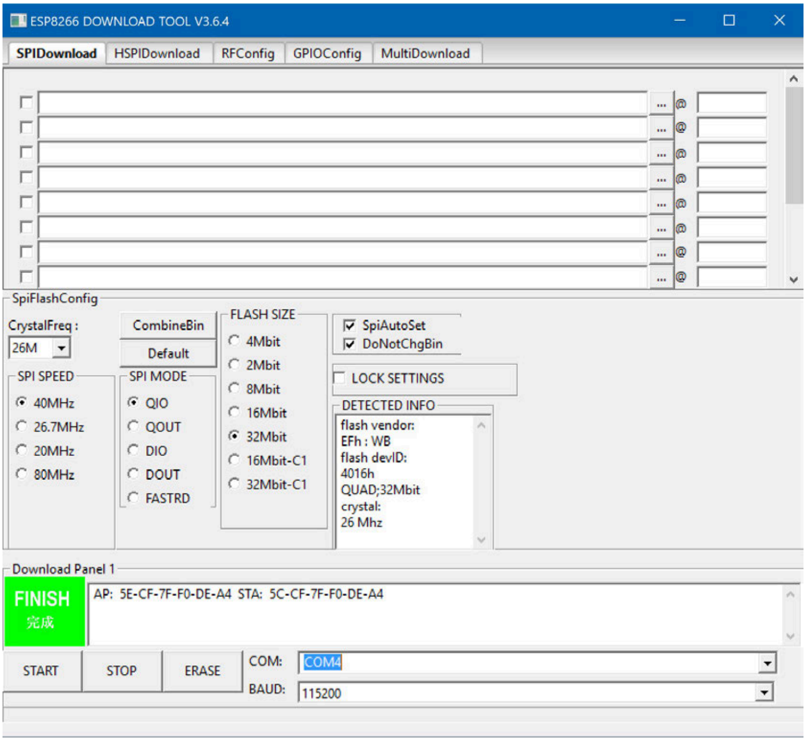
Figure 322: Programming ESP8266 - setting proper image file

- Click the START button and wait until the flashing process ends.

Flashing Procedure with a set of separate files

To flash the firmware from a set of files or to restore the original firmware:

- Detect ESP8266 module parameters. Start the ESP Flash Download Tool ("ESPFlashDownloadTool_v3.9.5"), set the COM port corresponding to your programmer, and then click the START button to detect the board's specs. After detection, you should see something like the view present in figure 323.



CrystalFreq : 26M

SPI SPEED: [✓ 40MHz] 26.7MHz 20MHz 80MHz

SPI MODE: [✓ QIO] QOUT DIO DOUT FASTRD

FLASH SIZE: 4Mbit 2Mbit 8Mbit 16Mbit [✓ 32Mbit] 16Mbit-C1 32Mbit-C1

☑ SpiAutoSet
☑ DoNotChgBin
☐ LOCK SETTINGS

DETECTED INFO
flash vendor:
EFh : WB
flash devID:
4016h
QUAD;32Mbit
crystal:
26 Mhz

Download Panel 1

AP: 5E-CF-7F-F0-DE-A4 STA: 5C-CF-7F-F0-DE-A4

COM: COM4
BAUD: 115200

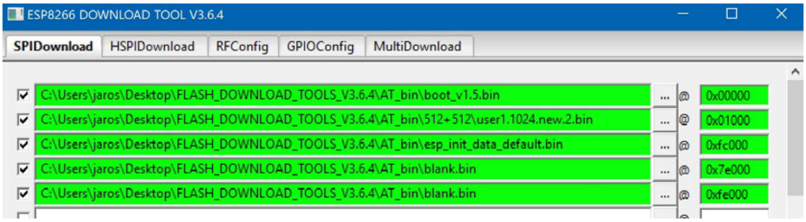
Figure 323: Programming ESP8266 - detected parameters

- From the downloaded AT firmware folder, open the "readme.txt" file containing the information for flashing the firmware. Inside the file, there should be a "BOOT MODE" section as follows:

```
# BOOT MODE
## download
### Flash size 8Mbit: 512KB+512KB
    boot_v1.2+.bin              0x00000
    user1.1024.new.2.bin        0x01000
    esp_init_data_default.bin   0xfc000 (optional)
    blank.bin                   0x7e000 & 0xfe000

### Flash size 16Mbit: 512KB+512KB
    boot_v1.5.bin               0x00000
    user1.1024.new.2.bin        0x01000
    esp_init_data_default.bin   0x1fc000 (optional)
    blank.bin                   0x7e000 & 0x1fe000

### Flash size 16Mbit-C1: 1024KB+1024KB
    boot_v1.2+.bin              0x00000
    user1.2048.new.5.bin        0x01000
    esp_init_data_default.bin   0x1fc000 (optional)
    blank.bin                   0xfe000 & 0x1fe000
```

```
### Flash size 32Mbit: 512KB+512KB
    boot_v1.2+.bin              0x00000
    user1.1024.new.2.bin        0x01000
    esp_init_data_default.bin   0x3fc000 (optional)
    blank.bin                   0x7e000 & 0x3fe000

### Flash size 32Mbit-C1: 1024KB+1024KB
    boot_v1.2+.bin              0x00000
    user1.2048.new.5.bin        0x01000
    esp_init_data_default.bin   0x3fc000 (optional)
    blank.bin                   0xfe000 & 0x3fe000
```

- Indicate – correct for your ESP8266 flash size – firmware files & addresses. The firmware is broken down into several files. They must be provided to the ESP Flash Download Tool and the corresponding addresses in the readme.txt file above. For our ESP8266 example, it should look like in figure 324.



```
☑ C:\Users\jaros\Desktop\FLASH_DOWNLOAD_TOOLS_V3.6.4\AT_bin\boot_v1.5.bin                 @ 0x00000
☑ C:\Users\jaros\Desktop\FLASH_DOWNLOAD_TOOLS_V3.6.4\AT_bin\512+512\user1.1024.new.2.bin  @ 0x01000
☑ C:\Users\jaros\Desktop\FLASH_DOWNLOAD_TOOLS_V3.6.4\AT_bin\esp_init_data_default.bin     @ 0xfc000
☑ C:\Users\jaros\Desktop\FLASH_DOWNLOAD_TOOLS_V3.6.4\AT_bin\blank.bin                     @ 0x7e000
☑ C:\Users\jaros\Desktop\FLASH_DOWNLOAD_TOOLS_V3.6.4\AT_bin\blank.bin                     @ 0xfe000
```

Figure 324: Programming ESP8266 - reflashing settings

- Click the START button and wait until the flashing process ends.

Basic ESP8266 Networking

After uploading AT firmware and connecting the module to the PC, an ESP8266 can be used as a modem with simple AT commands.

It is possible to connect ESP8266 to a PC with a TTL-Serial-to-USB adapter. Connection to any microcontroller with a serial interface does not need an adapter. The default baud rate settings are 115200,N,8,1. To check if the module works properly, a simple “AT” command can be used:

```
AT
```

If the response is “OK”, the ESP8266 module is ready to use and accept other commands. For example, to figure out exactly what firmware version is installed, the following command can be used:

```
AT+GMR
```

The AT command interpreter requires full “Enter” code. Both “CR” and “LF” characters must be sent. Some popular terminal programs like Putty do not send both characters. Be sure that the serial terminal software sends “CRLF” at the end of the line.

As a WiFi device, ESP8266 can connect to the network in such modes:

- mode 1 - client mode - the ESP8266 connecting to an existing wireless network,
- mode 2 - access point mode (AP) - other wireless network devices can be connected to the ESP8266,
- mode 3 - dual mode (router) - the ESP8266 is an access point and connects
- simultaneously to an existing wireless network.

By default, the ESP8266's stock firmware is set to AP mode. To confirm that, send the following command:

```
AT+CWMODE?
```

The response should look like `+CWMODE:2`, where 2 corresponds to AP mode. To switch ESP8266 to client device mode, the following command can be used:

```
AT+CWMODE=1
```

To scan the airwaves for all WiFi access points in range, the following command can be used:

```
AT+CWLAP
```

Then, the ESP8266 will return a list of all the access points in range. In each line will be an item consisting of the security level of the access point, the network name, the signal strength, the MAC address, and the wireless channel used. Possible security levels of the access point <0-4> mean:

- 0 - open,
- 1 - WEP,
- 2 - WPA_PSK,
- 3 - WPA2_PSK,
- 4 - WPA_WPA2_PSK.

The following command establishes the connection to the available access point with proper `ssid_name` and `password`:

```
AT+CWJAP=<ssid_name>,<password>
```

If everything is OK, the ESP8266 will answer:

```
WIFI CONNECTED
WIFI GOT IP
OK
```

ESP8266 is connected to the chosen AP and obtained a proper IP address. The following command checks what is the assigned IP address:

```
AT+CIFSR
```

To set up ESP8266 to behave both as a WiFi client and a WiFi Access point, the mode should be set to 3:

```
AT+CWMODE=3
```

7.1.2. Programming ESP8266 for the Network

Programming networking services with Espressif SoCs requires the connection established on the networking layer between parties, mainly with TCP protocol.
Below are two code examples for ESP8266 of how to implement access point and network station modes using libraries that came during installation of the development environment for Arduino framework.
The third example shows how to send and receive a UDP packet while in client mode. It is the full solution to connect ESP8266 to the NTP (Network Time Protocol) server to obtain the current date and time from the Internet.
Examples on further pages show how to make a handy WiFi scanner showing available networks nearby.

ESP8266 AP (Access Point) Mode

Based on a standard example, this program demonstrates how to program ESP8266 in AP mode. After compilation and uploading this program, an ESP8266 starts serving as the access point that can be connected to, e.g., a smartphone. It presents a simple web server available at the local IP address 192.168.4.1 (the default address of the ESP access point). This web server responds with a short message: “You are connected”.

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>

/* Set these variables to your desired credentials. */
const char *ssid = "APmode";
const char *password = "password";

ESP8266WebServer server(80);

void hRoot() {
        server.send(200, "text/html", "<h1>You are connected</h1>");
}
```

```
/* Initialization */
void setup() {
        delay(1500);
        /* You can remove the password parameter
           if you want the AP to be open. */
        WiFi.softAP(ssid, password);

        IPAddress myIP = WiFi.softAPIP();

        server.on("/", hRoot);
        server.begin();
}

void loop() {
        server.handleClient();
}
```

ESP8266 Client Mode

This standard example demonstrates how to program ESP8266 in client mode. It tries to connect to the WiFi network with a specified name (SSID) and password.

```
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>

ESP8266WiFiMulti WiFiMulti;

void setup() {
    delay(1000);

    // Initialise serial port to monitor program behaviour
    Serial.begin(115200);

    // We start by connecting to a WiFi network
    WiFi.mode(WIFI_STA);
    WiFiMulti.addAP("SSID", "password");

    while(WiFiMulti.run() != WL_CONNECTED) {
        delay(500);
    }
    delay(500);
}


void loop() {
    const uint16_t port = 80;
    const char * host = "192.168.1.1"; // ip or dns

    // Use WiFiClient class to create TCP connections
    WiFiClient client;

    if (!client.connect(host, port)) {
        delay(5000);
        return;
    }
    // This will print the IP address assigned by the DHCP server
    Serial.println(WiFi.localIP());
    // This will send the request to the server
    client.println("Send this data to server");
    // Trying to send the GET request possibly responses (with error)
    // client.println("GET /echo");

    //read back one line from server
    String line = client.readStringUntil('\r');
    Serial.println(line);

    Serial.println("closing connection");
    client.stop();

    Serial.println("wait 5 sec...");
    delay(5000);
}
```

ESP8266 and UDP

This sketch (based on a standard example) demonstrates how to program ESP8266 as an NTP client using UDP packets (send and receive):

```
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>


char ssid[] = "**************";  //  your network SSID (name)
char pass[] = "**************";  // your network password

unsigned int localPort = 2390;   // local port to listen for UDP packets

// NTP servers
IPAddress ntpServerIP; // 0.pl.pool.ntp.org NTP server address
const char* ntpServerName[] =
 {"0.pl.pool.ntp.org","1.pl.pool.ntp.org","2.pl.pool.ntp.org","3.pl.pool.ntp.org"};

const int timeZone = 1;  //Central European Time
int servernbr=0;

// NTP time stamp is in the first 48 bytes of the message
const int NTP_PACKET_SIZE = 48;

//buffer to hold incoming and outgoing packets
byte packetBuffer[ NTP_PACKET_SIZE];

// A UDP instance to let us send and receive packets over UDP
WiFiUDP udp;

// Prototype of the function defined at the end of this file
// (required in Visual Studio Code)
void sendNTPpacket(IPAddress& address);

void setup()
{
  Serial.begin(115200);
  Serial.println();

  Serial.print("Connecting to ");
  Serial.println(ssid);
```

```
//  WiFi.persistent(false);
  WiFi.mode(WIFI_OFF);
  delay(2000);

// We start by connecting to a WiFi network
  WiFi.mode(WIFI_STA);
  delay(3000);
  WiFi.begin(ssid, pass);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");

  Serial.println("WiFi connected");
  Serial.println("DHCP assigned IP address: ");
  Serial.println(WiFi.localIP());

  Serial.println("Starting UDP");
  udp.begin(localPort);
  Serial.print("Local port: ");
  Serial.println(udp.localPort());

  // first ntp server
  servernbr = 0;
}

void loop()
{
  //get a random server from the pool

  WiFi.hostByName(ntpServerName[servernbr], ntpServerIP);
  Serial.print(ntpServerName[servernbr]);
  Serial.print(":");
  Serial.println(ntpServerIP);

  // send an NTP packet to a time server
  sendNTPpacket(ntpServerIP);

  // wait to see if a reply is available
  delay(1000);

  int cb = udp.parsePacket();
  if (!cb) {
    Serial.println("no packet yet");
    if ( servernbr = 5 ) {
      servernbr =0;
    }
    else {
      servernbr++;
    }
  }
  else {
    Serial.print("packet received, length=");
    Serial.println(cb);
    // We've received a packet, read the data from it
    // read the packet into the buffer
    udp.read(packetBuffer, NTP_PACKET_SIZE);

    // the timestamp starts at byte 40
    // of the received packet and is four bytes,
    // or two words, long. First, extract the two words:

    unsigned long highWord = word(packetBuffer[40], packetBuffer[41]);
    unsigned long lowWord = word(packetBuffer[42], packetBuffer[43]);
    // combine the four bytes (two words) into a long integer
    // this is NTP time (seconds since Jan 1 1900):
    unsigned long secsSince1900 = highWord << 16 | lowWord;
    Serial.print("Seconds since Jan 1 1900 = " );
    Serial.println(secsSince1900);

    // now convert NTP time into everyday time:
    Serial.print("Unix time = ");
    // Unix time starts on Jan 1 1970.
    // In seconds, that's 2208988800:
    const unsigned long seventyYears = 2208988800UL;
    // subtract seventy years:
    unsigned long epoch = secsSince1900 - seventyYears;
    // print Unix time:
    Serial.println(epoch);


    // print the hour, minute and second:
    // UTC is the time at Greenwich Meridian (GMT)
    Serial.print("The UTC time is ");
    // print the hour (86400 equals secs per day)
    Serial.print((epoch  % 86400L) / 3600);
    Serial.print(':');
    if ( ((epoch % 3600) / 60) < 10 ) {
      // In the first 10 minutes of each hour, we'll want a leading '0'
      Serial.print('0');
    }
    // print the minute (3600 equals secs per minute)
    Serial.print((epoch  % 3600) / 60);
    Serial.print(':');
    if ( (epoch % 60) < 10 ) {
      // In the first 10 seconds of each minute, we'll want a leading '0'
      Serial.print('0');
    }
    Serial.println(epoch % 60); // print the second
  }
  // wait ten seconds before asking for the time again
  delay(10000);
}

// send an NTP request to the time server at the given address
void sendNTPpacket(IPAddress& address)
{
  Serial.print("sending NTP packet to: ");
  Serial.println( address );
  // set all bytes in the buffer to 0
  memset(packetBuffer, 0, NTP_PACKET_SIZE);
  // Initialize values needed to form NTP request
  // (see URL above for details on the packets)
  packetBuffer[0] = 0b11100011;   // LI, Version, Mode
  packetBuffer[1] = 0;     // Stratum, or type of clock
  packetBuffer[2] = 6;     // Polling Interval
  packetBuffer[3] = 0xEC;  // Peer Clock Precision
  // 8 bytes of zero for Root Delay & Root Dispersion
  packetBuffer[12]  = 49;
  packetBuffer[13]  = 0x4E;
  packetBuffer[14]  = 49;
  packetBuffer[15]  = 52;

  // all NTP fields have been given values, now
  // you can send a packet requesting a timestamp:
  udp.beginPacket(address, 123); //NTP requests are to port 123
  udp.write(packetBuffer, NTP_PACKET_SIZE);
  udp.endPacket();
}
```

7.1.3. ESP8266 Wifi Scanner

This sketch demonstrates how to scan WiFi networks. ESP8266 is programmed in access point mode. All found WiFi networks will be printed in the serial monitor window (TTY).

```
#include "ESP8266WiFi.h"

void setup() {
  Serial.begin(115200);

  // Set WiFi to station mode and disconnect
  // from an AP if it was previously connected
  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(100);

  Serial.println("Setup done");
}

void loop() {
  Serial.println("scan start");

  // WiFi.scanNetworks will return the number of networks found
  int n = WiFi.scanNetworks();
  Serial.println("scan done");
  if (n == 0)
    Serial.println("no networks found");
  else
  {
    Serial.print(n);
    Serial.println(" networks found");
    for (int i = 0; i < n; ++i)
    {
      // Print SSID and RSSI for each network found
      Serial.print(i + 1);
      Serial.print(": ");
      Serial.print(WiFi.SSID(i));
      Serial.print(" (");
      Serial.print(WiFi.RSSI(i));
      Serial.print(")");
      Serial.println((WiFi.encryptionType(i) == ENC_TYPE_NONE)?" ":"*");
      delay(10);
    }
  }
  Serial.println("");

  // Wait a bit before scanning again
  delay(5000);
}
```

7.1.4. Controlling LED with Simple Web Server

A sample Web application hosted on ESP8266 MCU is presented below.
This application allows it to control the state of the LED remotely, connecting to the ESP8266 board with a web browser. The program presented is based on the example "HelloServer" available in the `ESP8266WebServer` library. Some modifications were made to simplify the program and to handle requests to turn the LED on and off. To check if it works, adding WiFi network credentials and setting the `led` constant with the number of GPIO to which the LED is connected is required. After a successful connection to the WiFi, ESP8266 would present through the serial monitor the IP address (e.g. 192.168.4.1). Writing in the address bar in the browser "HTTP://192.168.4.1" should return the serial monitor message "hello from esp8266!".

Assuming the address in the terminal is `192.168.4.1` one may use the following URLs to disable and enable the LED, respectively:

```
http://192.168.4.1/LED0
http://192.168.4.1/LED1
```

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>
#include <ESP8266mDNS.h>

#ifndef STASSID
#define STASSID "*********"
#define STAPSK  "*********"
#endif

const char* ssid = STASSID;
const char* password = STAPSK;

ESP8266WebServer server(80);

const int led = 2;

void handleRoot() {
  //Originally LED was controlled for every root request
  //so it is required to comment the lines which modify the LED state
  //digitalWrite(led, 1);
  server.send(200, "text/plain", "hello from esp8266!\r\n");
  //digitalWrite(led, 0);
}
```

```
void handleNotFound() {
  //digitalWrite(led, 1);
  String message = "File Not Found\n\n";
  message += "URI: ";
  message += server.uri();
  message += "\nMethod: ";
  message += (server.method() == HTTP_GET) ? "GET" : "POST";
  message += "\nArguments: ";
  message += server.args();
  message += "\n";
  for (uint8_t i = 0; i < server.args(); i++) {
    message += " " + server.argName(i) + ": " + server.arg(i) + "\n";
  }
  server.send(404, "text/plain", message);
  //digitalWrite(led, 0);
}

void setup(void) {
  pinMode(led, OUTPUT);
  //digitalWrite(led, 0);
  Serial.begin(115200);
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.println("");

  // Wait for connection
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  if (MDNS.begin("esp8266")) {
    Serial.println("MDNS responder started");
  }

  server.on("/", handleRoot);

  // request for turning led on
  server.on("/LED1", [](){
    server.send(200, "text/plain", "LED is ON");
    digitalWrite(led, 1);
  });

  // request for turning led off
  server.on("/LED0", [](){
    server.send(200, "text/plain", "LED is OFF");
    digitalWrite(led, 0);
  });

  server.onNotFound(handleNotFound);

  server.begin();
  Serial.println("HTTP server started");
}
```

```
void loop(void) {
  server.handleClient();
  MDNS.update();
}
```

7.2. Networking in Python

IoT microcontrollers contain an external or integrated communication module. Note that IoT MCUs may use a variety of wireless communication interfaces such as Bluetooth, 802.15.4 standards (Zigbee, Thread) or Lora; this chapter does not deplete all scenarios but instead presents a general idea. We have chosen a simple WiFi interface, which is the easiest to use in most scenarios. Still, we are obviously not paying attention to their drawbacks, e.g., high energy consumption. Below are code samples regarding programming in Python for Raspberry Pi and Micropython for RP2040 (Pico W) microcontrollers that integrate WiFi networking.

Connecting to the WiFi router

Python

In the case of fog class devices with Linux or Windows operating systems, connecting to the network is controlled at the OS level with configuration tools. It is possible to use Python script to execute those commands, but they are OS-specific or require a dedicated hardware-specific library installed for Python. Monitoring progress or failure is problematic and needs analysis of the standard output; thus, this approach is not advised. Once the connection is present on the OS level, Python can quickly implement application-level servers such as WWW, MQTT, CoAP, etc.

Micropython

In the case of the Micropython, it is necessary to set up a WiFi client on the Python code level and explicitly connect it to the WiFi router. A sample code that connects to the existing WiFi access point and prints the obtained configuration from the DHCP server is present below:

```
import network
import socket
from time import sleep
import machine

ssid = '<your SSID comes here>'
password = '<your WiFi passphrase comes here>'

def connect():
    #Connect to WLAN
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(ssid, password)
    while wlan.isconnected() == False:
        print('Waiting for connection...')
        sleep(1)
    print(wlan.ifconfig())

try:
    connect()
except KeyboardInterrupt:
    machine.reset()
```

Setting up an access point

Python
Similarly, in the case of the STA mode, when using fog class devices with Linux or Windows operating systems, the hosting of the WiFi access point is controlled on the OS level, and it is done with OS configuration tools.

Micropython
IoT end node (edge) devices programmed in Python, such as RPI, RP2040 (RPI Pico), ESP32, and many other network-enabled microcontrollers, can set up an access point to connect other devices. Obviously, due to the limited resources (mainly RAM), the number of hosted clients in parallel is limited. Below is a sample code (without an application layer server, just networking layer AP) for RP2040.

A simple ESP 8266 (ESP-01) IoT module can act as AP and STA in parallel, and it is possible to set up a fully functional IP router (including DHCP server and NAT) with quite decent bandwidth and automated mesh WiFi-based capability [187].

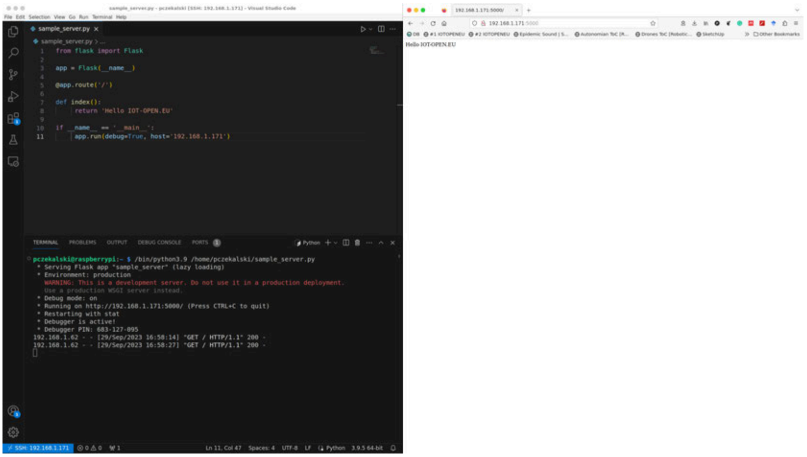
```
import network

ssid = 'MicroPython-WiFi-AP'
password = '0987654321'

ap = network.WLAN(network.AP_IF)
ap.config(essid=ssid, password=password)
ap.active(True)

while ap.active()==False:
    pass

print(ap.ifconfig());
```

Hosting a service

IoT devices and cloud solutions usually host some service, e.g. providing users with a temperature sensor reading or doing some activity, e.g. rotating a servo to unlock a smart lock at the front door. In the case of the cloud and PCs, services used to be implemented using some containerisation solution, such as Docker. End-node IoT devices (edge) have no virtualisation capability due to limited resources, lack of the multitasking OS and also because of direct access to the hardware components. However, fog-class devices such as Raspberry Pi or nVidia Jetson can use containerisation.

Python
Sample web server for RPi in Python, using Flask[188] is straightforward and takes just a few lines of code. Code, development and output are present in the figure 325. 192.168.1.171 is RPI's sample IP address obtained from the router, as the WiFi connection is managed on the OS level, not via the Python app. The default WWW port for Flask services is 5000; the sample result is in figure 325:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def index():
     return 'Hello IOT-OPEN.EU'
if __name__ == '__main__':
     app.run(debug=True, host='192.168.1.171')
```



```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def index():
    return 'Hello IOT-OPEN.EU'

if __name__ == '__main__':
    app.run(debug=True, host='192.168.1.171')
```

```
pczekalski@raspberrypi:~ $ /bin/python3.9 /home/pczekalski/sample_server.py
 * Serving Flask app "sample_server" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://192.168.1.171:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 683-127-095
192.168.1.62 - - [29/Sep/2023 16:58:14] "GET / HTTP/1.1" 200 -
192.168.1.62 - - [29/Sep/2023 16:58:27] "GET / HTTP/1.1" 200 -
```

192.168.1.171:5000

Hello IOT-OPEN.EU

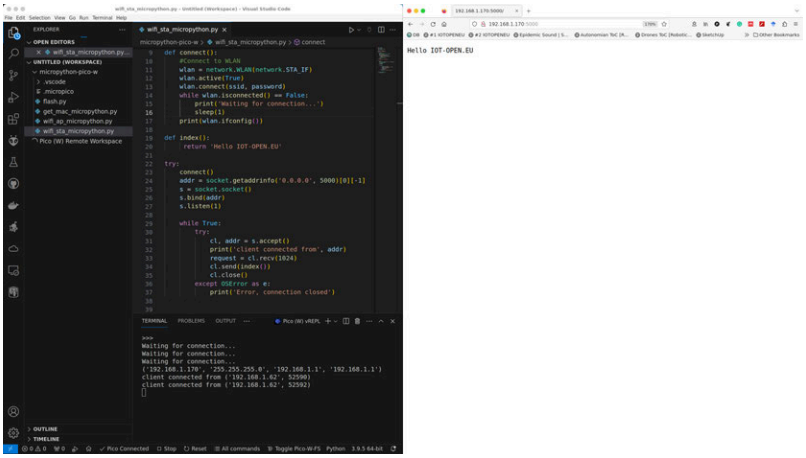
Figure 325: Sample web server for RPi, using Flask library

Micropython

In the case of Micropython, the network-level connection is included in the script, so there are two main sections in the following sample: connecting to the router and hosting a dummy service, as in the example above (for RPi). Code, development and output are present in the figure 326. The IP address of the example Micropython device is 192.168.1.170, and the service is hosted on port 5000:

```
import network
import socket
from time import sleep
import machine

ssid = '<Your WiFi SSID is here>'
password = '<Your WiFi passphrase is here>'

def connect():
    #Connect to WLAN
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(ssid, password)
    while wlan.isconnected() == False:
        print('Waiting for connection...')
        sleep(1)
    print(wlan.ifconfig())
```

```
def index():
     return 'Hello IOT-OPEN.EU'

try:
    connect()
    addr = socket.getaddrinfo('0.0.0.0', 5000)[0][-1]
    s = socket.socket()
    s.bind(addr)
    s.listen(1)

    while True:
        try:
            cl, addr = s.accept()
            print('client connected from', addr)
            request = cl.recv(1024)
            cl.send(index())
            cl.close()
        except OSError as e:
            print('Error, connection closed')

except KeyboardInterrupt:
    machine.reset()
```



```
def connect():
    #Connect to WLAN
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(ssid, password)
    while wlan.isconnected() == False:
        print('Waiting for connection...')
        sleep(1)
    print(wlan.ifconfig())

def index():
     return 'Hello IOT-OPEN.EU'

try:
    connect()
    addr = socket.getaddrinfo('0.0.0.0', 5000)[0][-1]
    s = socket.socket()
    s.bind(addr)
    s.listen(1)

    while True:
        try:
            cl, addr = s.accept()
            print('client connected from', addr)
            request = cl.recv(1024)
            cl.send(index())
            cl.close()
        except OSError as e:
            print('Error, connection closed')
```

```
>>>
Waiting for connection...
Waiting for connection...
Waiting for connection...
('192.168.1.170', '255.255.255.0', '192.168.1.1', '192.168.1.1')
client connected from ('192.168.1.62', 52590)
client connected from ('192.168.1.62', 52592)
```

```
Hello IOT-OPEN.EU
```

Figure 326: Sample web server for RP2040 (Pico W)

An example of how to integrate GPIO and web service in one solution for Micropython can be found on the official RPI website [189].

8. IoT Frameworks and Firmware

Internet of Things frameworks play a crucial role in developing IoT applications by providing integration systems for implementing home automation complemented with ready-to-use firmware for various hardware platforms. It allows the developers to create whole control systems and IoT devices without writing the entire software from scratch. This makes developing new ideas easier for non-experienced beginners and enthusiasts, in some situations, to modify the behaviour of devices available on the market. What is even more important is that they make it possible to integrate IoT equipment coming from different vendors. There are several IoT frameworks available, with popular home automation systems: Domoticz[190], OpenHAB[191], Home Assistant[192], and ready-to-use firmware including Tasmota[193], ESPHome[194], ESPEasy[195], and ESPurna[196]. All kinds of firmware initially was developed for ESP8266 SoCs but now have been redesigned to support ESP32 and other hardware platforms. With the appearance of new microcontrollers by Beken and Realtec companies, new versions of firmware were created with OpenBeken[197] as the example. Each of these firmware choices has its characteristics and use cases. They usually implement MQTT communication protocol and specific protocols used in popular home automation systems, including Domoticz, OpenHAB, and Home Assistant.

Tasmota
Tasmota is popular among IoT enthusiasts and developers who want complete control and customization over their devices. It supports a wide variety of sensors and output devices. Tasmota uses a web interface for configuration, making it easy to configure the hardware connection of the microcontroller and peripheral elements. Tasmota provides scripting capabilities, allowing users to define complex internal automation rules. The support for the MQTT protocol allows easy integration with home automation platforms. What is very important is that Tasmota has an active and supportive user community that constantly extends the software's possibilities, including support for ESP32-based devices.

ESPEasy
ESPEasy is designed for users who want a simplified IoT device configuration and automation approach. It offers a user-friendly web interface for configuring devices. It provides a set of pre-built plugins for everyday tasks and supports MQTT for integration with platforms like Domoticz and OpenHAB. ESPEasy has an active community, although not as active as Tasmota.

ESPHome
ESPHome is popular among Home Assistant users who want a seamless integration experience. It uses a YAML-based configuration, which is highly readable and well-documented. It's tightly integrated with Home Assistant, making it an excellent choice for users. The IoT node can be configured in the Home Automation system, automatically generating the proper firmware version with the final unit's configuration. ESPHome allows users to define device configurations, sensor readings, and automation straightforwardly. ESPHome users benefit from the Home Assistant community, providing strong support.

ESPurna

ESPurna is the least active but still interesting project. ESPurna supports MQTT for home automation systems integration and compatibility with Domoticz, Home Assistant, and other platforms.

OpenBeken
OpenBeken is the software created for the BK72xx series of SoCs based on Tasmota functionality.

Table 40: Frameworks and their hardware compatibility

Framework	Platform
Tasmota	ESP8266, ESP32
ESPHome	ESP8266, ESP32, RP2040, BK72xx, RTL87xx
ESPEasy	ESP8266, ESP32
ESPurna	ESP8266
OpenBeken	BK72xx

The choice between Tasmota, ESP Easy, ESPHome, ESPurna and OpenBeken largely depends on the user's specific needs, selection of hardware platform, and familiarity with IoT device configuration. Table 40 presents hardware compatibility.

Node-RED tool

Node-RED [198] is an open-source, flow-based development tool and runtime environment designed for visual programming. IBM Emerging Technology Services initially developed it and is now part of the OpenJS Foundation. It can be used for any purpose that uses a flow-based programming model, which is especially useful for IoT (Internet of Things) and home automation applications.

Key aspects and features of Node-RED:

- Flow-Based Programming - Node-RED uses a flow-based programming paradigm where developers create applications by connecting nodes in a visual editor. Each node represents a data source, data output, or a specific function or task. Flows, defined as sequences of connected nodes, represent the logic and behaviour of the application.

- Visual Editor - Node-RED has a web-based visual editor, which makes it easy for users to create and edit flows. The editor provides a drag-and-drop interface for adding, configuring, and connecting nodes and flows to build applications visually.

- Extensible and Customizable - Node-RED has a wide range of pre-built nodes that can be used for various tasks but are also highly extensible. Users can install additional nodes from the Node-RED library, allowing for integration with various hardware devices, services, and protocols. It is also possible to create custom nodes with the functionality programmed in JavaScript.

- Integration Capabilities - Node-RED connects and integrates with various devices, platforms, and APIs. It has built-in nodes for MQTT, HTTP, WebSocket, and more. That's why it is popular in IoT and home automation.

- Debugging and Logging - Node-RED provides built-in debugging and logging capabilities, making it easier to troubleshoot and monitor the behaviour of your applications.

- Open Source - Node-RED is open source and has a vibrant and active community of users and developers. This community contributes to its development and maintains a repository of third-party nodes.

Node-RED is used in various applications, including home automation, industrial automation, data processing, and IoT solutions. Its visual approach to programming and extensive library of nodes make it a valuable tool for rapidly prototyping and building applications that involve data processing and automation.

9. Notes for Further Studying

It is worth mentioning that new IoT ideas, hardware, software, and applications are introduced every second. Because of that, technical, specific knowledge, mostly on hardware and software, becomes rapidly outdated. Moreover, due to the amount of information related to embedded systems development and IoT development, it is impossible to assemble all information regarding the IoT world.

The IOT-OPEN.EU project is instantly evolving and always brings new content, but it cannot be the only source of knowledge in the current stage of development. We distribute all content via a single starting point, the website http://iot-open.eu, but we suggest navigating to the online resources presented below. Those projects, websites and resources are not related to our project. Still, we consider them a valuable source.

Please also note even if the IOT-OPEN.The EU project is CC BY-NC licenced, and the resources listed below may need an access fee, registration, etc.

Many online platforms provide online courses by different universities on relevant topics like the Internet of Things, embedded systems, programming languages, connectivity and security, robotics, big data, computer vision, and many more. Some of the most popular platforms are **Coursera** [199], **edX** [200], **Udacity** [201], **Udemy** [202], **Skillshare** [203]. Some of these courses are free of charge, and at the end of these courses, a certificate about skills can be acquired (often for an additional price).

Electronics Tutorials website [204] offers multiple basic electronics tutorial topics, including AC and DC circuit theory, amplifiers, semiconductors, filters, Boolean algebra, capacitors, power electronics, transistors, operational amplifiers, sequential logic, and many more. It contains an extensive description of the theory with graphics and explanations.

Embedded Experts website [205] focuses on commercial, certified courses mainly related to the embedded platforms. It may help study technologies related to the Edge Class and Fog Class devices that are fundamental for IoT and bare metal IoT development.

Instructables [206] is a project platform with plenty of Internet of Things projects for different knowledge levels. It is also possible to enrol on other classes with many lessons that teach about specific related topics that are not limited only to electronics but also cover issues such as sewing, food, craft, 3D printing, etc. One section of the Instructables website offers multiple contests and challenges related to the topic, with valuable prizes.

Tinkercad is a simple, online 3D design and 3D platform that also allows users to model and test circuits (https://www.tinkercad.com/circuits). With Tinkercad, it is possible to program and simulate a virtual Arduino board online using different libraries and serial monitors. There are also plenty of starter examples already available.

Wokwi [207] is an online (in-browser) IoT device simulator. You can implement some simple and limited approaches, going beyond embedded systems. Note that it is not a distant lab (such as our IOT-OPEN.EU VREL lab [208]) but rather a software simulation of the IoT hardware development boards, so your experience will be limited.

---

[1] “ITU Internet Reports 2005: The Internet of Things.” http://www.itu.int/osg/spu/publications/internetofthings/

[2] “Special Report: The Internet of Things”, in “the Institute”, IEEE 2014, http://theinstitute.ieee.org/static/special-report-the-internet-of-things

[3] “Towards a definition of the Internet of Things (IoT)”, IEEE 2015

[4] Standard for an Architectural Framework for the Internet of Things (IoT) http://grouper.ieee.org/groups/2413/

[5] Ovidiu Vermesan, Peter Friess (eds.): Digitising the Industry, Internet of Things Connecting the Physical, Digital and Virtual Worlds, River Publishers Series in Communications, 2016

[6] Vision and Challenges for Realising the Internet of Things, CERP-IoT 2010, http://www.internet-of-things-research.eu/pdf/IoT_Clusterbook_March_2010.pdf

[7] Salim Elbouanani, My Ahmed El Kiram, Omar Achbarou: “Introduction To The Internet Of Things Security. Standardisation and research challenges”, 2015 11th International Conference on Information Assurance and Security (IAS), IEEE 2015

[8] Ovidiu Vermesan, Peter Friess (eds.): Digitising the Industry, Internet of Things Connecting the Physical, Digital and Virtual Worlds, River Publishers Series in Communications, 2016

[9] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, Moussa Ayyash: Internet of Things: A Survey on Enabling Technologies, Protocols and Applications, IEEE Communications Surveys & Tutorials, Volume: 17, Issue: 4, 2015

[10] Arslan Munir, IFCIoT: Integrated Fog Cloud IoT Architectural Paradigm for the Future Internet of Things, IEEE Consumer Electronics Magazine, Vol. 6, Issue 3, July 2017

[11] Arslan Munir, IFCIoT: Integrated Fog Cloud IoT Architectural Paradigm for the Future Internet of Things, IEEE Consumer Electronics Magazine, Vol. 6, Issue 3, July 2017

[12] S.Matthews at http://www.ibmbigdatahub.com/blog/what-cognitive-iot, Cited: 11.06.2018.

[13] Cloudonomics: The Business Value of Cloud Computing

[14] Top 10 IoT security challenges

[15] IOTA: A permissionless distributed ledger for a new economy

[16] https://www.researchgate.net/publication/273389706_Towards_a_smart_city_based_on_cloud_of_things_a_survey_on_the_smart_city_vision

[17] https://hal.archives-ouvertes.fr/hal-01581127/file/2016-TE2016-Taxonomy-for-IoT-Sensors.pdf

[18] https://www.w3.org/WoT/IG/wiki/Use_cases_across_application_domains#Use_Cases_and_Applications

[19] https://hal.archives-ouvertes.fr/hal-01581127/file/2016-TE2016-Taxonomy-for-IoT-Sensors.pdf

[20] http://internetofthingsagenda.techtarget.com/blog/IoT-Agenda/IoT-as-a-solution-for-precision-farming

[21] https://hal.archives-ouvertes.fr/hal-01581127/file/2016-TE2016-Taxonomy-for-IoT-Sensors.pdf

[22] https://news.panasonic.com/global/topics/2015/44009.html

[23] https://www.fitbit.com

[24] http://www.businessinsider.com/wearable-technology-iot-devices-2016-8

[25] https://www.freertos.org/
[26] https://platformio.org/
[27] https://docs.platformio.org/en/latest/librarymanager/index.html
[28] https://tasmota.github.io/docs/
[29] https://esphome.io/index.html
[30] https://github.com/openshwprojects/OpenBK7231T_App
[31] https://github.com/letscontrolit/ESPEasy
[32] https://github.com/xoseperez/espurna
[33] https://www.freertos.org/
[34] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html
[35] https://gcc.gnu.org/
[36] https://www.eclipse.org/ide/
[37] https://www.youtube.com/watch?v=YSyaH5v3hys
[38] https://platformio.org/
[39] https://www.arduino.cc/en/software
[40] https://docs.platformio.org/en/stable/projectconf/index.html
[41] https://docs.platformio.org/en/latest/librarymanager/index.html
[42] https://www.arduino.cc/reference/en/language/functions/time/delay/
[43] https://www.arduino.cc/reference/en/language/functions/time/millis/
[44] https://www.arduino.cc/en/Tutorial/DigitalPins
[45] https://www.arduino.cc/reference/en/language/functions/communication/serial/readstring/
[46] https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-reference/peripherals/adc.html
[47] https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-reference/peripherals/dac.html
[48] https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/api/timer.html
[49] https://dotnet.microsoft.com/en-us/apps/iot
[50] https://www.raspberrypi.com/documentation/computers/getting-started.html
[51] https://raspberrytips.com/install-pycharm-raspberry-pi/
[52] https://www.youtube.com/watch?v=GssM7hkwJr
[53] https://micropython.org
[54] https://learn.sparkfun.com/tutorials/pro-micro-rp2040-hookup-guide/examples-micropython
[55] https://code.visualstudio.com/docs/remote/remote-overview
[56] https://raspberrytips.com/thonny-ide-raspberry-pi/
[57] https://learn.microsoft.com/en-us/previous-versions/windows/iot-core/connect-your-device/iotdashboard?source=recommendations
[58] https://www.tutorialspoint.com/csharp/
[59] https://www.tutorialspoint.com/csharp/
[60] https://www.tutorialspoint.com/csharp/

[61] https://www.tutorialspoint.com/csharp/

[62] Internet of Things: Architectures, Protocols, and Applications; P. S. Smruti, R. Sarangi. https://doi.org/10.1155/2017/9324035

[63] Internet of Things: Security Vulnerabilities and Challenges; I. Andrea, C. Chrysostomou, G. Hadjichristofi, The 3rd IEEE ISCC 2015 International Workshop on Smart City and Ubiquitous Computing Applications, https://doi.org/10.1109/ISCC.2015.7405513

[64] Rajan Arora, I2C Bus Pullup Resistor Calculation, Texas Instruments Application Report

[65] https://www.maximintegrated.com/en/products/digital/one-wire.html

[66] https://en.wikipedia.org/wiki/ESP8266

[67] http://espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

[68] https://en.wikipedia.org/wiki/ESP8266

[69] https://www.esp8266.com/wiki/doku.php?id=esp8266-module-family

[70] https://www.wemos.cc/

[71] https://en.wikipedia.org/wiki/ESP8266

[72] https://en.wikipedia.org/wiki/ESP8266

[73] https://en.wikipedia.org/wiki/ESP8266

[74] https://en.wikipedia.org/wiki/ESP8266

[75] https://www.wemos.cc/

[76] https://www.espressif.com

[77] https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[78] https://www.espressif.com/sites/default/files/documentation/esp32-pico-d4_datasheet_en.pdf

[79] https://docs.espressif.com/projects/esp-idf/en/v4.3/esp32/hw-reference/modules-and-boards.html

[80] https://www.espressif.com/sites/default/files/documentation/esp32-pico-d4_datasheet_en.pdf

[81] https://www.espressif.com/sites/default/files/documentation/esp32-pico-v3_datasheet_en.pdf

[82] https://www.espressif.com/sites/default/files/documentation/esp32-pico-v3-02_datasheet_en.pdf

[83] https://www.espressif.com/sites/default/files/documentation/esp32-pico-mini-02_datasheet_en.pdf

[84] https://www.espressif.com/sites/default/files/documentation/esp32-pico-v3-zero_datasheet_en.pdf

[85] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html

[86] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-pico-kit.html

[87] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-pico-kit-1.html

[88] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-pico-devkitm-2.html

[89] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-pico-kit.html
[90] https://docs.m5stack.com/en/atom/atomhub
[91] https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf
[92] https://www.espressif.com/en/products/modules
[93] https://docs.espressif.com/projects/esp-idf/en/latest/esp32s2/hw-reference/esp32s2/user-guide-devkitm-1-v1.html
[94] https://docs.espressif.com/projects/esp-idf/en/latest/esp32s2/hw-reference/esp32s2/user-guide-s2-devkitc-1.html
[95] https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf
[96] https://www.espressif.com/sites/default/files/documentation/esp32-s3-pico-1_datasheet_en.pdf
[97] https://www.espressif.com/sites/default/files/documentation/esp32-s3-mini-1_mini-1u_datasheet_en.pdf
[98] https://www.espressif.com/sites/default/files/documentation/esp32-s3-wroom-1_wroom-1u_datasheet_en.pdf
[99] https://www.espressif.com/sites/default/files/documentation/esp32-s3-wroom-2_datasheet_en.pdf
[100] https://www.espressif.com/en/products/modules
[101] https://www.waveshare.com/esp32-s3-pico.htm
[102] https://shop.m5stack.com/products/m5stamps3-with-2-54-header-pin
[103] https://docs.espressif.com/projects/esp-idf/en/v5.0/esp32/hw-reference/chip-series-comparison.html
[104] https://www.espressif.com/sites/default/files/documentation/esp8684_datasheet_en.pdf
[105] https://www.espressif.com/sites/default/files/documentation/esp8684_datasheet_en.pdf
[106] https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf
[107] https://www.espressif.com/sites/default/files/documentation/esp32-c3-mini-1_datasheet_en.pdf
[108] https://www.espressif.com/sites/default/files/documentation/esp32-c3-wroom-02_datasheet_en.pdf
[109] https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html
[110] https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/hw-reference/esp32c3/user-guide-devkitc-02.html
[111] https://docs.espressif.com/projects/espressif-esp-dev-kits/en/latest/esp32c3/esp32-c3-lcdkit/user_guide.html
[112] https://www.adafruit.com/product/5405
[113] https://wiki.seeedstudio.com/XIAO_ESP32C3_Getting_Started/
[114] https://shop.m5stack.com/products/m5stamp-c3-mate-with-pin-headers
[115] https://docs.espressif.com/projects/esp-idf/en/v5.0/esp32/hw-reference/chip-series-comparison.html

[116] https://www.espressif.com/sites/default/files/documentation/esp32-c6_datasheet_en.pdf
[117] https://docs.espressif.com/projects/espressif-esp-dev-kits/en/latest/esp32c6/esp32-c6-devkitm-1/index.html
[118] https://docs.espressif.com/projects/espressif-esp-dev-kits/en/latest/esp32c6/esp32-c6-devkitc-1/index.html
[119] https://www.espressif.com/sites/default/files/documentation/esp32-h2_datasheet_en.pdf
[120] https://docs.espressif.com/projects/espressif-esp-dev-kits/en/latest/esp32h2/esp32-h2-devkitm-1/user_guide.html
[121] https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers.html
[122] https://www.st.com/en/microcontrollers-microprocessors/stm32wl-series.html
[123] https://www.st.com/en/microcontrollers-microprocessors/stm32wb0-series.html
[124] https://www.st.com/en/microcontrollers-microprocessors/stm32wb-series.html
[125] https://www.st.com/en/microcontrollers-microprocessors/stm32wba-series.html
[126] https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus/products.html
[127] https://www.st.com/en/microcontrollers-microprocessors/stm32-mainstream-mcus/products.html
[128] https://www.st.com/en/microcontrollers-microprocessors/stm32-ultra-low-power-mcus/products.html
[129] https://www.st.com/en/microcontrollers-microprocessors/stm32-wireless-mcus/products.html
[130] https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md
[131] https://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/Raspberry-Pi-Zero-V1.3-Schematics.pdf
[132] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#raspberry-pi-zero-w
[133] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#raspberry-pi-zero-2-w
[134] https://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/Raspberry-Pi-A-Plus-V1.1-Schematics.pdf
[135] https://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/Raspberry-Pi-B-Plus-V1.2-Schematics.pdf
[136] https://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/Raspberry-Pi-2B-V1.2-Schematics.pdf
[137] https://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/Raspberry-Pi-3B-V1.2-Schematics.pdf
[138] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#raspberry-pi-4-model-b
[139] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#raspberry-pi-4-model-b
[140] https://www.raspberrypi.org/documentation/hardware/camera/README.md
[141] https://www.raspberrypi.org/documentation/hardware/computemodule/cmio-camera.md
[142] https://www.raspberrypi.org/documentation/hardware/raspberrypi/dpi/README.md

[143] https://www.raspberrypi.org/documentation/hardware/computemodule/cmio-display.md

[144] https://github.com/ElectronicCats/mpu6050/tree/master

[145] http://www.electronics-tutorials.ws/io/io_3.html

[146] https://www.engineersgarage.com/articles/humidity-sensor

[147] http://www.circuitbasics.com/how-to-set-up-the-dht11-humidity-sensor-on-an-arduino/

[148] http://wiki.seeedstudio.com/Grove-GPS/

[149] https://learn.adafruit.com/adafruit-arduino-lesson-16-stepper-motors/breadboard-layout

[150] Kuaban, G. Suila, E. Gelenbe, T. Czachórski, P. Czekalski, and J. Kewir Tangka, “Modelling of the Energy Depletion Process and Battery Depletion Attacks for Battery-Powered Internet of Things (IoT) Devices”, sensors, vol. 23, issue 6183, 2023

[151] Kuaban, G. Suila, T. Czachórski, E. Gelenbe, P. Czekalski, “A Markov model for a Self-Powered Green IoT Device with State-Dependent Energy Consumption”, 2023 4th International Conference on Communications, Information, Electronic and Energy Systems (CIEES) 23 25 November, 2023, Plovdiv, Bulgaria, IEEE, 2023 (in press).

[152] Fredrik Häggström and Jerker Delsing, “IoT Energy Storage – A Forecast”, Energy Harvesting and Systems 2018; 5(3-4)

[153] https://www.geothermal-energy.org/pdf/IGAstandard/EGC/2013/EGC2013_CUR-16.pdf

[154] 11 Internet of Things (IoT) Protocols You Need to Know About, DesignSpark, https://www.rs-online.com/designspark/eleven-internet-of-things-iot-protocols-you-need-to-know-about

[155] http://www.ieee802.org/15/

[156] https://en.wikipedia.org/wiki/Network_address_translation

[157] https://support.microsoft.com/en-gb/help/103884/the-osi-model-s-seven-layers-defined-and-functions-explained

[158] https://earthobservatory.nasa.gov/Features/OrbitsCatalog/

[159] http://web.mit.edu/modiano/www/6.263/lec22-23.pdf

[160] RFC 1631: http://www.faqs.org/rfcs/rfc1631.html

[161] https://en.wikipedia.org/wiki/ANT%2B

[162] http://www.zigbee.org/

[163] https://nodered.org/

[164] http://www.restapitutorial.com/lessons/whatisrest.html

[165] https://www.w3.org/TR/soap/

[166] https://www.w3schools.com/tags/ref_httpmethods.asp

[167] https://en.wikipedia.org/wiki/Power_over_Ethernet

[168] https://www.techworld.com/data/what-is-li-fi-everything-you-need-know-3632764/

[169] https://www.bluetooth.com/learn-about-bluetooth/tech-overview/

[170] https://www.bluetooth.com/learn-about-bluetooth/feature-enhancements/mesh/

[171] https://www.threadgroup.org/Portals/0/documents/support/ThreadOverview_633_2.pdf

[172] https://www.sigfox.com/en

[173] https://www.thethingsnetwork.org/airtime-calculator

[174] https://www.lora-alliance.org/
[175] https://en.wikipedia.org/wiki/IPv4
[176] https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/
[177] Jonas Olsson, “6LoWPAN demystified”, 2014, Texas Instruments
[178] https://www.hivemq.com/mqtt/
[179] https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920/
[180] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
[181] https://www.espressif.com/en/support/download/other-tools
[182] https://github.com/nodemcu/nodemcu-flasher
[183] https://github.com/binaryupdates/esp01-firmware
[184] https://github.com/tasmota/tasmotizer
[185] https://bbs.espressif.com/viewforum.php?f=46
[186] http://www.electrodragon.com/w/ESP8266_AT-Command_firmware
[187] https://github.com/martin-ger/esp_wifi_repeater
[188] https://flask.palletsprojects.com
[189] https://www.raspberrypi.com/news/how-to-run-a-webserver-on-raspberry-pi-pico-w/
[190] https://www.domoticz.com/
[191] https://www.openhab.org/
[192] https://www.home-assistant.io/
[193] https://tasmota.github.io/docs/
[194] https://esphome.io/index.html
[195] https://espeasy.readthedocs.io/en/latest/
[196] https://github.com/xoseperez/espurna
[197] https://github.com/openshwprojects/OpenBK7231T_App
[198] https://nodered.org
[199] https://www.coursera.org/
[200] https://www.edx.org/
[201] https://www.udacity.com
[202] https://www.udemy.com/
[203] https://www.skillshare.com/
[204] https://www.electronics-tutorials.ws/
[205] https://embeddedexpert.io
[206] https://www.instructables.com
[207] https://wokwi.com/
[208] https://iot-open.eu

ITT GROUP
INNOVATSIOON - TEADUS - TEHNIKA

ITMO UNIVERSITY

1862
RIGA TECHNICAL
UNIVERSITY

STUDIORUM UNIVERSITAS
MESSANAE

TAL
TECH