Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT INDEX

Internet of Things - Intermediate Module

Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT INDEX

About This Course

The Internet of Things (IoT) course covers information about internet protocol suites, different type of connections, application layer protocols, services and use of microcontroller units. Learners will acquire knowledge on the use, application and commercialisation of concepts pertaining to the broad context of IoT.

Format

The form of education is e-learning with aprrox. 10 hours of lessons, 10 hours of self-studying and 20 hours of practical activities. Weekly lessons include lectures, thematic videos and performing test tasks. An important part of this course is performing final exam in the form of multiple choices quiz, which contains answers based on study material. The course is set up in compliance with the ECVET System with possibility to obtain the Certificate of attendance.

Who can take this course?

To enroll in this course you need to have successfully completed 'INDEX: Industrial Expert - Basic Module'.

The course has no specific prerequisites, but basic knowledge of Information Technology is beneficial.

It will be of particular interest to:

- Senior executives or development department managers of your enterprise interested in learning about the Internet of Things;
- Professionals interested in the applications of the Internet of Things in their areas of expertise;
- Founders of high-tech startups;
- Young engineers of your company, who are already working on the development of specific solutions of the Internet of Things;
- Educators teaching graduate and postgraduate courses focusing on the Internet of Things;
- Students or postgraduates interested in the Internet of Things.

Programme of the course

Internet protocol suite

- OSI model
- OSI model: Comparison with TCP/IP model

- TCP/IP model

Connections

- Introduction
- Wired solutions
- Short-distance wireless solutions
- Medium-distance wireless solutions
- Long-distance wireless solutions

Application layer protocols

- Introduction
- HTTP and CoAP
- Common formats found with HTTP
- MQTT
- Comparison of HTTP-REST and MQTT

Services

- Introduction
- Functions
- Providers and examples

Hands on with microcontroller units

- Microcontroller Units
- Arduino
- Programming Arduino for basic tasks
- Using Arduino in IoT projects

Course staff

Dr Fulvio Ratto

Fulvio received his M.Sc. degree in Physics from University of Trieste (Italy) in 2002 and his Ph.D. degree in Energy and Materials Science from University of Quebec (Canada) in 2007. His current interests lie at the crossroads of plasmonics and biomedical optics, and in particular the integration of

noble-metal components in bionic systems and photonic devices. **Current position:** Researcher at Consiglio Nazionale delle Ricerche - Istituto di Fisica Applicata (Italy). **Course sections:** Internet protocol suite, Application layer protocols, Services, Hands on with microcontroller units

Dr Lucia Cavigli
Lucia received her M.Sc. degree in Physics from University of Florence (Italy) in 2002 and her Ph.D. degree in Materials Science at Department of Physics of University of Florence (Italy) in 2006. Her skills are in the field of photonics and nanostructured materials. **Current position:** Researcher at Consiglio Nazionale delle Ricerche - Istituto di Fisica Applicata (Italy). **Course sections:** Connections

Dr Filippo Micheletti
Filippo received his M.Sc. degree in Electronic Engineering from University of Florence in 2011 and his Ph.D. in Information Engineering and Mathematical Sciences from University of Siena in 2016. His skills are in the development of photonic technologies for applications in medicine and biology. **Current position:** Researcher at Consiglio Nazionale delle Ricerche - Istituto di Fisica Applicata (Italy). **Course sections:** Services

Dr Ilias Kalfas
Ilias is an Agronomist with a specialty in Plant Protection, holding a degree from Faculty of Agriculture of Aristotle University of Thessaloniki. He holds a Ph.D. from University of London, Diploma of Imperial College and M.Sc. in Production Organization and Quality Management in the Food Industry. **Current position:** Project Leader regarding plant production projects of Strategic Programs Management Office of the American Farm School (Greece). **Course sections:** Connections - Medium-distance wireless solutions

Dr Nikos Tsotsolas
Nikos holds a Diploma in Production Engineering and Management from Technical University of Crete, Greece (1997), a M.Sc. in Operational Research (2001) from the same University and a Ph.D. in Statistical Science (2009) from University of Piraeus. From 2006 to date, he is member of the Board of the Hellenic Operational Research Society (HELORS). His research interests fall into the areas of multi-criteria analysis, decision support systems, service quality, and post-optimality analysis in mathematical programming. **Current position:** co-founder and R&D Consultant of Green Project SA (Greece).
Course sections: Connections - Medium-distance wireless solutions

Results

As the result of completing the Internet of Things- intermediate level course, learners will know:

- The key aspects of communication protocols and abstraction layers and the key elements of OSI and TCP/IP models
- The key features of the principal kinds of connections in communication technology
- The key features of the principal application-layer protocols found in communication technology

- The key features of the principal kinds of online services
- The key features of microcontroller units and basic elements of C programming language

Competences

By completing the Internet of things intermediate level course, learners will be able:

- to identify the key features of internet protocol suite
- to make a decision on the best kinds of transports for a certain IoT application
- to make a decision on the best kinds of application layer protocols for a certain IoT application
- to make a decision on the best kinds of online services for a certain IoT application

Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT

Internet of Things - Intermediate Module

Internet protocol suite

OSI model

Introduction

Communication is the most distinctive and critical part of the Internet of Things. We begin our journey through this exciting topic with a rather theoretical description of the various components needed to establish communication across a network of things, humans, or both. In the next sessions, we will essentially come back and analyse these components one by one from a much more practical perspective. But here, we focus on their hierarchy and mutual interactions, which will set the ideal background to make our journey seamless.

Quiz

Imagine that someone lands on Mars and, much to their surprise, finds a crowd of aliens that, incidentally, look pretty different from earthlings.
What sequence of steps could lead to the commencement of communication, in your opinion?

☐ Speak slowly and loudly.

☐ Make great sweeping gestures.

☐ First establish agreed-upon standards and protocols in a stack, i.e. how to address and make sure to deliver messages to destination first, then what means to use, such as sound or visuals or contact, etc., how to arrange a conversation, how to create a dictionary, etc.

☐ First establish agreed-upon standards and protocols in a stack, i.e. what means to use first, such as sound or visuals or contact, etc., then how to address and make sure to deliver messages to destination, how to arrange a conversation, how to create a dictionary, etc.

Standards and protocols are a fundamental issue in communication systems, as is the use of script and language in human conversation. The **Open Systems Interconnection (OSI)** model is a result of the Open Systems Interconnection project of the International Organization for Standardization (ISO). Its formulation establishes a theoretical construction that separates the complex set of functions needed in communication systems into different layers, with the intent of providing a consistent framework for the development of specific protocols and fostering their interoperability. So, for instance, in human conversation, script and language may be different layers, say layer 1 and layer 2. Print and cursive alphabets would be alternative examples of layer 1 protocols. English or Italian idioms would both serve as layer 2 protocols. In principle, any combination of layer 1 and layer 2 protocols would be feasible. However, this simple example already suggests limitations in the extent of interoperability. It may be difficult to use hiragana symbols to transcribe a Russian poem.

The OSI model consists of 7 layers, where layer n provides services to layer n+1 and rests on those of layer n-1. For instance, if a layer n protocol was about the identification of a path in a network, layer n+1 protocol may include the transportation of a packet of information through that network, and layer n-1 protocol may ensure the connection from single node to node. Two entities of the same layer N on both ends of the communication link are called layer N peers and exchange so-called protocol data units (PDUs). Each layer N PDU is made of a payload called service data unit (SDU) plus headers or footers that provide instructions about the layer N protocol in use. In practice, communication begins with a PDU composed at layer 7. Such layer 7 PDU is passed to layer 6, where it is treated as a layer 6 SDU and concatenated with headers and footers to make a layer 6 PDU, etc. On the other side of the link, the layer 1 PDU is stripped of its headers and footers to obtain a layer 1 SDU. Such layer 1 SDU is passed to layer 2, where it is processed as a layer 2 PDU, etc.

The OSI model was devised in the late 70s and early 80s as a response to an emerging need for interoperability among a variety of communication protocols that were under construction in those days. It was conceived more as a set of guidelines for future protocols rather than the standardization of a suite of best practices in use, and has probably never really come into effect. However, it still represents a useful reference to understand the hierarchy of issues in communication protocols.

Among the principal problems with the OSI model, we mention some reservations on the general concept for strict layering, and the need to accommodate inter-layer and sublayer functions, such as those associated with management and security. For instance, in wireless communication, the Physical Layer and the Data Link Layer outlined in the next two units need to frequently exchange information in order to handle flaky connections and to avoid wasting overall resources.

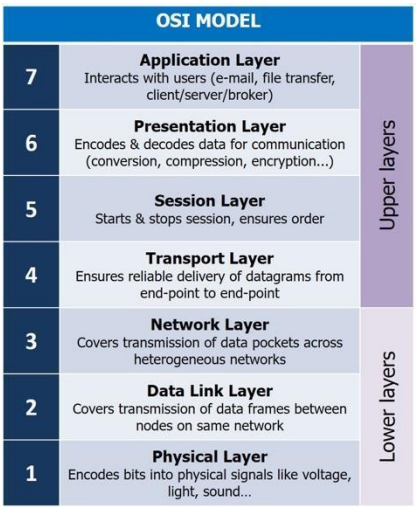


Figure: overview over OSI model, CC BY-SA by INDEX consortium

INDUSTRIAL EXPERT INDEX

Layer 1: Physical Layer

Imagine a group of children playing whisper game. Maybe it goes without saying, but their first decision will be on what physical means to use to exchange messages, such as sounds emitted with voice and received with hearing. But it may also be visuals emitted with fingers and received with sight, as in sign language. Or handwritten notes, etc.

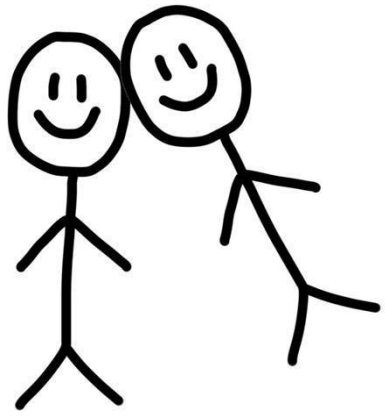


Figure: Analogy to Layer 1 of OSI model CC BY-SA by INDEX consortium

Layer 1 of the OSI model is the so-called Physical Layer, which practically receives raw sequences of logical bits from Layer 2 and takes care of their physical encoding into voltage, light, or sound, etc. and transmission through a medium as a coaxial cable, optical fiber, or radio wave, etc., as well as its decoding the other way around. It answers the question how to encode and decode logical bits into and from a physical signal.

The features defined in a Physical Layer protocol include the physical parameters of the transmission medium, such as shape, size and number of pins of a connector and its mechanical specifications; their functional features, such as the meaning of each pin in simplex, half duplex or full duplex communication; the electrical, optical, or acoustical, etc. translation of the logical levels into a so-called line code, which includes voltage, intensity, duration, bit rate, etc. of a physical signal; coupling of this signal to a transmission medium, signal processing, error control over unreliable or noisy channels, etc.

In particular, the transmission media may be electrical conductors as twisted pairs and coaxial cables; optical systems as multimode or single-mode fibers or air; wireless systems as radio or microwaves ; or others, such as acoustic systems or many more. Important features of this media include their attenuation, which is the intensity of signal lost per unit of distance travelled, their susceptibility to signal degradation due to noise or interference or to signal distortion; their bandwidth or maximum

bit rate; their cost; need for maintenance, etc. The same medium may host different bands, and, in this case, its features may depend from band to band. This is the case, for instance for radio waves.

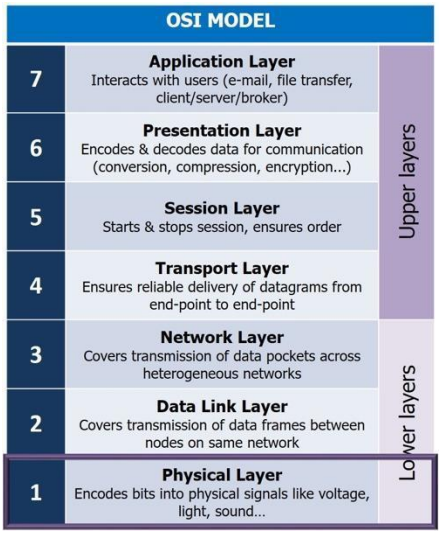


Figure: position of Layer 1 in OSI model CC BY-SA by by INDEX consortium

Layer 2: Data Link Layer

The second question in front of our children playing the whisper game is how to exchange messages from one child to the next. Maybe by whispering to each other. But how to identify the right neighbour? And how to manage in the event of more children wishing to address the same player at the same time?

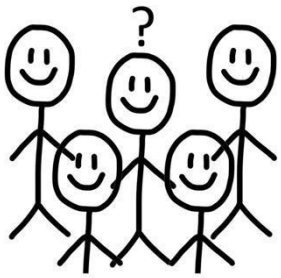


Figure: analogy to Layer 2 of OSI model CC BY-SA by INDEX consortium

Layer 2 of the OSI model is the so-called The Data Link Layer, which receives data packets from Layer 3 and takes care of their encapsulation into so-called frames, which are short sequences of bits, and node-to-node transfer within the same network segment over the Physical Layer. It answers the question of how to manage communication between two nodes on the same network level, in terms

of medium access, local addressing, flow control and sometimes low-level error checking and retransmission.

The Data Link Layer is actually subdivided into two sub-layers that really interact as separate layers, in the sense that any one protocol in the first sublayer may potentially interoperate with any other protocol in the second sublayer.

The lower sublayer is the Medium Access Control (MAC) sublayer. Its main purposes are to regulate the access of multiple nodes to a shared communication channel by minimizing or managing the occurrence of collisions and loss of data resulting from a simultaneous occupation of the same medium; and to implement operations for physical addressing. There are two main types of protocols for multiple access control based on distributed or centralized algorithms. In distributed algorithms, collisions may occur but appropriate mechanisms are in place to reduce their probability and retransmit collided frames. For instance, it may be decided that, before transmitting, each node waits a random delay after the shared medium has become free, so that it would take an accidental same hold for collisions to occur. In centralized algorithms, collisions may be hampered by strict scheduling according to rules established during network initialization. At this level, the format of the frame is encapsulated with headers and footers containing instructions as the physical addresses of sender and recipient.

The higher sublayer is the Logical Link Control (LLC) sublayer, which covers functions as asynchronous or synchronous encoding in serial communication, flow control and error control. In serial communication, asynchronous transmission requires start and stop signals to delimit each character, whereas synchronous transmission functions by frames and entails the addition of a header for synchronization. Flow control serves to harmonize the bit rate for a faster sender to avoid overwhelming a slower recipient and requires some feedback mechanism to be in place. Error control may be implemented with different quality at this level and cooperate with similar functions over more layers. The simplest control is parity as it may be implemented by the addition of a bit representing whether the number of ones or zeros in the frame was even or odd. Of course, parity may easily fail in the presence of multiple errors per frame over a poor connection. A checksum is a more robust solution to encode the overall number of ones or zeros. Finally, the request for acknowledgements is the safest solution to check the reception and integrity of entire frames. In the case of errors, frames may be just abandoned or discarded and resent according to the protocol in use, and whether to prioritize reliability or speed.

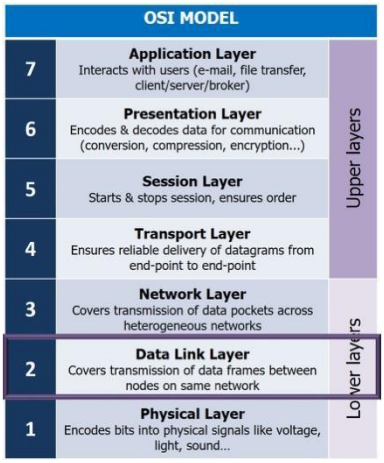


Figure: position of Layer 2 in OSI model CC BY-SA by INDEX consortium

Layer 3: Network Layer

The third question in front of our children playing the whisper game is how to identify the best path for a message to travel from origin to destination across multiple boundaries. Just so. They have decided to scatter across a large field instead of forming a line! And so, there may be multiple options. Some choices may be better in terms of length, others in terms of congestion.

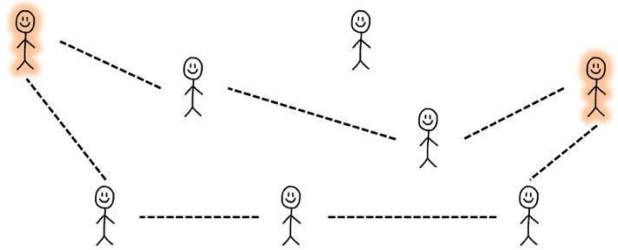


Figure: analogy to Layer 3 of OSI model CC BY-SA by INDEX consortium

Layer 3 of the OSI model is the so-called Network Layer, and serves to route and forward data packets from Layer 4 across different networks by using the services provided by the Data Link Layer. It answers the question how to identify the best pathway and manage the transmission of data across heterogeneous networks.

Protocols in the Network Layer deal with different functions. Routing means determining the best route for data transmission over the network from the sender to the recipient address. Every host in the network is denoted by a unique address that determines its location and is normally assigned from a hierarchical system. In most cases, routing is dynamic and depends on parameters as network conditions, the use of routing tables and service priorities. Some protocols require that a fixed and

dedicated communication channel be in place before two hosts can start exchanging data according to higher layer functions; other protocols instead simply deliver datagrams to destination without prior connection nor guarantee, and most transport ends at this layer. Functions sometimes present are congestion control or quality of service guarantee by reserving resources over all nodes along the pathway. Forwarding means receiving a packet on one port, storing it and retransmitting it on another port. This function is present in all nodes and may entail the combination of different Data Link Layer protocols as well as packet splitting and reassembling, when received packets are larger than the Maximum Transmission Unit available across a node. In addition, in geographic networks (WAN or MAN), pricing can be managed and calculated on the basis of connection time or other parameters.

The Network Layer is the last level present in network switches or internal nodes, while the upper architectural levels are present only in terminal nodes.

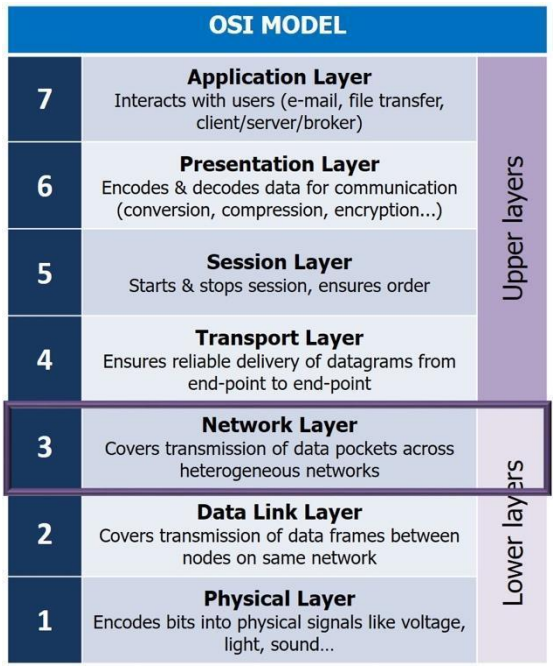


Figure: position of Layer 3 in OSI model CC BY-SA by INDEX consortium

Layer 4: Transport Layer

The fourth question in front of our children playing the whisper game is how to make sure that messages received at destination correspond to those sent from origin. This may be the funniest part of the story! The sender and the recipient of a message do not care about its exact path and all underlying details, at this level. They may just wish to find a way to verify the integrity of their communication.

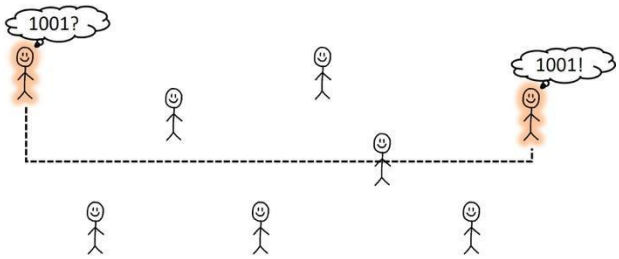


Figure: analogy to Layer 4 of OSI model CC BY-SA by INDEX consortium

Layer 4 of the OSI model is the so-called Transport Layer, which receives and segments messages from Layer 5 and provides for their reliable transmission from end-point to end-point over the Network Layer, according to different classes of services. It is the first layer that solely involves the end-points on both sides of the communication channel.

The Transport Layer may offer multiple services, but none is mandatory and each protocol may provide a different combination of features. For connection-oriented services, such as those entailing the exchange of feedback and acknowledgments, it takes care of creating a persistent connection, which it then closes as soon as no longer needed. It verifies and ensures correct reordering of data segments by the recipient, which may follow different pathways over the Network Layer. It provides for reliable transfer by making sure that all segments sent by the sender are received by the recipient and provides for their retransmission in the form of corrupt files as needed. It provides for flow control on request of the receiver and for congestion control when the sender identifies a problem in the network, by optimizing the data rate. In some cases, it may impart a byte orientation, in order to streamline the communication as a stream of bytes rather than segments. Last but not least, it may dictate multiplexing by establishing multiple connections between the same two hosts through the use of virtual ports.

Protocols in this layer are divided into five classes numbered from 0 to 4 according to the functions implemented in their provisions. Class 0 provides the least features and is suited for the most reliable networks. Class 1 ensures basic error recovery. Class 2 adds multiplexing. Class 3 features full error recovery and multiplexing. Class 4 adds error detection, and may be recommended for the most unreliable networks, such as the Internet.

The name of this layer may be misleading, because it does not actually implement any physical or logical transportation of data, which are covered by lower architectural levels, but rather makes up for their lack of reliability and closes the loop, by implementing functions that ensure the overall quality of service from end-point to end-point.

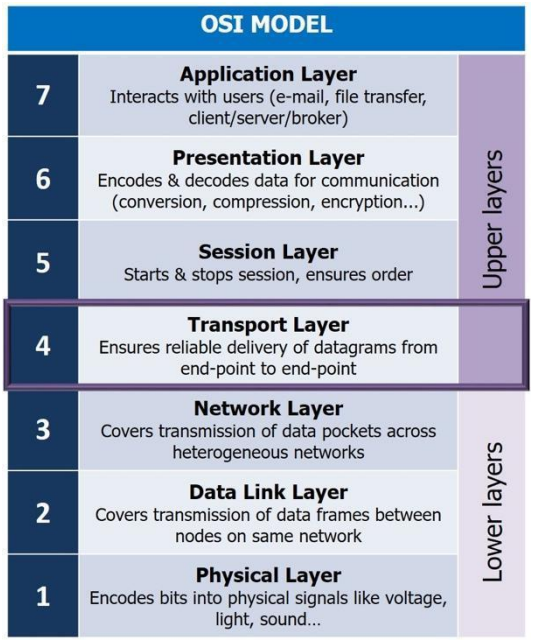


Figure: Position of Layer 4 in OSI model CC BY-SA by INDEX consortium

Layer 5: Session Layer

The fifth question in front of our children playing the whisper game is how to establish a communication channel that may support a full conversation made of multiple messages sent back and forth. Maybe without the need to verify the identity of the sender and the recipient each time, which may consume most of their playtime with boring procedures!

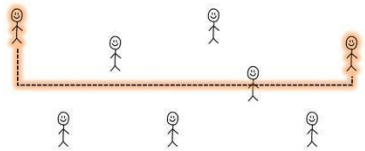


Figure: analogy to Layer 5 of OSI model CC BY-SA by INDEX consortium

Layer 5 of the OSI model is the so-called Session Layer, which offers services to Layer 6 by setting the rules, establishing, managing and terminating a dialogue between two hosts over the Transport Layer. This layer transcends the notion of a sender and a recipient and connects two hosts in a semi-permanent communication channel, which may be simplex, half-duplex, or full-duplex.

The Session Layer establishes procedures for starting, checkpointing, restarting, suspending and terminating a session with a graceful close, for synchronizing activities on both sides and for negotiating the quality of service implemented during the dialogue over the Transport Layer.

Functions such as authentication and authorization belong to this level. So-called sync points are used for resetting and recovering a session in the case of problems. The Session Layer plays an especially critical role in application environments that use remote procedure calls in the context of distributed computing.

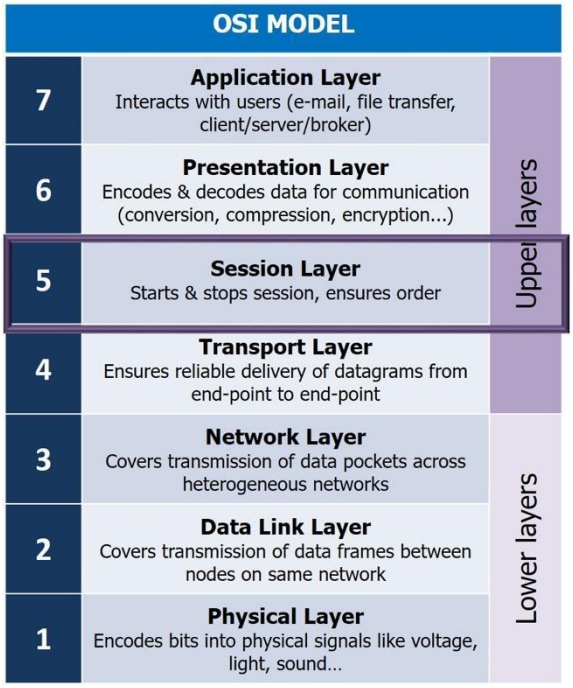


Figure: position of Layer 5 in OSI model CC BY-SA by INDEX consortium

Layer 6: Presentation Layer

The sixth question in front of our children playing the whisper game is how to encode their messages. If they were talking animals, for instance, it may be funny to zip their messages as noises, which nosy adults may be unable to understand! Or maybe they may want to play draughts, and just need standards to understand each other on what pieces to move and where across the chessboard.

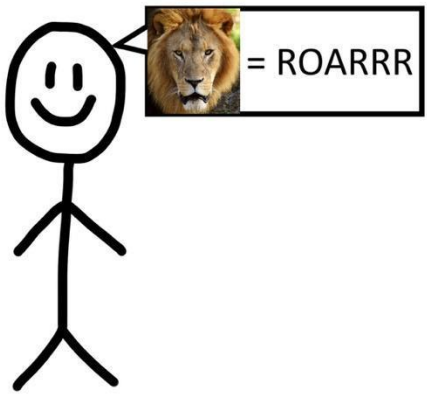


Figure: analogy to Layer 6 of OSI model CC BY-SA by INDEX consortium

Layer 6 of the OSI model is the so-called Presentation Layer, and provides Layer 7 with the facility to encode and decode data in a format that may be suitable for communication through the Session Layer. The translation may include multiple steps for serialization of complex data structures into flat byte-strings, compression and cryptographic encryption as well as the reverse operations on the other side of the communication channel.

The Presentation Layer allows the user applications on the two ends of the communication channel to use different semantics and syntax without worries, for instance one applying EBCDIC and the other using ASCII to represent the same characters, by providing the appropriate conversion between multiple formats. Although it may also belong to the Data Link and Physical Layers, security by cryptographic encryption is done at this level, in order to avoid processing all headers and footers concatenated at lower layers, when the payload coming from the user applications may be the only sensitive information, and so to save computational resources on reception. However, in real implementations of network architectures, where it may be desirable to protect more data, security may be implemented at multiple levels by encrypting relevant overhead information. In this case, the initial payload may even undergo multiple consecutive codifications. The same applies to data compression. It is often useful to compress the payload coming from the user applications, but it may be harmful to zip the overhead information added at lower levels, because a lossy compression may induce a fatal loss of integrity and jeopardize the entire communication flow. Also, in this context, compression done at this level imposes less computational burden on reception.

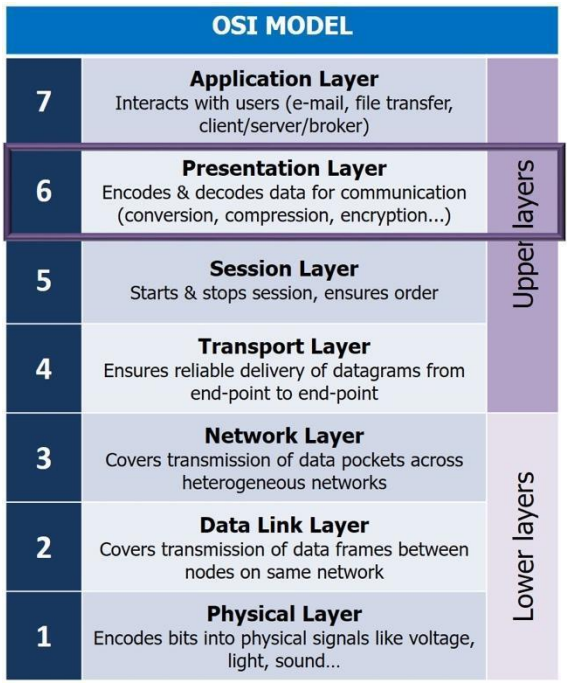


Figure: position of Layer 6 in OSI model CC BY-SA by INDEX consortium

Layer 7: Application Layer

The final question in front of our children playing the whisper game is how to structure their communication at the highest level. For instance, they may decide that some children are there to ask questions, and others to look them up in a textbook and reply as soon as they find an answer. Great for a written exam! Or that some children are collecting messages on different topics, which they are forwarding to anybody who has shown interest therein, etc. They are not yet considering what to talk about at this level, but just creating a functional infrastructure to support a variety of games. And now the fun is ready to start!

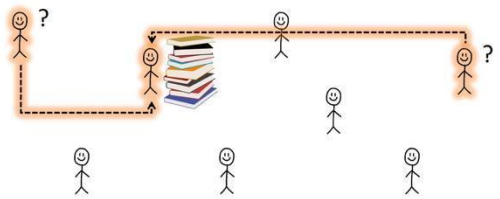


Figure: analogy to Layer 7 of OSI model CC BY-SA by INDEX consortium

Layer 7 and last of the OSI model is the so-called Application Layer, and is the one closest to the end user and its software interface. It exploits services provided by the Presentation Layer to implement specific behaviours in software applications that involve certain communication tasks.

It is still not the final user application, like e.g. a web browser or a Mail User Agent, but the set of underlying tools needed to perform fundamental and common communication tasks, such as identifying remote resources and communication partners, transferring files, etc. The final user application goes beyond the scope of this layer and the entire stack of OSI protocols, which stop at the creation of the infrastructure needed for general communication.

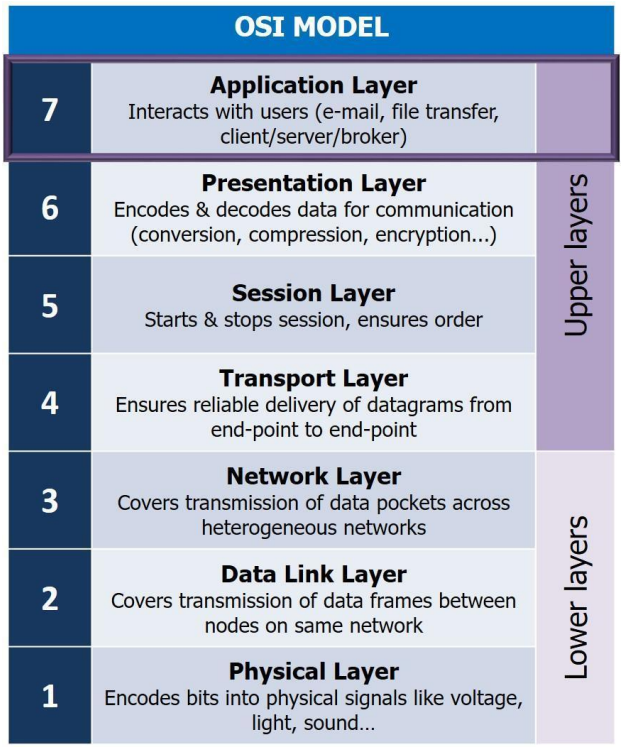


Figure: position of Layer 7 in OSI model CC BY-SA by INDEX consortium

Exercise

Which of the following statements makes sense?

☐ In an IoT network, the protocols in the various layers of the OSI model had better be chosen in a consistent manner.
☐ In an IoT network, like any other network, the protocols in the various layers of the OSI model are independent of each other, from a practical perspective.
☐ In an IoT network, there may be no Application Layer nor Presentation Layer, because there may be no human interface at all.
☐ In an IoT network, the lower layers of OSI model may include sensors and actuators for cyber-physical integration.

INDUSTRIAL EXPERT INDEX

OSI model vs TCP/IP model

The Internet Protocol Suite is often named after its most representative protocols as the TCP/IP model, and is a framework alternative to the OSI model that represents the standard de facto for the Internet. With respect to the OSI model, it places much less emphasis on layering and the encapsulation of functions in a hierarchical stack, and takes a more pragmatic and lightweight approach to the establishment of a global communication network. However, its protocols may still broadly be assigned to four levels that map rather well to those of the OSI model both in terms of inherent scopes and mutual interactions.

The TCP/IP Link Layer corresponds to the OSI Data Link Layer and may also include functions of its Physical Layer, although the underlying hardware implementation is more assumed than prescribed, and also some components at the interface with its Network Layer.

The TCP/IP Internet Layer covers the largest part of the OSI Network Layer.

The TCP/IP Transport Layer corresponds to the OSI Transport Layer plus some functions of its Session Layer as the so-called graceful close.

Finally, the TCP/IP Application Layer maps to the OSI Application Layer, Presentation Layer and most of Session Layer, although formatting and presenting data is more left to libraries and application programming interfaces than specified in protocols within the TCP/IP stack.

<table>
<tr><th>OSI MODEL</th><th>TCP/IP MODEL</th></tr>
<tr><td>Application Layer
Interacts with users (e-mail, file transfer, client/server/broker)</td><td rowspan="3">Application Layer</td></tr>
<tr><td>Presentation Layer
Encodes & decodes data for communication (conversion, compression, encryption...)</td></tr>
<tr><td>Session Layer
Starts & stops session, ensures order</td></tr>
<tr><td>Transport Layer
Ensures reliable delivery of datagrams from end-point to end-point</td><td>Transport Layer</td></tr>
<tr><td>Network Layer
Covers transmission of data pockets across heterogeneous networks</td><td>Internet Layer</td></tr>
<tr><td>Data Link Layer
Covers transmission of data frames between nodes on same network</td><td rowspan="2">Link Layer</td></tr>
<tr><td>Physical Layer
Encodes bits into physical signals like voltage, light, sound...</td></tr>
</table>

Figure: comparison between OSI and TCP/IP models

INDUSTRIAL EXPERT INDEX

Link Layer

The Link Layer is the first level in the TCP/IP model and deals with local network connections among all hosts accessible before traversing routers. Its functions include framing, transmitting and receiving data packets over a physical medium. It specifies procedures to identify and communicate to specific hosts through Medium Access Control (MAC) addresses, and operates through the drivers or the firmware of dedicated network cards or chipsets. However, the presence of an underlying physical level is assumed as an implicit provision, and so are its tangible parameters, such as distance, bandwidth, etc.

A representative example of protocol within this layer is the Neighbor Discovery Protocol (NDP), which is specifically geared at providing services to the Internet Protocol version 6, such as identifying hosts and routers on the same local network, retrieving their parameters, such as their Maximum Transmission Unit (MTU), and mapping their physical addresses to their network addresses.

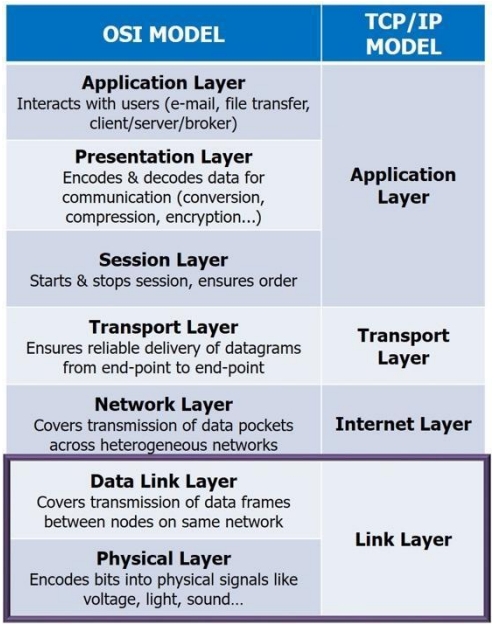


Figure: position of Link Layer in TCP/IP model CC BY-SA by INDEX consortium

Watch the following video by RIPE CNN for a short description of NDP.

Neighbor Discovery Protocol

Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before use.

Internet Layer

The Internet Layer is the second level in the TCP/IP model and deals with the process of routing datagrams across potentially different networks. It defines an approach to identify and address hosts and to unreliably forward datagrams from source until destination from router to router by using a hierarchical system of addresses.

The most important protocol in this layer is the Internet Protocol (IP), which is responsible for encapsulating data into datagrams and addressing host interfaces, in order to route packets from source to destination across multiple networks. Datagrams are made of an IP header and a payload. The IP header includes the IP addresses of source and destination, and other metadata needed for delivery. IP addressing entails the assignment of IP addresses and other parameters to host interfaces. The address space is subdivided into subnetworks by the use of prefixes. All hosts and routers collaborate toward the process of IP routing. In particular, routers are responsible for transporting packets across network boundaries, and communicate via dedicated procedures according to network topology. The first really popular version of the IP protocol was number 4, which reserves 32 bits for addressing, thus accommodating a maximum of around 4.3×10^9 hosts. In order to overcome this limitation, which has become stringent over the years, beginning from 2006, IP protocol version number 6 has started to complement and replace its predecessor. With an 128-bit address space, this

version provides for as many as around 3.4×10^{38} IP addresses and is fit to support current trends as the Internet of Things.

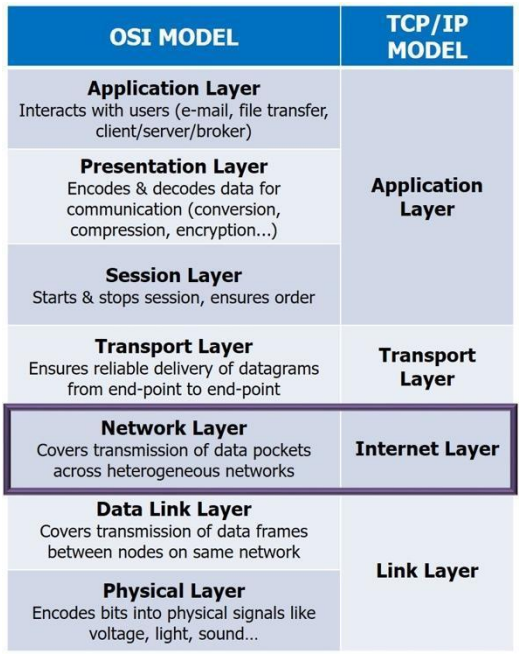
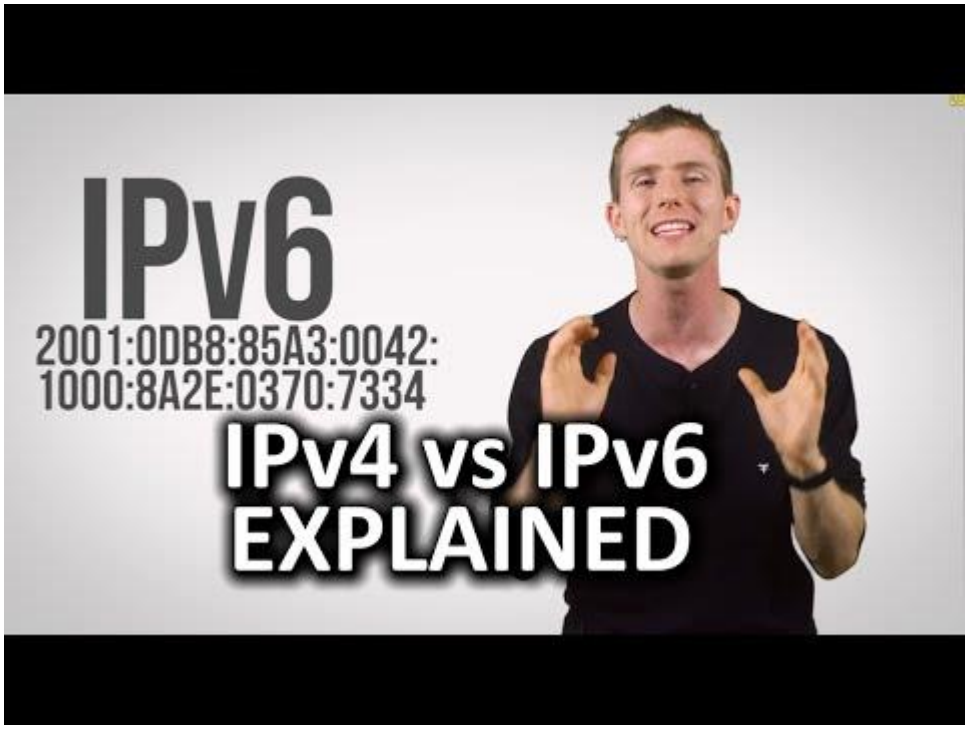


Figure: position of Internet Layer in TCP/IP model CC BY-SA by INDEX consortium

Watch the following video by TechQuickie on the main differences between IPv4 and IPv6, and main advantages and possible barriers met in the relevant transition.

Internet Protocol - IPv4 vs IPv6 as Fast As Possible



Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before use.

Transport Layer

The Transport Layer is the third level in the TCP/IP model and establishes the procedures for host-to-host connectivity independent of both the overlying application design as well as the underlying network architecture. Functions that may optionally belong to this level include data segmentation, flow control, congestion control, application addressing via virtual ports, and error control. The use of virtual ports is a convenient tool to create communication channels dedicated to certain processes or applications. For many applications port numbers have been standardized, so that, for instance, clients requesting certain services may directly inquire servers through the appropriate ports without additional mediation.

Protocols at this level may be connection-oriented, as the Transmission Control Protocol (TCP), or connectionless, as the User Datagram Protocol (UDP).

The TCP protocol ensures high reliability at the expense of lengthy latency, and compensates for the lack of dependability of the IP protocol as a best-effort system. It solely concerns the terminal hosts and runs at the level of their operating systems. Its main features are that it is connection-oriented and includes the creation, maintenance and eventual closure of a semi-permanent communication channel between both hosts; that it guarantees the delivery of data segments by implementing

acknowledgments; that it supports a full-duplex flow of byte; that it creates and addresses data segments to ports dedicated to specific applications, such as port 80 for the Hypertext Transfer Protocol (HTTP); that it guarantees the receipt of data segments in the right order and only once, through acknowledgments and timeout retransmissions; that it offers error checking through a checksum field encoded in its Protocol Data Unit; that it features flow control between both terminals and congestion control over the network; that it supports multiplexing through multiple ports. TCP is often implemented in combination with applications where reliability is a priority, such as the File Transfer Protocol (FTP), the Simple Mail Transfer Protocol (SMTP) or HTTP.

In a sense, the UDP protocol is a perfect counterpart to TCP for applications where minimal latency is a priority and the loss of data segments may be tolerable, such as in real-time audio and video streaming, or in cases when the addition of lengthy headers would be disproportionate. It is transaction-oriented for simple query-response protocols, and stateless, meaning that, for instance, servers retain no information on their clients, which helps streaming media to support a large number of guests. Error control is available as a simple checksum, but feedback and retransmission are totally absent. The lack of retransmission and associated delays is also perfect for real-time applications such as Voice over IP, online games, and many others using the Real Time Streaming Protocol (RTSP).

Of course, there are many more protocols in this layer, but TCP and UDP cover the vast majority of uses over the Internet.

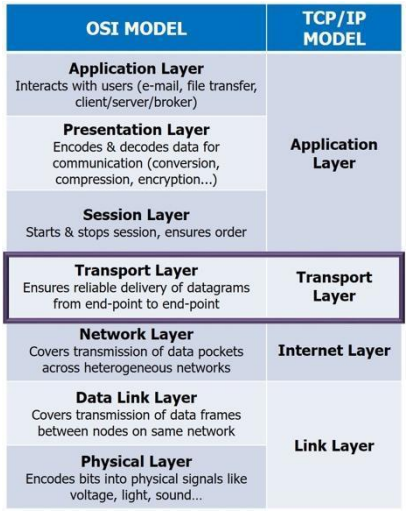


Figure: position of Transport Layer in TCP/IP model CC BY-SA by INDEX consortium

Watch the following video by Cisco for an overview over the main differences and applications of TCP and UDP.

Application Layer

The Application Layer is the fourth and topmost level in the TCP/IP stack and provides high-level functionalities for implementation in software applications by assuming the existence of underlying channels of communication. At this level, the TCP/IP model makes a distinction between support protocols and user protocols, where support protocols maintain the network infrastructure, such as the Domain Name System (DNS) for assigning host names and their resolution into IP addresses, and user protocols define actual end-user applications. There are many user protocols tailored to different purposes. Common examples are the File Transfer Protocol (FTP) to exchange data files in a client-server architecture; the Simple Mail Transfer Protocol (SMTP) to exchange emails among dedicated servers and for a client to send emails across its server; the Internet Message Access Protocol (IMAP) and the Post Office Protocol (POP) for a client to receive emails from its server; the Hypertext Transfer Protocol (HTTP) to communicate among hosts in a client-server architecture; and the Message Queue Telemetry Transport (MQTT) protocol for an alternative client-broker architecture of particular relevance in contexts such as the Internet of Things. We will come back to some of these protocols in later sessions of this course.

The TCP/IP model does not establish specifications for presenting data, and does not entail additional layers between the application and transport levels, as in the OSI model. Such functions are the realm of libraries and application programming interfaces implemented at application level. Data encoded at application level are then directly passed to the transport level for processing through relevant protocols, such as TCP or UDP in most cases. In principle, application level protocols are unaware of the underlying procedures, but may have knowledge of key features of the Transport Layer protocols in use, such as IP addresses and port numbers of the end-points. For instance, HTTP conventionally uses server port 80, and Telnet number 23. In turn, Transport Layer and lower-level protocols are unconcerned with the contents passed by the Application Layer protocols. Routers and switches primarily transmit the encapsulated traffic without parsing its payload. However, some firewall and bandwidth throttling services need to examine and interpret the application data, in order to ensure safety and quality.

We conclude this unit by remarking, as we have seen here and in previous units by means of a few real-world examples, that, within the Internet Protocol Suite, the interoperability between protocols at different layers is more a theoretical possibility than a practical option. So, for instance, it would be unsafe to run SMTP on top of UDP over geographical distances, or unfeasible to couple RTSP to TCP. For these reasons, layering in the context of the TCP/IP model is not a straitjacket, but rather a resilient and convenient framework to classify the entities needed in communication.

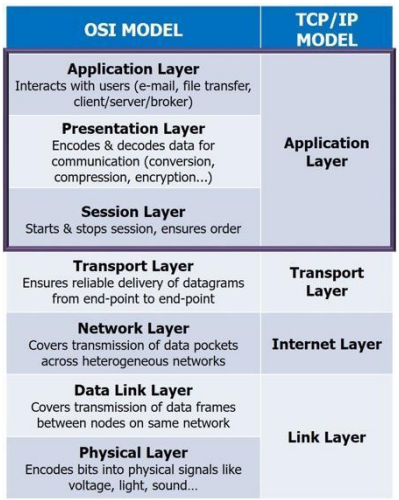


Figure: position of Application Layer in TCP/IP model CC BY-SA by INDEX consortium

Exercise

Which of the following statements is correct?

☐ In a typical configuration, TCP may run over UDP, which in turn may run over IPv6.

☐ In a typical configuration, IMAP may run over HTTP, which in turn may run over IPv6.
☐ In a typical configuration, IPv6 may run over HTTP, which in turn may run over TCP.
☐ In a typical configuration, HTTP may run over TCP, which in turn may run over IPv6.

Connections

Introduction

How to connect "things" has clearly a fundamental role in the context of the Internet of Things. When talking about link-level connections, it includes both wired options, such as Ethernet, and wireless solutions, such as Bluetooth or Wi-Fi.

The choice of the most suitable connection technology is one of the most important decisions to make when planning to develop any kind of IoT solution in different environments, whether it be a factory, home or farm, etc., since network connectivity is an essential part of the story.

But the assessment of which type of network connectivity is the best for a given IoT application is a complex issue, due to the multitude of different technologies that are available, ranging from Ethernet to wireless standards like cellular, Wi-Fi or Bluetooth, which are all possible options to connect sensors, actuators or smart devices. In some cases, a combination of different technologies may be the best solution.

The best choice depends on several factors, including the distance or range needed, the connection speed desired, or the power consumption of the edge devices. The ideal transport would consume very little energy, would have an unlimited range and would have high bandwidth for the transmission of large amounts of data. But since this option does not exist, each connectivity option will represent a trade-off between **POWER**, **RANGE** and **BANDWIDTH**.

In many practical contexts, IoT nodes should also enjoy:

- Limited processing power
- Possibility of battery supply and support for sleep mode operation
- Robust, suitable for harsh environments and weatherproof
- Easy to set up and maintain
- Low cost

Taking into account these parameters, the various connection options can be divided into three main groups:

INDUSTRIAL EXPERT

1. ***Short-distance, high bandwidth, low power consumption***

   Aim: to exchange a lot of data at reduced power consumption.

   To fulfill these conditions, it is necessary to reduce the distance. Connection modalities like WiFi, Bluetooth and Ethernet belong to this group. Ethernet is a wired connection, so its range is limited by the length of the cables. WiFi and Bluetooth are both wireless connections with lower power consumption than cellular and satellite, with comparable bandwidth but much lower range.

2. ***Medium-distance, low bandwidth, low power consumption***

   Aim: to exchange smaller amounts of data over longer distances, such as for agriculture 4.0.

   Connectivity options that combine low bandwidth with low power consumption but relatively long range are the best options. Solutions like low-power wide-area networks (LPWAN) are included in this group.

3. ***Long-distance, high bandwidth, high power consumption and costs***

   Aim: to exchange a lot of data, such as videos, over geographical distances, without limitations on power consumption or costs.

   Cellular and satellite technologies belong to this group. For instance, smartphones are able to receive and transmit large amounts of data over long distances, but need to be charged every 1-2 days at best.

The overall features of the connection options for the IoT are summarized in the following figure.

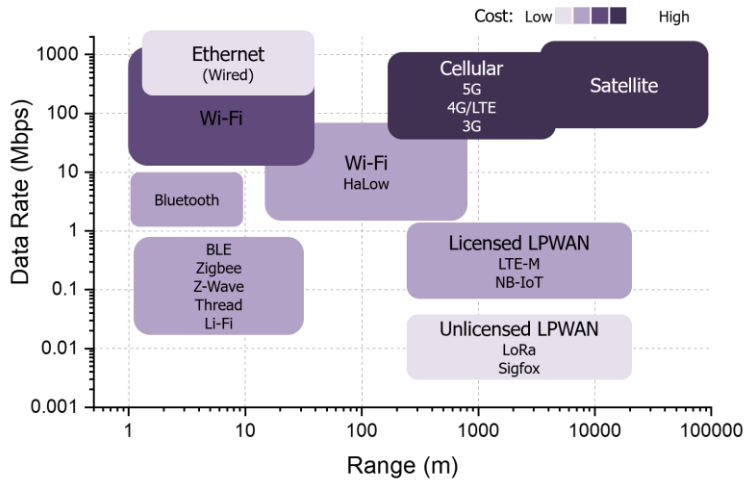


Figure: data rate and cost vs connection range for different connection technologies, CC BY-SA by INDEX consortium

Wired solutions

Ethernet

As mentioned in the previous unit, there is no such thing as a best data-link solution for all IoT technologies.

The best choice depends on the particular situation and the specific application.

In some cases, a wired network can still be the best option, with the **Ethernet** standard representing the most widespread technology.

In this case, the computer or other device is connected to a Digital Subscriber Line (DSL) via a network gateway or router through an Ethernet cable (RJ-45 standard). If phone lines, power lines, and coaxial cable lines are already available, this is an easy way to get plugged in, and often, even wireless networks are eventually connected to a wired network at some point!

Here are some of the benefits of **Ethernet**:

- **Worldwide universality**.
- **High speed**: wired connections are less affected by factors like distance or proximity to other devices, and this enables wired connectivity to be much faster than wireless counterparts. Gigabit Ethernet is now available on many routers with data speed up to 1 Gbit/s, but even classic Ethernet has better high-speed performance than Wi-Fi (802.11b/g).
- **Reliability**: Ethernet is not susceptible to interference from other wireless protocols and is less prone to dropped connections than wireless protocols.
- **Range:** up to 100 meters on a single cable.
- **Security**: wired connections are usually housed behind your Local Area Network (LAN) firewall, and hence allow for complete control over the communication flow. This means there is no broadcasting data that can be hacked into.

The limitations of Ethernet are:

- Need for a permanent cable connection.
- High power consumption.

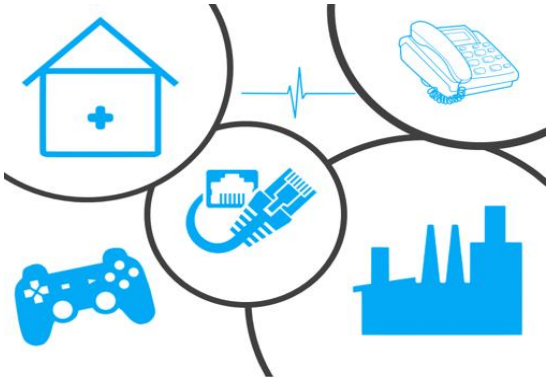


Figure: some examples of applications of Ethernet for the IoT, CC BY-SA by INDEX consortium.

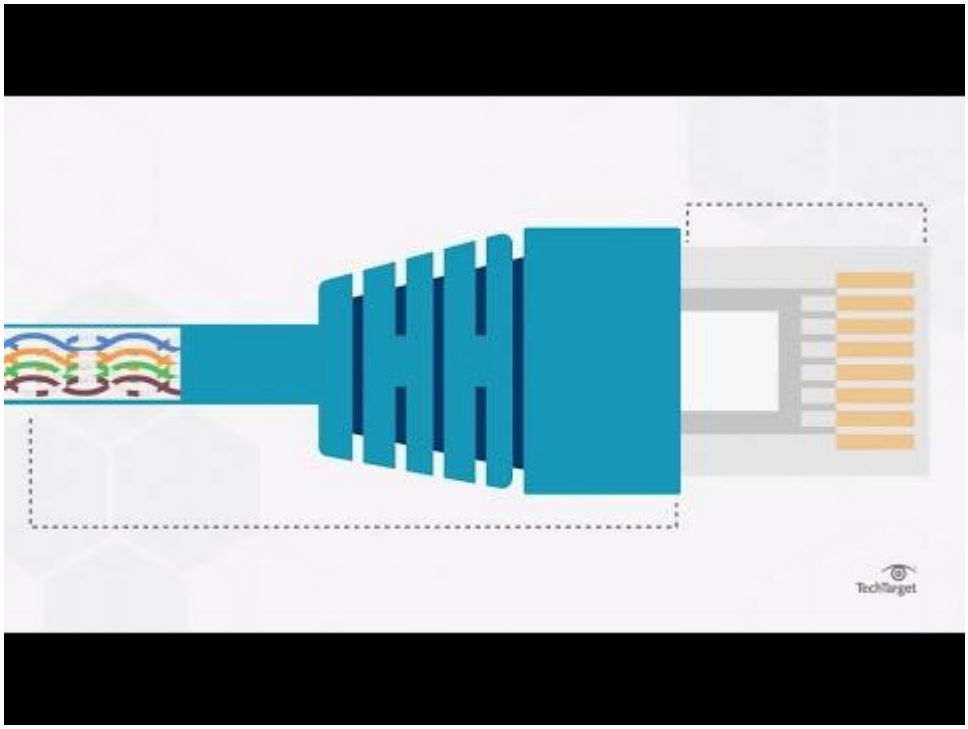
Typical uses of the **Ethernet** standard include all cases when one does not need a lot of range, but to transfer large amounts of data, such as:

- Voice over Internet Protocol (VoIP) applications,
- Game systems,
- Permanently installed industrial equipment,
- Applications that require high-reliability control like industrial control, robotics or medical uses, etc.
- Devices / environments, such as nuclear power plant controls, military networks or computerized medical equipment that need secure networks. An example is air gapped computers or networks, i.e. systems that are physically isolated from the Internet.

Pros and cons of Ethernet vs wireless systems are recapped in the following YouTube video.

An example of practical use of Ethernet in the IoT for Process Automation is shown in the following video.

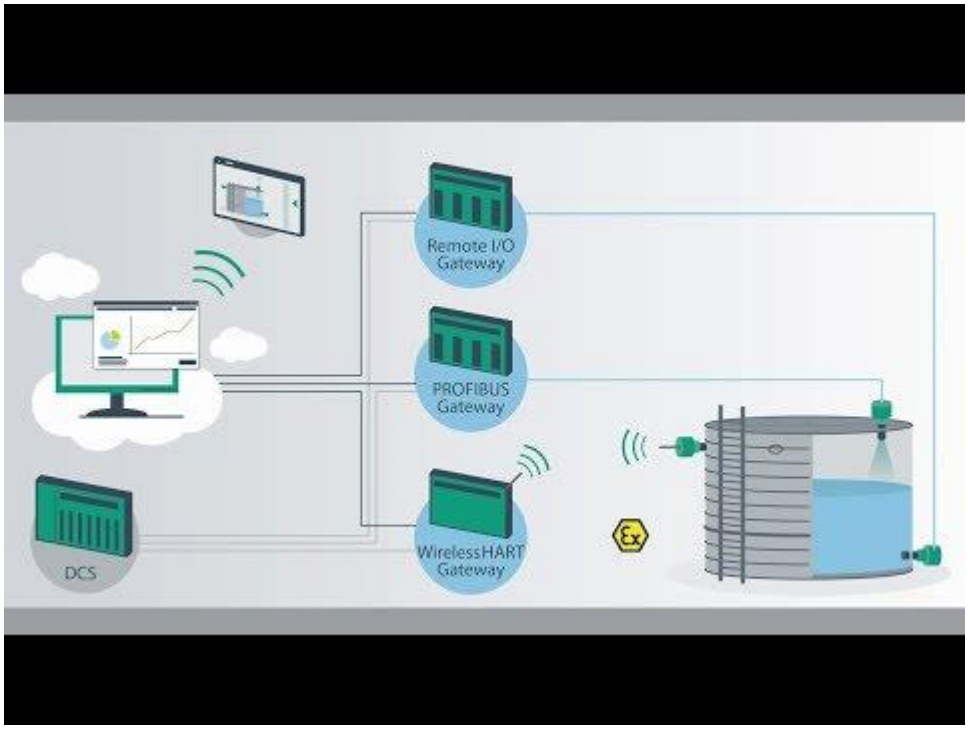
What is Ethernet? Ethernet vs Wi-Fi



Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before use.

An example of practical use of Ethernet in the IoT for Process Automation is shown in the following video.

IoT Solutions for Process Automation



Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before use.

Short-distance wireless solutions

Wi-Fi

Wi-Fi is by far the most commonly used wireless technology. It is present everywhere, providing ubiquitous access to the Internet in many different environments, such as homes, office buildings, academia and schools, campuses, residential homes, public buildings and so on. Wi-Fi is based on IEEE 802.11 standards, and the Wi-Fi Alliance, a non-profit association, guarantees the standardization and compatibility of commercial devices. It can be used to connect devices, sensors and actuators in small networks, or to connect systems to the Internet through Access Points.

However, despite their wide diffusion, interoperability and high bandwidth, Wi-Fi networks are not always the optimum for IoT applications, especially when the amount of data exchanged is not so big, but longer range, power autonomy, and reliability are required.

For these reasons, in some applications where power can be wired to the device (e.g. a thermostat), or routinely recharging a battery can be feasible (e.g. a smartphone), and the range is limited (indoor applications), Wi-Fi devices may be ideal for implementation in an IoT network, while for open air or remote applications, other solutions may be more preferable.

The main features of Wi-Fi are summarized in the next table:

Table: Wi-Fi main features

Range	Wi-Fi networks are classified as short-distance range. The typical point-to-point transmission range of Wi-Fi access points is between 30 m indoors to 100 m outdoors. In order to extend Wi-Fi coverage, it may be needed to install more access points and to create a wireless local area network (WLAN)
Power Consumption	Wi-Fi features a high-power consumption with respect to other common standards, such as Bluetooth or Low-Power Wide-Area Networks.
Frequency and data rates	2.4 GHz and 5 GHz radio frequency (RF) bands. The latest generation of Wi-Fi devices, based on the 802.11ax standard (Wi-Fi 6 devices), support a maximum link-rate of 600-9608 Mbit/s. While the majority of devices present in the market at the time of writing are based upon the 802.11ac standard (Wi-Fi 5 devices) that support a maximum link-rate of 433-6933 Mbit/s. Older equipment that support 802.11n standard (Wi-Fi 4 devices) exhibit a maximum link-rate of 72-600 Mbit/s. These data rates are ideal for data-intensive applications such web surfing, or audio or video streaming.
Reliability	Wi-Fi uses unlicensed RF bands within the 2.4GHz and 5GHz range. These frequencies are crowded and may raise reliability issues that may practically limit the range of the gateways to a few metres .
Interoperability	Wi-Fi is the most widespread wireless technology, thus assuring the universal interoperability of systems. Already in 2016, the Wi-Fi Alliance, which manages the certification requirements to ensure certain standards of interoperability, announced that Wi-Fi shipments had reached 12 billion units (https://www.wi-fi.org/news-events/newsroom/wi-fi-device-shipments-to-surpass-15-billion-by-end-of-2016).
Component availability and cost	Wi-Fi modules are widely available from dozens of vendors for a few dollars per unit.

Security	Security issues are a challenge for Wi-Fi connections. Access to WiFi networks requires authentication in order to control usage. Moreover it requires securing the data from unauthorized access by using encryption.

Wi-Fi HaLow is a recent and customized version of Wi-Fi specifically designed for IoT applications. It has been developed to overcome some of the limitations of Wi-Fi standards.

Wi-Fi HaLow operates in a spectrum of RF frequencies below one gigahertz, and so offers longer range and lower power consumption to fit most of the requirements of IoT scenarios with the advantage of Wi-Fi certification.

Highlights of this version are summarized in the next table.

Table: Wi-Fi HaLow main features.

Features	Benefits
• 1 GHz operation band; • Power saving modes; • Native IP support; • Wi-Fi security.	• Longer range (about 1 Km) and better penetration through walls and obstacles; • Longer life for batteries (months or even years); • No need for proprietary hubs or gateways.

Exercise

Why may the new Wi-Fi HaLow standard overcome some of the limitations of former Wi-Fi standards for the IoT?

☐ Wi-Fi HaLow allows higher data rates by maintaining the same power consumption as standard Wi-Fi, as well as compatibility with existing Wi-Fi networks.
☐ Wi-Fi HaLow operates below one gigahertz, thus offering longer range and lower power consumption than standard Wi-Fi, with the advantage of easy integration with existing Wi-Fi networks.

☐ Wi-Fi HaLow operates below one gigahertz, offering longer range and lower power consumption than standard Wi-Fi, at the cost of incompatibility with existing Wi-Fi networks.
☐ Wi-Fi HaLow operates below one gigahertz, thus offering longer range and compatibility with existing Wi-Fi networks, at the expense of higher power consumption than standard Wi-Fi.

The following video is about a practical example of one of the first wireless security camera on the market based on Wi-Fi Halow, which allows for an extended range while consuming very little power.

The First WiFi HaLow Security Camera

Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before.

Short-distance wireless solutions

Bluetooth and Bluetooth Low Energy (BLE)

Bluetooth technology is a short-range communication protocol widely employed for connecting IoT devices.

The Bluetooth standard was originally conceived at Ericsson back in 1994 as a way to replace RS-232 telecommunication cables by using low-power radio frequencies. The name "Bluetooth" comes from the second King of Denmark, Harald "Bluetooth" Gormsson, who became famous for having united the peoples of Denmark and Norway.

The certification of Bluetooth standards is controlled by the non-profit, non-stock corporation Bluetooth Special Interest Group (Bluetooth SIG), founded in September 1998.

Bluetooth protocol is classified as a short-range (up to 100 m), high power consumption, even if significantly less than Wi-Fi, and is particularly handy to connect devices like keyboards, headphones and so on. Classic Bluetooth operates in the same 2.4 GHz unlicensed spectrum as Wi-Fi, and is designed for continuous data transfer in one-to-one communication mode, with data rate up to 2 Mbps.

In order to overcome its limitations on power consumption, a new protocol was introduced in 2011, named **Bluetooth Low Energy** (BLE) or Bluetooth Smart, which significantly improves the energy use of connected devices. By activating the connection only when necessary and maintaining sleep mode the rest of the time, a small battery can last up to 4 or 5 years. The flip side of the coin is a lower bandwidth, as reported in the table below.

BLE does not only support one-to-one communication mode, but also broadcasting (one-to-many) and mesh networks (many-to-many). The introduction of many-to-many communication mode implies the possibility to significantly extend the range, thus potentially breaking the theoretical limit of about 100 m.

Bluetooth classic and BLE are actually very different protocols, and cannot interoperate. This means that Bluetooth classic and BLE devices cannot communicate with each other, although most modern devices, such as recent smartphones, support dual-mode Bluetooth modalities and can communicate with both kinds of devices.

At the time of writing, Bluetooth classic chips are still common in wireless telephone connections, such as wireless headphones or selfie sticks, while BLE is the first choice for the IoT, wearable devices or fitness monitoring equipment.

Table: comparison between Bluetooth Classic and BLE

INDUSTRIAL EXPERT

	Bluetooth Classic	Bluetooth Low Energy
Data rate	1-3 Mbps	1 Mbps
Frequency	2.4 GHz	2.4 GHz
Range	100 m	> 100 m
Power consumption	1 W	0.01 to 0.05 W
Time for sending data	Typically 100ms	Typically 3ms
Nodes	7	Unlimited
Target applications	Products that need continuous data/voice streaming	Products that need to transmit small amounts of data at low duty cycles (sensors, etc)
Examples	 Headphones, wireless speakers	 Wearable devices or fitness monitoring equipment

At the end of 2016, Bluetooth 5 was released to specifically target IoT applications. Bluetooth 5 combines some of the best features of Bluetooth classic and BLE, in terms of speed, range and power consumption, by achieving:

- 2x speed with respect to BLE;
- 4x range;
- Improved wireless coexistence.

These new features make Bluetooth 5 particular promising for IoT applications, such as low-quality video streaming over short distances, long-distance remote-control applications or synchronized monitoring of sensor data without the need for a connection.

In comparison with other technologies, Bluetooth 5 has similar range as Wi-Fi, the same smartphone support and also enables a mesh topology. But its power consumption is orders of magnitude lower than Wi-Fi.

The new features of Bluetooth 5 are also closer to those of Home Area Network radios, such as ZigBee, Z-Wave and Thread, which will be presented in the next unit, with the advantage of standard smartphone support.

Ellisys Bluetooth Video 9: Bluetooth 5 & IoT

Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before.

'Home Area Network' Radios

For applications like home or office-scale networks and automation, the so-called low-power radio networks represent a reasonable alternative.

These protocols, such as Z-Wave, ZigBee and Thread, are designed to cover areas comparable to those covered by Wi-Fi or BLE, but are extremely efficient from the point of view of power consumption for integration in battery-operated devices, and usable in mesh and high-density point-multipoint or multipoint-multipoint networks.

On the other hand, while Ethernet, Wi-Fi or Cellular rely on existing networks that a user can join as a client, and Bluetooth builds upon a phone or computer acting as network manager, with these low-power radios, one needs to run its own network.

The main differences between Wi-Fi, BLE and low-power radio networks are shown in the following table.

Table: comparison between Wi-Fi, BLE and low-power radio networks

	Wi-Fi	BLE	ZigBee	Z-Wave	Thread
Wide area coverage					
Power efficient					
High data bandwidth					

Zigbee

Zigbee is an open standard for wireless device-to-device communication built on the IEEE 802.15.4 specification and operates in the 2.4GHz RF band. Therefore, it shares the same frequency as Wi-Fi and Bluetooth.

Zigbee products are widely adopted by consumers, with dozens of chips available and flexible protocols. The large availability of different protocols for different applications, such as smart energy or commercial building automation, poses issues about their interoperability, but the Zigbee Alliance, a non-profit alliance of over 400 companies from around the globe, works towards the definition of unique Zigbee standards for communications.

Zigbee is a cost-effective and low-power solution for wireless personal area networks, and supports small amounts of data transmission over wide ranges, thanks to a mesh topology.

The main features of a Zigbee network are as follows:

- Low power: suitable for battery-operated devices with several years of battery life. For example, some Zigbee devices like gas meters can achieve as many as 20 years of battery life;
- Low data rate: radio data rate of 250 kbps, making Zigbee ideal for sampling or monitoring applications;
- Small and large networks are relatively simple to install and manage. The Zigbee standard supports various types of network topologies;

- Range sufficient to cover a normal home.

In a ZigBee system architecture, there are 3 device types:

- **Zigbee Coordinator (ZC)**: There is only one Coordinator in the network. It is a router with some additional functionality: to select the network topology, establish the network, and administer configuration information. The ZC must be running at all times;
- **Zigbee Routers (ZRs)**: ZRs provide routing services to network devices. A ZR may also serve as a sensor node, but unlike end devices, a ZR must always be on;
- **Zigbee End Devices (ZEDs)**: ZEDs are leaf nodes that communicate only through their parent nodes. ZEDs are usually battery powered and can be placed in low-power or sleep mode for long periods of time.

Zigbee supports three types of network topologies:

- **Star Networks**, formed by a group of ZEDs with a ZC as their parent serving as network hub;
- **Full Mesh Networks**, where all nodes are ZRs, including the ZC after it has established the network;
- **Hybrid Mesh Networks**, which combines star and mesh strategies. Several star networks exist, but their hubs can communicate as a mesh network. A hybrid network allows for longer distance communication than a star topology and more capability for hierarchical design than a mesh topology.

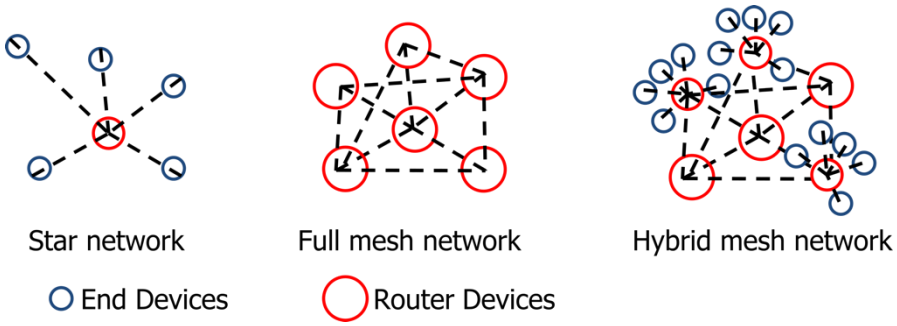


Figure: Zigbee network topologies, CC BY-SA by by INDEX consortium

Z-Wave

The Z-Wave protocol is another low-power, IoT wireless technology specifically designed for control, monitoring and status reading applications in home automation.

It is a proprietary solution broadly deployed, with over 100 million products sold worldwide. The Z-Wave Alliance that includes 700 companies, defines protocol standards and ensures interoperability between systems and devices from all members.

Z-Wave technology essentials are summarized in the following:

- Low Powered RF communications technology;
- Sub-1GHz RF band. As such, it suffers less from interference than its competitors like ZigBee, which operates in the 2.4GHz range;
- Supports data rates up to 100kbps, with AES128 encryption, IPV6, and multi-channel operation;
- Supports full Mesh Networks Topology, where any node can connect to any other.

In a Z-wave network there are two kinds of nodes: **controllers** and **slaves.**

- Controllers: a ll Z-Wave networks require at least one controller. The first one is the primary controller, and is responsible for including and excluding nodes and assigning individual Node IDs to each new device that is added to the network.
- Slaves: t hese nodes are devices such as lightbulbs, switches, etc., which do not feature a pre-programmed Home ID, as they take one assigned by the primary controller.

Thread

At the time of writing, Thread is an emerging IoT wireless technology among the 'Home Area Network' Radio solutions.

Thread is an open Internet Protocol resting on IPv6 and based on the broadly supported IEEE 802.15.4 radio standard, which is designed from the ground up for extremely low power consumption and low latency.

Unlike ZigBee, Thread enables not only device-to-device communication, but also device-to-cloud communication.

Ultimately, Thread can be considered as a networking equivalent to Wi-Fi, with the advantages of mesh topology and low-power consumption, making it ideal for home automation applications.

Therefore, this new and emerging protocol will probably make a significant impact in the context of short-range RF networks, in the near future.

Exercise

What is the approximate distance between devices connected through a so-called short-distance wireless connection?

☐ From a few centimetres to 1 metre.
☐ 1 to 3 metres.
☐ 1 to 100 metres.
☐ From a few centimetres to more than 1 kilometre.

Li-Fi

Li-Fi, or Light-Fidelity, is an alternative to the previous wireless technologies based on RF waves. The idea of Li-Fi was introduced for the first time by Harald Hass in the TED Global talk on Visible Light Communication (VLC) in July 2011.

This technology is based on the Visual Light Communication (VLC) principles: it is similar to Wi-Fi, with the main difference that Li-Fi uses the modulation of ultraviolet, infrared and visible light intensity, instead of radio frequency waves, to exchange data, thus allowing for much broader bandwidth. Li-Fi can theoretically transmit at speeds up to 100 Gbit/s, and is less susceptible to electromagnetic interference. Therefore, it may be ideal in critical environments like hospitals or planes.

The main components of a Li-Fi device are:

1. A high brightness LED as transmitter;
2. A silicon photodiode as receiver.

The spreading of solid state lighting technologies based on LEDs drove the birth of Li-Fi, because LEDs are electronic devices and their optical intensity can be modulated and switched on and off at very high speeds.

In this way, it is possible to generate digital strings of 1s and 0s by switching LEDs on and off. Then, data can be encoded in the light by varying the LED flickering rate at frequencies that are high enough to make the LED output constant to the human eye.

At the moment, the main applications of Li-Fi are envisaged for indoor environments, such as smart buildings, smart offices, hospitals and so on, thanks to the growing use of LED lamps in such environments.

However, Li-Fi is still in a development phase.

For more info on Li-Fi, listen to Harald Hass talk at TEDGlobal2011 in the following video.

Wireless data from every light bulb | Harald Haas

Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before

References

Short-range Wireless Communication (Third Edition), Editor(s): Alan Bensky, Newnes, (2019) ISBN 9780128154052, https://doi.org/10.1016/B978-0-12-815405-2.09989-9.

Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT

Medium-distance wireless solutions

Low Power Wide Area Networks (LPWAN): Introduction

As introduced in the previous units, the Internet of Things (IoT) constitutes the communication network of a variety of devices, home appliances, cars, and any other object that incorporates electronic media, software, sensors, actuators and network connectivity, and allows data connection and exchange. Simply put, the philosophy of the IoT is to connect all electronic devices to one another (local area network) and / or to the Internet (world wide web) creating Wireless Sensor-Actuator Networks (WSANs). Its operating network layer incorporates a fusion of public and private networks such as Local Area Networks - LANs (Bluetooth, Zigbee, Wi-Fi), Cellular Networks (GSM, 3G, 4G, 5G) and recently **Low Power Wide Area Networks - LPWANs** (LoRa, Sigfox, NB-IoT) and Satellite Networks (VSAT). The choice of the mixture of various networks intends to achieve a communication level of a certain quality and reliability by also taking into account the security needs concerning the transmitted data (Wei & Lv, 2019; The Things Network, 2019).

LPWANs are more suitable for IoT applications where each device needs to transfer a very small amount of data over a long range. The three most popular LPWAN technologies that have arisen at global level are **Sigfox**, **LoRa** and **NB-IoT**, which involve many technical and functional differences. The most important of these differences are the range (up to 40 km for Sigfox, 20 km for LoRa, 10 km for NB-IoT), the power consumption, where LoRa technology is the winning one, and the business model of operation, where LoRa is a public network, Sigfox is a private network, albeit both are using unlicensed bands, and NB-IoT is operated by cell carrier companies on licensed bands. As far as the topology is concerned, LoRa and Sigfox technologies need to connect to a gateway, which typically uses high bandwidth networks such as Wi-Fi, Ethernet or Cellular in order to connect to corresponding servers, thus implementing a star-of-stars topology, where gateways relay messages between end-devices and a central network server (The Things Network, 2019; LoRa Alliance, 2019).

The transmitted data from sensors belonging to a Wireless Sensors Network (WSN) follows specific standards that come together in so-called middleware integrating sensor nodes, actuator nodes and a base station. The latter takes on the role to collect, pre-process and transmit all data. Various middleware protocols have been developed focusing on different transmission needs, which can be evaluated against their efficiency, reliability, modularity, security and energy consumption (Nikolidakis, Kandris, Vergados & Douligeris, 2015). A very popular standard based middleware for transporting messages between devices is MQTT (Message Queuing Telemetry Transport), which is a Publish–Subscribe protocol invented in 1999 (Stanford-Clark & Hunkeler, 1999). It is a lightweight-messaging protocol developed to support conditions of low-bandwidth, high-latency and networks of low reliability. Consequently, and due to its mentioned lightweight properties, an extension version of MQTT has been developed for WSNs, and is called MQTT for Sensor Networks (MQTT-SN) (Al-Roubaiey, Sheltami, Mahmoud & Salah, 2019).

Each WSN can follow one of the three different topologies: star topology, tree topology or mesh topology (Wang et al., 2015). What LoRaWAN and Sigfox architectures use is a long range star topology, which makes the most sense for preserving battery lifetime when long-range connectivity may be needed (LoRa Alliance, 2015). In any case, prior to the selection of the most appropriate technology, different solutions should be evaluated in terms of performance criteria like received signal strength (RSS) and packet loss rate (PLR) (Jedermann et al., 2018).

LoRa vs Sigfox

Authors: Nikos Tsotsolas (Green Projects) and Ilias Kalfas (American Farm School)

LoRa and Sigfox are two of the main competitors in the LPWAN space. And while the business models and technologies behind the companies are quite different, the end goals of both Sigfox and the LoRa Alliance are very similar: t hat mobile network operators adopt their technology for IoT deployments over both city and nationwide low power, wide-area networks (LPWANs).

Sigfox is a network operated by Sigfox S.A., a private company based in France, which sets up antennas (gateways) on towers (like a mobile phone company). Sigfox is a narrowband (or ultra-narrowband) technology that uses a standard radio transmission method called binary phase-shift keying (BPSK), and it takes very narrow chunks of spectrum and changes the phase of the carrier radio wave to encode the data. This allows the receiver to only listen in a tiny slice of spectrum, which mitigates the effect of noise. It requires an inexpensive endpoint radio but a more sophisticated gateway (base-station) to manage the network. Sigfox communication tends to be better if it is headed up from the device to the gateway. It has bidirectional functionality, but its capacity going from the gateway back to the device is constrained, and you will have less link budget going down than going up. This is because the receiver sensitivity on the device is not as good as on the more expensive gateway (Ray, 2018).

LoRa stands for Long Range Radio. It is an open-standard governed by the LoRa Alliance, and it was developed by a company called Sentech. The expansion of this network is due to the rapid increase in the number of gateways both by telecommunication providers and in the context of the development of private networks. LoRa is a physical layer technology that modulates the signals in sub-GHz ISM (Industrial, Scientific and Medical) band using a proprietary spread spectrum technique. Like Sigfox, LoRa uses unlicensed ISM bands, i.e., 868 MHz in Europe, 915 MHz in North America, and 433 MHz in Asia. Both are mainly for applications that have many end-points and require uplink only. However, a bidirectional communication is provided by the chirp spread spectrum (CSS) modulation that spreads a narrow-band signal over a wider channel bandwidth. The resulting signal has low noise levels, enabling high interference resilience, and is difficult to detect or jam. Being a multiple access protocol, it follows star-on-star topology to relay messages. Communication to the central server is done using gateways by sending information on different frequency channels and data rates using coded messages. This method is efficient as the messages are less likely to collide and interfere with

one another. It therefore increases the capacity of the gateway. LoRa provides mobility, a bi-directional communication which is secure, and localization services to provide simple, seamless interoperability among smart things (Pandey, 2018; Ray, 2018).

In the following table, the LoRa and Sigfox technologies are compared against basic technical and non-technical characteristics.

Table: overview of LPWAN technologies: Sigfox, LoRa (Mekki, Bajic, Chaxel & Meyer, 2019)

	LoRa	**Sigfox**
Bands	Free bands (868 MHz in Europe)	Free bands (868 MHz in Europe)
Frequency	250 kHz	100 Hz
Data rate	50 kbps	100 bps
Duplex	Yes / Half-duplex	Limited / Half-duplex
Maximum number of messages / day	Unlimited	140
Maximum payload	243 bytes	12 bytes
Range	20 km, 5 km (urban)	40 km, 5 km (urban)
Authentication and encryption	Yes (AES 128 bits)	Not supported

Operators	Public	Sigfox
Allow private networks	Yes	No

Design a LPWAN Network

An important issue while developing LPWAN networks is the study of the overall coverage of the network. For the deployment of an IoT network, and the corresponding behaviour of the signal from the time that it is transmitted, until it reaches the receiver, no matter if it is initiated by the sensor device or the gateway, the coverage will be analyzed based on the radio channel characteristics. This analysis is crucial because the electromagnetic wave that is generated by the transmitter suffers attenuation (reduction in power density) before reaching the receiver. This phenomenon is called path loss, and is a major component in the analysis and design of the link budget of the network, since it allows to predict the coverage area, and also to quantify how reliable the radio link is for the area of interest.

The link budget in the IoT network includes several factors, such as emission power values, various equipment losses (gateway, sensors, cables, connectors, etc.), antenna gains and propagation effects. As for the latter, the signal in the propagation medium, which is air in most cases, is subject to attenuation, such as fading (multipath or Doppler), shadowing (interference caused by obstacles), interference and noise, further described in this lecture. All these factors should be considered and be part of the input parameters in the software that will be used for the development of such a study.

Before the selection of the exact placement of each gateway, a simulation procedure is strongly recommended, concerning the signal intensity for a range of 10 km (for urban areas) to 100 km (for rural areas) around the alternative mounting points. For such a simulation, several software tools are available in the market such as Cloud-RF™, which is an open source online service for radio signal propagation modelling taking into account the terrain of each area. In these software tools, the following information is fed to run the simulation process concerning the deployment of a LoRa network:

- LoraWAN signal characteristics;
- Number and exact position of the gateways;
- Technical characteristics of the gateways (e.g. antenna polarization, antenna gain, feeder line loss);
- Number of edge devices per type;

- Technical characteristics of the edge devices that affect the communication process (e.g. height above ground level, receive gain, sensitivity);
- Packets sizes;
- Packet arrival rates;
- Traffic characteristics.

The detailed coverage analysis, which will be produced through the simulation process, will provide maps that will show the signal strength for various spreading factors per gateway over the area where the edge-devices are planned to be placed. Different parameters will be taken into consideration within the propagation models / tools, such as line-of-sight and buildings and vegetation, for the coverage analysis. This analysis will predict coverage results such as best serving cell, overlapping area, and it will also propose detailed adjustments, such as gateway number, gateway configuration, antenna technical characteristics and parameters (height and tilt) after analyzing the coverage prediction results. The coverage analysis will provide simulation results for different edge-nodes positions, such as signal strength, signal-to-noise ratio, uplink and downlink throughput, delay and packet error rate. The coverage analysis shall also include result maps in 2D and 3D, propagation paths in 2D and 3D, statistical evaluations of result maps, downlink and uplink quality indicator (BLER, PER, etc.) analyses and coverage redundancy analysis. The following image shows an example in Google EarthTM of a coverage map produced by Cloud-RF™

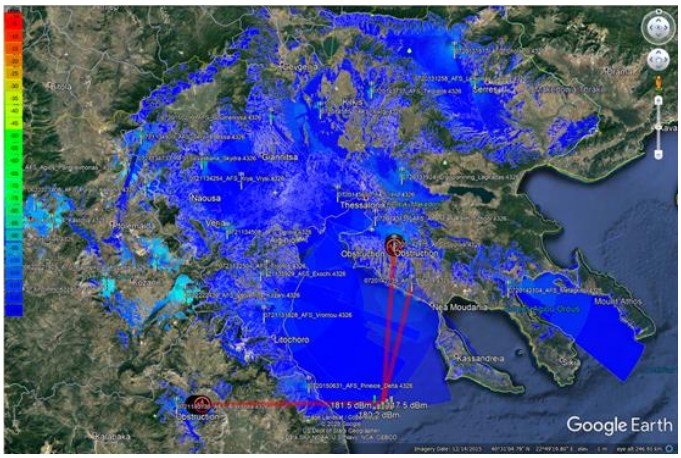


Figure: Coverage Map of a LoRa Network produced in Cloud-RF™, CC BY-SA by by INDEX consortium

References

Al-Roubaiey, A., Sheltami, T., Mahmoud, A., & Salah, K. (2019). Reliable Middleware for Wireless Sensor-Actuator Networks. IEEE Access, 7, 14099-14111. doi: 10.1109/access.2019.2893623

Jayaraman, P., Yavari, A., Georgakopoulos, D., Morshed, A., & Zaslavsky, A. (2016). Internet of Things Platform for Smart Farming: Experiences and Lessons Learnt. Sensors, 16(11), 1884. doi: 10.3390/s16111884

LoRa Alliance. (2019). About LoRaWAN® | LoRa Alliance®. Retrieved 14 December 2019, from https://lora-alliance.org/about-lorawan

Mekki, K., Bajic, E., Chaxel, F., & Meyer, F. (2019). A comparative study of LPWAN technologies for large-scale IoT deployment. ICT Express, 5(1), 1-7. doi: 10.1016/j.icte.2017.12.005

Pandey, P., 2018. Comparative study of long range communications, Pilani, India: Birla Institute of Technology and Science.

Exercise

As illustrated in the previous units, the choice of the most suitable data-link technology is not a trivial issue.

Look at this example reported in the open-access paper: An Internet-of-Things (IoT) Network System for Connected Safety and Health Monitoring Applications about the development of an IoT network system for connected safety and health monitoring. The following figure provides a sketch of the corresponding architecture.

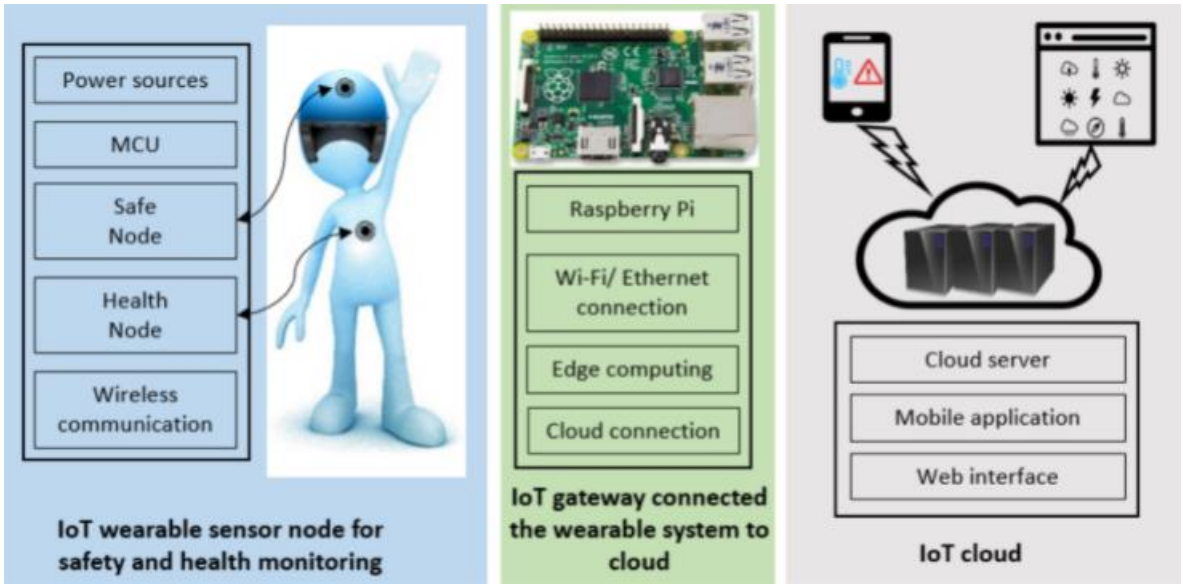


Figure: figure 1 from Wu, F.; Wu, T.; Yuce, M.R. An Internet-of-Things (IoT) Network System for Connected Safety and Health Monitoring Applications. Sensors 2019, 19, 21., CC BY-SA by Sensors 2019, 19, 21

This system is divided into 3 sections: 1) the wearable sensor node (Wearable Body Area Network - WBAN) to collect sensor data, 2) an IoT gateway to connect the WBAN with the Internet , and 3) the IoT cloud.

Question

In the end, which wireless technologies have been selected for this project?

☐ BLE for both WBAN and the IoT gateway.
☐ Wi-Fi for both WBAN and the IoT gateway.
☐ LoRa for both WBAN and the IoT gateway.
☐ A hybrid network consisting of BLE for WBAN and LoRa for the IoT gateway.

Long-distance wireless solutions

Cellular

Broadband cellular connectivity plays a fundamental role for the IoT, thanks to its global reach, scalability, and high bandwidth capabilities, but at the expense of substantial power consumption.

This is acceptable for devices that are connected to a power grid or that can be recharged on a frequent basis, but much less so for IoT applications that require remote sensors, actuators or devices to last months or years on battery.

For this reason, cellular connectivity is usually reserved for so-called backhaul networks. For instance, a gateway may use a LPWAN system to connect all sensors and actuators, and then use cellular to transmit the collected data to the cloud. Other uses are in devices that need to exchange a lot of data without issues related to battery life.

In this context, new cellular technologies like LTE-M (Long-Term Evolution-M), and Narrowband Internet of Things (NB-IoT) have been developed specifically by 3GPP (3rd Generation Partnership Project) for IoT applications, as upgrades to cellular networks like 2G, 3G or 4G with the specific goal to reduce data costs per device and power requirements.

These technologies are LPWANs, based on the GSM/EDGE and UMTS/HSPA technologies. As an example, NB-IoT is a LPWAN radio technology standard developed by 3GPP to enable a wide range of cellular devices and services with downlink and uplink peak rates up to 127 and 159 kbit/s, respectively.

Some of these cellular technologies are currently available, and others are still under development at the time of writing, including 5G.

The fifth-generation wireless technology will make an impact not only in high-speed mobile communication but also for IoT applications. In comparison with the existing 4G network, the 5G protocol:

- will enjoy a higher bandwidth throughput;
- will be 100 times faster than the current network;
- will support a massive scale of IoT communications, in the range of 1 million IoT devices per square kilometer;
- will yield a 90% reduction in power consumption.

Satellite

Satellite connectivity finds a place in IoT applications mainly when broad coverage is required or remote contexts are concerned, since a single network of satellites is capable of providing full coverage over the entire planet. In fact, cellular networks are widespread in populated areas, but their coverage in remote locations is limited or unreliable.

Conversely, the range of satellite networks can overcome the limitations of cellular networks and guarantee connection in places where infrastructures are underdeveloped or absent, such as the middle of the ocean or on high mountains.

For IoT applications, two kinds of connectivity configurations may be implemented: direct and backhaul.

Direct: dual mode or satellite only

The dual mode satellite connectivity uses cellular data when available, and switches to satellite only when necessary. This guarantees the best coverage, while exploiting the lower cost and higher bandwidth of cellular whenever possible. This system is employed in container ships, for instance, which may use cellular when close to coastlines, but then resort to satellite on the open water.

The satellite-only option is typically used by standstill resources like oil and gas equipment, which may need to send moderate amounts of data from remote locations.

Backhaul

The second major type of configuration is backhaul, and makes use of a main tower that connects directly to a satellite network, and then a different kind of connectivity, such as an LPWAN platform, to connect with the sensors, actuators and edge-devices in the area. This connectivity option is typically used when one needs many low bandwidth devices in remote areas.

Satellite requires high power usage, and can require larger pieces of equipment, such as dishes, for connectivity. This raises the cost for individual devices, and can make direct connections unfeasible for groups of sensors or actuators that do not exchange much data.

One example of this kind is a farm that uses a set of moisture sensors to collect soil data. All sensors may use an LPWAN network to connect to a main tower, which then transmits the data over a satellite connection. This saves on battery life and may considerably reduce the overall costs of the sensors.

References

Wei, J.; Han, J.; Cao, S. Satellite IoT Edge Intelligent Computing: A Research on Architecture. Electronics 2019, 8, 1247. https://doi.org/10.3390/electronics8111247

M. De Sanctis, E. Cianca, G. Araniti, I. Bisio and R. Prasad, "Satellite Communications Supporting Internet of Remote Things," in IEEE Internet of Things Journal, vol. 3, no. 1, pp. 113-123, Feb. 2016, doi: 10.1109/JIOT.2015.2487046.

Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT

Exercise
Question

In which of the following IoT architectures do you think that a long-range solution, such as cellular or satellite, could be the most suitable solution?

☐ Outdoor environmental monitoring in open nature to manage natural disasters, such as earthquakes, fires or floods.
☐ Traffic monitoring in a big city.
☐ Real-time healthcare monitoring of patients in hospitals.
☐ Smart whiteboard for connected classrooms.

Poll
IOT CONNECTIONS

In your opinion, which IoT connection technologies will boost more in the next 2 years?

☐ Next Wi-Fi generation, i.e. 5g connection.

☐ Low power, short range connections like BLE, ZigBee or Z-wave.

☐ Low Power Wide Area Networks like LoRa, Sigfox or NB-IoT.

☐ New disruptive technologies, such as Li-Fi.

Application layer protocols
Introduction

An application-layer protocol is the language used by computers and devices to communicate above a transport-layer infrastructure, and defines the procedures for transferring data at the highest level of abstraction. A comprehensive discussion of this topic falls outside of the scope of this section, which is to do a roundup of some common solutions, and to discuss their advantages and limitations for IoT applications, with the intent to provide the trainee with general concepts and practical guidelines.

As we have already mentioned, in principle, the choice of a particular application-layer protocol is independent of the underlying transport, network and data-link stack, as well as the overlying scope, i.e. the final user application. However, in reality, practical considerations create a strong interconnection among all these components:

- The overall amount of data that is exchanged in the communication process, in combination with the size of the network packets and with the bandwidth that are supported in the underlying layers;
- The need to interface to overlying services, such as external web resources, for instance.

For these reasons, the entire stack from link to application-layer protocols is often devised as a coherent suite, such as the cases of networks based on Bluetooth, LoRaWAN or Zigbee. Here, we shall focus on the most common solutions of application-layer protocols met in the so-called TCP/IP model of the Internet.

HTTP: General features

The **Hypertext Transfer Protocol** (HTTP) is the most popular application-layer protocol running over transport-layer protocols as TCP and UDP for communication through the World Wide Web and is primarily designed to exchange so-called *hypertexts* through so-called *hyperlinks*. The default ports in use with HTTP are 80 or alternatively 8080. The protocol is complex and has undergone various revisions until the current and upcoming versions, HTTP/2 and HTTP/3. Here, we shall focus on a narrow set of classical features, although there exist complementary techniques that enable different behaviours and versatile personalization.

Client-server *architecture*: the main characters in HTTP are **clients** and **servers**. In this architecture, one client, such as a web browser, issues a request message to one server that returns a response message often containing HTML code, for instance. Response messages may be cached in intermediate servers to optimize network traffic. From the get-go, we notice that such client-server architecture may be problematic in many IoT contexts. Consider a generic case where a client may need to receive critical updates from a server. Since the server cannot notify the client on its own initiative, the classical solution is polling, i.e. the client querying the server at frequent intervals. However, polling may be either too sporadic or consume too much bandwidth, especially in contexts where updates are rare but latency is an issue. In a later unit, we will mention the scope of server-push technologies that are intended to patch these situations. However, it is important to understand that the client-server architecture may pose these kinds of nuisances.

Consider, for instance, a server collecting data from a client posting the temperature probed in a compartment of a greenhouse. Another client may want to get this information, in order to make urgent decisions, such as turning a heater on below a certain threshold. Although the first client may post new data without delay, the second client needs to poll the server with a certain frequency in order to be able to react as soon as possible. But the threshold may be crossed only once in a while, and so this setup may result in a substantial waste of bandwidth.

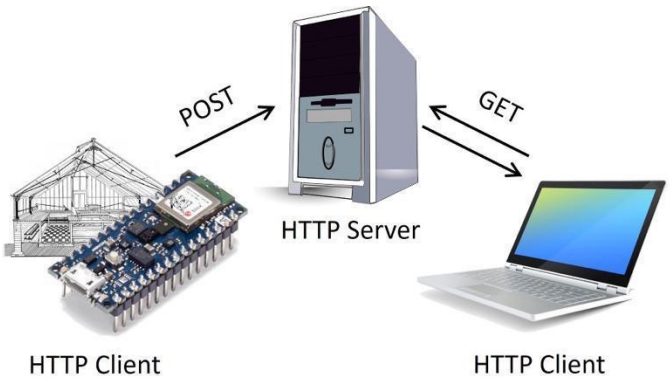


Figure: Client-server architecture, CC BY-SA by INDEX consortium

Session- and state-lessness: the original version of HTTP provided that every single couple of request and response messages consisted of a new session. This structure was designed to streamline the exchange of hypertexts hosted in different servers through hyperlinks cross-referenced in different webpages, without the burden to keep too many sessions open. The flip side of the coin is poor latency, because every single request message sets up a new connection over a protocol like TCP that may entail complex handshake procedures, such as 3-way handshake rules. Later versions of HTTP have introduced different ways to implement persistent connections for bandwidth optimization. Another peculiar feature of HTTP is the concept of statelessness, i.e. that the server does not retain any information on the client. In this context, **cookies** are a common tool to overcome possible limitations of this feature. Cookies are textual strings transmitted by the server together with a response message, which the client may incorporate in the header field of any subsequent request message addressed to the same resource. For instance, cookies may contain a token meant to authenticate a client and maybe identify its chart in a website for online shopping, to name but one example.

Message format: messages are transmitted as textual strings.

A **request message** sent from a client to a server consists of the following components:

- A request line (e.g., GET /images/logo.png HTTP/1.1). In the next unit, we will expand on the main kinds of actions and the identification of resources requested through this line. The request line terminates with the chars carriage return *<CR>* and line feed *<LF>*;
- Request header fields (e.g., Host: en.wikipedia.org). These fields define various parameters of the transaction formulated as an arbitrary number of colon-separated key-value pairs, and also end with the chars *<CR><LF>*;

- An empty line. I.e. the chars *<CR><LF>*;
- An optional message body. For instance, this part may contain a textual string to be put or posted in a form.

Instead, a **response message** sent from a server to a client contains the following components:

- A status line (e.g., HTTP/1.1 200 OK, or 404 Not Found). This line includes a status code in numerical format and a reason message as plain text. Also the status line terminates with the chars *<CR> <LF>*;
- Response header fields (e.g., Content-Type: text/html). These fields are similar to those of request messages and always end with the chars *<CR> <LF>*;
- An empty line. I.e. the chars *<CR><LF>*;
- An optional message body. For instance, this part may consist of HTML code for creating a webpage.

In practice, messages in HTTP are based on textual instructions and often contain substantial overhead and chunks of HTML code. In the next unit, we will introduce the concept of REST architecture with a focus on the request line of request messages.

HTTP: RESTful architecture

The concept of the **Representational State Transfer** (REST) is an architectural style that is common in many practical implementations of HTTP. Besides the concepts of client-server architecture, statelessness and cacheability that we have already mentioned, REST enforces the additional constraint of a uniform interface as a common set of rules intended to support the scalability of the Internet. These rules are implemented from the request line of request messages.

URIs: Data are organized in resources made addressable though **Unique Resource Identifiers** (URIs). Resources may be collections of elements or single elements represented in a standard format, such as HTML, XML or JSON. URIs are typically structured in hierarchical layers separated by slashes like https://www.cnr.it/en/department/513/engineering-ict-and-technologies-for-energy-and-transportation .

Methods: clients interact with these resources though a standard set of methods that may be expressed as verbs or other keywords. There exist many methods regimented in HTTP. The most common and iconic cases are the verbs **GET**, **PUT**, **POST** and **DELETE**.

Table: most common methods.

Method	Effect on a collection like http://api.example.com/resources/	Effect on an element like http://api.example.com/resources/myElement.html
GET	Returns a list of elements and additional details	Returns a representation of the element in a certain format
PUT	Replaces the entire collection with a new one	Replaces the element or creates a new one if it did not exist before
POST	Creates a new element in the collection. Its URI is automatically assigned and returned in the response message	Creates a new item within the element, if possible
DELETE	Deletes the entire collection	Deletes the element

Some methods are safe or *nullipotent*, such as GET, and do not alter the addressed resources. Others are *idempotent*, such as PUT and DELETE, because their repetition does not cause multiple effects.

While REST is not a standard definition per se, RESTful implementations often make use of standard protocols like HTTP and CoAP, and standard formats, such as HTML, XML and JSON, which will be the subject of the next section. However, before we move onto these formats, we briefly introduce CoAP as an interesting variation on HTTP.

HTTP and CoAP

CoAP

In previous units, we have seen that HTTP is an application-layer protocol originally designed to exchange hypertexts through hyperlinks for a human interface, such as a DIYer surfing on the Internet. Request and response messages are made of textual strings that often contain keys encoded in plain English. Resources are typically represented through a markup language like HTML, which will be the

subject of the next section. Overall, HTTP does not prioritize the optimization of bandwidth and was not optimized for M2M communication.

The **Constrained Application Protocol** (CoAP) is a lightweight variation on HTTP specialized for constrained devices called *nodes*, and for IoT applications. CoAP may serve communications between nodes on the same constrained network, as well as between nodes and general devices on the Internet and between nodes on different constrained networks joined by the Internet, and works over UDP. A convenient feature of CoAP is its interoperability with HTTP for integration with web services, while providing additional features such as multicast support for point-to-multipoint communication, minimal overhead and simplicity.

Like HTTP, CoAP makes use of request and response messages, but unlike HTTP, its header fields are in simple and binary format. All messages include the following components:

- A 4-byte header encoding the version number; the type of message, i.e. whether it is a request or a response and additional details; the length of a so-called token (see below); a request or response code containing a status code, a method or additional details; and a message ID to detect duplications and pair requests and responses. This is the only necessary component, and so messages can be as short as 4 bytes;
- An optional token serving as a local identifier of the client for use in case of concurrent transactions;
- Possible options;
- An optional payload.

CoAP adheres to REST principles, and implements the same concepts of URIs and methods as HTTP to address and interact with resources. Other improvements over HTTP are that CoAP provides an **OBSERVE** function that a client may add as a flag in a GET message to receive updates from a server, and so practically implement a push behaviour , and a built-in **DISCOVERY** function that a client may exploit to locate resources offered by a server. The format of resources found in HTTP will be the subject of the next section.

Exercise

The paper Analysis of CoAP Implementations for Industrial Internet of Things: A Survey describes various implementations of CoAP for applications at industrial level. Which of the following statements is true?

☐ The Group Communication for the CoAP extension allows CoAP to behave like HTTP in terms of multicasting and beaconing.
☐ CoAP is a lightweight alternative to C, Java, JavaScript and Python, which is designed for industrial use.
☐ CoAP cannot use the Transport Layer Security (TLS) protocol, because it runs over UDP, rather than TCP. Relevant alternatives include the Datagram Transport Layer Security (DTLS) protocol.
☐ The Block-Wise Transfers in the CoAP extension allows CoAP to run over TCP, and therefore to exchange larger payloads.

Common formats found with HTTP

HTML

The **Hypertext Markup Language** (HTML) is the most standard format for resources found in HTTP. What a server returns as a response message is likely to contain HTML code as a message body. Then, it is the task of a browser to render this code into a multimedia page. To get an idea, the option View Source is there to display the original payload in most common browsers.

HTML code provides a means to structure hypertexts by denoting items like headings, paragraphs, lists, links, quotes, etc., and by enabling the incorporation of images and other objects, such as interactive forms, etc. HTML elements are designated by tags delimited by angle brackets. Tags such as ** or *<input ... />* introduce content into the hypertext. Other pairs of start and end tags, such as *<p>* and *<\p>*, surround and provide instructions to format plain text, and may include attributes and more tags as sub-elements.

For instance, the following figure shows an example of a simple page rendered with HTML code.

INDUSTRIAL EXPERT

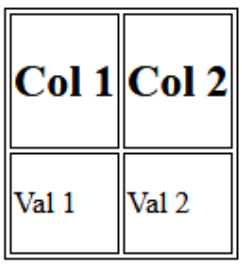


Figure: CC BY-SA by INDEX consortium

The source code responsible for the above figure may be the following:

```
<!DOCTYPE html>
<html>
  <body>
    <div>
      <h1>Par title</h1>
      <p>Par text</p>
    </div>
    <div>
      <h1>Table title</h1>
      <table style="border: 1px solid black;">
        <thead>
          <tr style="border: 1px solid black;">
            <th style="border: 1px solid black;">
              <p><h2>Col 1</h2></p>
            </th>
            <th style="border: 1px solid black;">
              <p><h2>Col 2</h2></p>
            </th>
          </tr>
```

```
        </thead>
        <tbody>
          <tr style="border: 1px solid black;">
            <td style="border: 1px solid black;">
              <p>Val 1</p>
            </td>
            <td style="border: 1px solid black;">
              <p>Val 2</p>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </body>
</html>
```

At first sight, the source code may be rather intimidating. The text between *<html>* and *</html>* describes the overall page, and the text between *<body>* and *</body>* is the visible content. The tag * * defines a division of the page, while the tags *<h1>* and *<p>* respectively indicate titles and paragraphs. The construction of a table may benefit from the definition of sections, and requires the indication of rows and columns with tags as *<tr>*, *<td>*, etc. Typical pages found in the Internet often arise from thousands of lines of HTML code.

As the word suggests, HTML is a markup language that is mostly designed for a human interface. Exchanging and parsing thousands of lines of HTML code in the context of M2M communication may be a very inefficient approach. For this reason, other formats may be more suitable for the IoT.

XML and JSON APIs

The concept of web **Application Programming Interface** (API) is there to simplify the consumption of resources in the IoT. In their broader meaning, APIs are computing interfaces that define the interactions between multiple software applications. In the particular context of web APIs based on HTTP, a practical approach is to encode resources in XML or JSON format as machine-friendly alternatives to HTML.

The **Extensible Markup Language** (XML) is a markup language similar to HTML, and defines rules for encoding documents in a format that is both human as well as machine-readable. With respect to HTML, XML puts more emphasis on the concepts of simplicity, generality and usability across the

Internet. Although its design focuses on textual data and Unicode characters, XML is widely used to represent arbitrary structures of data.

The **JavaScript Object Notation** (JSON) is an open format that uses human-readable text to store and transmit data as simple attribute-value pairs or arrays. JSON was derived from JavaScript, but any programming language may be suitable to generate and parse files in JSON format. With respect to HTML and XML, JSON is not a markup language.

Many servers provide web APIs featuring resources in XML or JSON format, which may be used in the IoT. One example is OpenWeatherMap, which provides the weather forecast or current situation in a specified location. Suppose for instance that a certain application requires the current temperature in Florence, Italy.

The HTML API available at an URI like http://api.openweathermap.org/data/2.5/weather?q=Firenze&mode=html&appid=... , where key app id is a personal identifier that has been blanked for security reasons, returns the web page shown in the following f igure.

Figure: CC BY-SA by INDEX consortium

The corresponding source is the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="keywords" content="weather, world,
openweathermap, weather, layer" />
```

```
  <meta name="description" content="A layer with current
weather conditions in cities for world wide" />
  <meta name="domain" content="openweathermap.org" />
  <meta http-equiv="pragma" content="no-cache" />
  <meta http-equiv="Expires" content="-1" />
</head>
<body>
  <div style="font-size: medium; font-weight: bold; margin-
bottom: 0px;">Florence</div>
  <div style="float: left; width: 130px;">
    <div style="display: block; clear: left;">
      <div style="float: left;" title="Titel">
        <img height="45" width="45" style="border: medium
none; width: 45px; height: 45px; background:
url(&amp;quot;http://openweathermap.org/img/w/10d.png&amp;quot;)
repeat scroll 0% 0% transparent;" alt="title"
src="http://openweathermap.org/images/transparent.png"/>
      </div>
      <div style="float: left;">
        <div style="display: block; clear: left; font-size:
medium; font-weight: bold; padding: 0pt 3pt;" title="Current
Temperature">15.86°C</div>
        <div style="display: block; width: 85px; overflow:
visible;"></div>
      </div>
    </div>
    <div style="display: block; clear: left; font-size:
small;">Clouds: 75%</div>
    <div style="display: block; clear: left; color: gray;
font-size: x-small;" >Humidity: 55%</div>
    <div style="display: block; clear: left; color: gray;
font-size: x-small;" >Wind: 2.6 m/s</div>
    <div style="display: block; clear: left; color: gray;
font-size: x-small;" >Pressure: 1003hpa</div>
  </div>
  <div style="display: block; clear: left; color: gray; font-
size: x-small;">
```

```
    <a
href="http://openweathermap.org/city/3176959?utm_source=openwe
athermap&utm_medium=widget&utm_campaign=html_old"
target="_blank">More..</a>
  </div>
  <script>(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=
r;i[r]=i[r]||function(){
(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
Date();a=s.createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.
insertBefore(a,m)
})(window,document,'script','//www.google-
analytics.com/analytics.js','ga');ga('create', 'UA-31601618-
9', 'auto');ga('send', 'pageview');</script>
</body>
</html>
```

Note that the identification of the info Current Temperature = 15.86°C may require a substantial consumption of memory space and computational resources. This API is clearly designed for a human interface. In spite of its minimal appearance, the text amounts to 2201 chars, let alone the image stored under http://openweathermap.org/img/w/10d.png .

The same resource represented as XML API at an URI like http://api.openweathermap.org/data/2.5/weather?q=Firenze&mode=xml&appid=... returns the following code:

```
<current>
  <city id="3176959" name="Florence">
    <coord lon="11.25" lat="43.77"/>
    <country>IT</country>
    <timezone>7200</timezone>
    <sun rise="2020-09-27T05:08:08" set="2020-09-
27T17:03:34"/>
  </city>
  <temperature value="288.8" min="288.15" max="289.26"
unit="kelvin"/>
```

```
  <feels_like value="286.2" unit="kelvin"/>
  <humidity value="55" unit="%"/>
  <pressure value="1003" unit="hPa"/>
  <wind>
    <speed value="2.6" unit="m/s" name="Light breeze"/>
    <gusts/>
    <direction value="30" code="NNE" name="North-northeast"/>
  </wind>
  <clouds value="75" name="broken clouds"/>
  <visibility value="10000"/>
  <precipitation value="0.32" mode="rain" unit="1h"/>
  <weather number="500" value="light rain" icon="10d"/>
  <lastupdate value="2020-09-27T14:47:19"/>
</current>
```

The structure is now much less verbose with 769 chars. The temperature is given in Kelvin, but its identification requires processing much less data.

Finally, the JSON API at an URI like http://api.openweathermap.org/data/2.5/weather?q=Firenze&appid=... provides the following file:

```
{"coord":{"lon":11.25,"lat":43.77},"weather":[{"id":500,"main"
:"Rain","description":"light
rain","icon":"10d"}],"base":"stations","main":{"temp":288.79,"
feels_like":286.54,"temp_min":288.15,"temp_max":289.26,"pressu
re":1002,"humidity":55},"visibility":10000,"wind":{"speed":2.1
,"deg":50},"rain":{"1h":0.32},"clouds":{"all":40},"dt":1601217
909,"sys":{"type":1,"id":6804,"country":"IT","sunrise":1601183
288,"sunset":1601226214},"timezone":7200,"id":3176959,"name":"
Florence","cod":200}
```

The information is even more concise with 483 chars only, and free of any markup instruction, although a human reader may still easily parse the content, which continues to be based on plain English, and write code to program a software application to do the same.

Overall, HTTP-REST is a ubiquitous protocol that was designed for a human interface surfing from hypertext to hypertext, which is being adapted to the arena of M2M communication as well as by the

integration of variants or tools like CoAP or XML and JSON APIs. In the next section, we will introduce an alternative protocol that was purposely designed for the IoT.

MQTT

General features

The **Message Queuing Telemetry Transport (MQTT)** technology is a lightweight publish-subscribe protocol for communication between devices, and typically runs over TCP over IP networks, or any other transport-layer protocol supporting bi-directional connections, such as Bluetooth. MQTT was specifically designed to exchange data in remote environments suffering from small bandwidth and unreliable connectivity, and its name is probably more of a historical value than a descriptive meaning. MQTT sends credentials as plain text and does not provide for security or authentication. However, it can be implemented over the Transport Layer Security (TLS) presentation-layer protocol for encryption and protection. The default port for unencrypted connections is 1883. The encrypted counterpart is 8883.

Client-broker architecture: the main characters in MQTT are **clients** and **brokers**. Therefore, the foremost difference with HTTP is that servers give way to brokers. The role of brokers is to receive and instantly route messages among the appropriate clients by pull and push operations. Data are stored in a hierarchy of so-called *topics* that will be the subject of the next unit. Clients may publish and subscribe to these topics. In practice, when a client publishes new data to a topic, the broker takes charge to distribute that information to all clients that have subscribed to that topic. The client publishing new data does typically not know the number nor the address of the clients receiving that information. Likewise, the clients subscribing to a certain topic may be unaware of the client originating the subscribed data.

Consider, for instance, the case of a broker collecting data from a client publishing the temperature probed in a compartment of a greenhouse, which we have mentioned in an earlier unit on HTTP-REST with due changes. Another client may want to subscribe to this information, in order to make urgent decisions, such as turning a heater on below a certain threshold. The first client may post new data without delay, and now also the second client is confident they will receive the alert of interest both without lag and only when really needed.

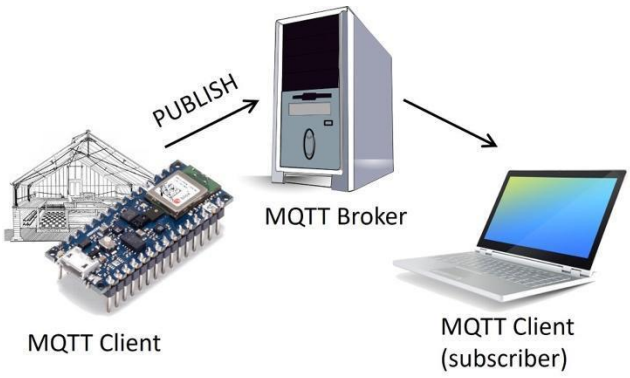


Figure: Client-broker architecture, CC BY-SA by INDEX consortium

Session- and state-fullness: MQTT works on so-called *sockets*, which are persistent sessions between a client and a broker. When a client connects to a broker, it can use a flag to set whether its session is durable or not, i.e., upon disconnection and reconnection, whether to resume or abandon its current subscriptions and any process left halfway. This feature may be useful to minimize the setup of clients over unreliable networks: the client is already identified to the broker by the CONNECT request, and does not need to resend SUBSCRIBE requests on every reconnection. When a subscription is set or resumed, the client is notified with the last message published to that topic. When a message has been distributed to all connected subscribers, the broker proceeds with its elimination, unless it was designated as a retained message. In that case, the retained message is made available to new or temporarily disconnected subscribers, and is modified only upon updating. When a client connects to a broker it can also define last-will messages for publication to certain topics in the case of disconnection. This feature may be vital in industrial environments where last-will messages may be about the configuration of a critical facility left without external control.

Message format: MQTT focuses on simplicity and lightweight. Information is encoded in binary format as much as possible. For instance, the CONNECT request features a 12-byte overhead containing protocol name and version number, various flags on durability, last-will settings, etc., and a keep-alive interval to ping the socket and check the connection; and a payload featuring client ID, username and password and the body of last-will messages. The overall request typically takes about 80 bytes. The shortest message to ping the socket is as small as two bytes. The largest message can reach 256 Mbytes. There exist fourteen message types used to connect and disconnect a client from a broker, to publish data, to acknowledge receipt of data, and to supervise the connection between client and broker.

The format of the data published in a topic is free, and it is the duty of the client subscribing to that topic to parse that data. For these reasons, an active cooperation among the clients and a strict organization of the topics are key prerequisites for a successful implementation of MQTT.

Topics

In a previous unit, we saw that HTTP-REST makes use of URIs to locate resources. In MQTT, data are organized in **topics**. Likewise, in HTTP-REST a client interacts with resources through verbs such as POST and GET. The counterpart in MQTT are publish, or **PUB**, and subscribe, or **SUB**, requests, with the key differences that we have already explained.

For a number of reasons that include the lack of control over the type of data and the possibility to use the wildcards described below, in MQTT, a hierarchical layout of topics is particularly beneficial. Consider for instance a greenhouse connected to a broker and made of ten compartments where each compartment contains read-only sensors and read/write actuators, such as thermometers, hygrometers, heaters and sprinkles. Of course, the agronomist may want to organize all components in different ways as shown in the following Figure, such as sensOrAct/param/compNumb, where sensOrAct may be sensors or actuators, param may be temperature, humidity, setTemperature or setFlowRate, and compNumber may be compartment0 to compartment9, for instance, or also compNumb/sensOrAct/param, or maybe remove an intermediate level like sensOrAct. In any case, it is critical that a subscriber or publisher knows the hierarchical layout as well as the format of the data expected in each topic. Suppose for instance that a publisher wants to modify the set temperature in compartment number 4. Of course it is critical to know whether to address the PUB request to actuators/setTemperature/compartment4 or compartment4/setTemperature, etc. In particular, some brokers support a dynamic generation of topics. So, for instance, if a new compartment is added, a client may start to publish data in a new topic called actuators/setTemperature/compartment10 without prior notice. This option may be very convenient, but also very dangerous in the lack of mutual understanding and cooperation among all clients. Imagine for instance a publisher that writes data to a topic like actuators/setTemperature/compartment10 when other clients await subscriptions to publish data in a topic compartment10/setTemperature.

A careful design of the hierarchical layout of topics is important also for the use of so-called *wildcards*. In MQTT, there exist two kinds of wildcards denoted by the symbols *+* and *#*. The symbol *+* denotes all items in a certain hierarchical level, while the symbol *#* means all items from there on. Suppose for instance that a client wishes to monitor all values of all sensors in compartment number 4. Then it may subscribe to sensors/+/compartment4. If it wanted to review the configuration of the actuators over the entire greenhouse, it may subscribe to actuators/#. In complex architectures, the ease to use wildcards may depend on the hierarchical structure set in the beginning, and its subsequent revision may be a substantial burden.

For a complementary overview over MQTT topics, watch the following tutorial video by MQTT provider HiveMQ.

MQTT Essentials - Part 6 | MQTT Topics & Best Practices

Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before use.

MQTT: QoS

MQTT provides a basic selection for the **Quality of Service** (QoS) managed by the broker on request from the client. The higher the level of the desired QoS is, the more dependable the transaction is, but the amount of data exchanged between the client and the broker is larger.

- **QoS0** means that the message reaches its destination at most once, because the data is transmitted one time without any acknowledgment, or *fire&forget*. QoS0 may be enough in cases where the loss of some data may be tolerable, or the underlying network is very reliable. Consider for instance a client that subscribes to the temperature probed in a compartment of a greenhouse that receives updates from another client once a minute. The loss of some updates may be irrelevant.

- **QoS1** ensures that the message reaches its destination at least once, by the use of acknowledgments. QoS1 may be ideal in cases where QoS0 may be too risky. Consider for instance a client that needs to write to the set temperature of a heater in a compartment of a greenhouse.
- Finally, **QoS2** provides that the message is received by its addressee exactly once, through the implementation of a four-step handshake. This quality may be needed in particular contexts, such as, for instance, toggling an actuator on or off like an alarm or a sprinkler in a compartment of a greenhouse. Guaranteeing the goal of QoS2 without a very reliable socket is generally very hard. In some cases, setting up a feedback system though additional sensors and topics might remain the best solution.

In spite of its light weight, the possibility to ask a certain QoS makes MQTT an application-layer protocol of the highest level for the IoT. In the next unit, we recapitulate a critical comparison between HTTP-REST and MQTT.

For a complementary overview over MQTT QoS, watch the following tutorial video by MQTT provider HiveMQ.

MQTT Essentials - Part 7 | Quality of Service Levels

Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT

Exercise

Suppose that NASA has sent a robot to Mars which is equipped with sensors streaming geolocation and other scientific data, and with motors to walk forward and turn left or right. The robot is somehow supervised from Earth through MQTT, and controlled by the use of a pad featuring three buttons, i.e. walk a step ahead and turn half a right angle left or right.

Which of the following considerations would make more sense?

☐ QoS0 may be ideal to optimize the use of bandwidth in such a remote situation, especially if the geolocation data are accurate.
☐ QoS2 is crucial, because the quality of the connection is likely to be poor in such a remote situation.
☐ QoS1 is the best choice in order to allow the robot to move faster, as it is the only option compatible with more than one trigger signal at a time.
☐ The quality of service is irrelevant in such a remote situation, where any interference with other radios is extremely implausible.

Comparison of HTTP-REST and MQTT

The table below summarizes the main differences between HTTP-REST and MQTT.

Table: list of the main differences between HTTP-REST and MQTT.

	HTTP	**MQTT**
Architecture	Pull-only client/server	Publish/subscribe client/broker
Stateness	Stateless: servers keep no info on clients. Clients may use cookies for their recognition	Brokers are aware of subscriptions and connection state of clients
Syntax and data	Text-based. Hypertexts are typically embedded as HTML files, or XML and JSON format	Mostly binary format. Payloads may be published in any format though

Location of data	A hierarchy of URIs often presented as hyperlinks and pointing to specific resources	A hierarchy of topics containing data in any format

Overall, the HTTP-REST technology is particularly suitable for clients as browsers managing a human interface, whereas the MQTT system is more specifically oriented towards M2M communication. What is the best solution for the IoT is not a simple question, because the answer depends on the specific application of interest, whether it requires extensive use of external web services or not, what is the underlying transport, how reliable and expensive it is, what is the size of the intended data, etc. In addition, please be aware that the table above only provides a rough schematization, and that there exists a broad variety of technologies intended to implement different behaviours .

For instance, let us focus in particular on the concepts of server and broker. We have described servers as a main character of HTTP serving clients through a pull-only approach, and brokers as the most distinctive feature of MQTT routing messages among publishers and subscribers. The fact that HTTP is pull-only represents a substantial limitation for the IoT.

However, there exist different ways to implement or simulate a **push/pull** behaviour in the framework of HTTP, where an HTTP server is, or seems to be, able to unsolicitedly update messages to an HTTP client.

Pushlets: in this technique, the HTTP server takes advantage of persistent sessions by leaving the response to an initial request perpetually open, thus fooling the browser to remain on standby after the initial download. The HTTP server then periodically sends snippets of typical JavaScript code to update the requested webpage. The main drawback of this method is the lack of control over the HTTP client timing out.

Long polling: Long polling is not a true push technology. It is a variation of traditional polling that emulates a push mechanism, where the HTTP client issues request messages without expecting the HTTP server to immediately respond. If the HTTP server is not ready to update any content, instead of sending an empty response, it holds the request message on standby. Once new information is available, the HTTP server immediately responds to the HTTP client, thus completing the open session. Upon receipt of this response, in turn, the HTTP client immediately issues another request. In this way, the usual latency associated with normal polling is removed.

In addition, there are possibilities to bridge HTTP and MQTT, so that a client may use HTTP, and another one MQTT within the scope of the same application. In a later unit within the section Hands on with Microcontroller Units, we will demonstrate a practical application of this concept.

Another important remark is that there exist many other protocols available for the IoT over the Internet, and many more for communication over other networks such as Bluetooth, LoRaWAN, ZigBee, etc. Here, we have decided to focus on HTTP, CoAP and MQTT as instructive cases that, at the time of writing, probably cover the majority of current applications of the IoT over the Internet.

Services

Introduction

An IoT service, often also referred to as a platform, is a multi-layer technology used to manage, automate and control the connected devices. In other words, an IoT service helps to bring the physical objects online, providing several functionalities useful to develop an IoT network. It typically refers to the facilities provided by third-party servers or brokers. The nature and scope of such services may be very heterogeneous and broad. There exist free services, paid services, large-scale infrastructures, enterprise-level services, or prototype-friendly services designed for hobbyists, for instance. Some services perform only one specific function, while others allow multiple solutions or even the design of custom sub-services. At the time of writing, the development of web services for IoT applications is a very dynamic direction of technological innovation. For this reason we will just present a few general concepts and mention an arbitrary selection of examples without claims to be exhaustive.

Here too, IoT services add another level above the Application Layer mentioned in the Internet protocol suite, which is, however, hardly independent of all underlying options. For instance, a web service that requires HTTP, and does not support more lightweight push/pull-based alternatives like MQTT, may imply a larger consumption of bandwidth and power. Likewise, if the Data-Link Layer entails the use of a low-power wide area network as LoRa or Sigfox, the web service may need the ability to hook into the relevant network through a gateway.

The best starting point to undertake the selection of an IoT service is to list the features needed in a particular application. As we shall see in the next unit, in most cases, these requirements are likely to fall into a handful of general categories, such as storing, retrieving and processing data from sensors, coordinating communication among sensors and actuators, exchanging information with third-party websites like a weather forecast API, receiving configuration changes or even firmware updates, and providing a human interface.

Functions

Basic functions

We can classify the basic functions offered by web services in three main categories:

- Managing the data produced by the things and providing a human interface;
- Implementing communication among devices and managing configuration changes;
- Interacting with the Internet.

The first and obvious feature of an IoT service is the ability to **retrieve data** from the IoT nodes. It is the very core of the IoT: smart things sending data to some remote listener. This listener may well be another object, but, in any case, if we are using a web service, it must be able to listen for data coming from peripheral nodes. This means that a common feature of every IoT service will be the ability to receive data from the things. It follows that the capability to **store data**, typically in a time-stamped database, is often another key feature of an IoT platform. The IoT nodes can produce large amounts of data but, often, they cannot store them, especially in constrained contexts. The IoT paradigm provides for a network delivery of data produced at edge level. And, if the data will be received somewhere over the Internet, they will be stored there as well. Not only: remote storage of data is also the first and necessary step in order to join or merge the raw information produced by the IoT nodes and to process it, in order to work out more complex information.

The ability to **process data** is therefore another key feature of an IoT service and, as we will see in a later unit, some platforms are specifically born around this purpose. The power of a remote server to handle data of large size, heterogeneity and complexity is the link between the IoT concept and other iconic features of the Fourth Industrial Revolution, such as the fields of big data analytics and artificial intelligence, which will be mentioned in the next units.

Usually, IoT services also provide a way to **access data**, either in the form of a user interface, and / or as an API that other things or apps can use. In particular, most of the IoT services provide a way for the things to deal with human users, i.e. they offer a user interface (even if “owner interface” would be a more appropriate phrase), such as a dashboard. In some cases, the use of native notifications may be a useful option to inform the user of some event, even when the relevant app is in the background or not running.

This last option, in particular, introduces another distinctive job for IoT services, i.e. to **allow and coordinate communication** among peripheral nodes and, more in general, other devices on the IoT network. Communication among devices is fundamental in complex networks, and allows things to take local actions after some remote decision. IoT networks indeed are not always “read-only”, as one may be led to think when picturing the things as “sensors” producing data. Often the things work as “actuators” rather than sensors, thus giving the IoT infrastructure the ability to modify the physical world.

Let us think about a predictive maintenance network, for example: the sensors in the network measure something related to the health of a plant or a process, and send data to a remote service

that will elaborate them to extract some parameters useful to understand how close an eventual fault condition may be. This information can now be used by a human decision-maker. But it can also be directly consumed to trigger some automated maintenance or setting in the process, or even to safely stop it in order to prevent a harmful break. In this quite simple and popular context, the communication among devices is fundamental for a successful implementation of the IoT network.

Another very common and important role met in communication is to ensure that the devices in the IoT network are up-to-date. The ability to **manage configuration settings**, credentials, and even firmware updates at the device level is a great challenge to be handled by human users. Unsurprisingly, many web services provide an administration panel as well as a REST API for configuring and maintaining things.

Finally, another fundamental feature of an IoT service is to **interact with the Internet**. Internet is the main world of the IoT, and being able to interact through the Internet is necessary not only to share data and information among things, but also to use data and information coming from the outside of the IoT network. In this way, the IoT platform can integrate virtually any information available on the Internet with that produced by its sensors. The information is generally integrated by using an API, and it is even possible to exploit this functionality to connect multiple IoT services together. Many of the available IoT services support this functionality by enabling the creation of "event listeners", where specific conditions trigger a so-called "webhook" in the form of a REST call to a third-party server.

Big data analytics

In this and the next unit we focus on two concepts of web services providing extensive functionality for processing data with unprecedented added value, which have become key cornerstones of the Fourth Industrial Revolution, i.e. big data analytics and artificial intelligence.

Big data analytics refers to processing massive and heterogeneous amounts of data. Over the years the availability of data has steadily been rising. And the advent of the IoT revolution has consolidated this trend with the release of ever more cheap and numerous information-sensing machines, such as mobile devices, aerial or remote sensors, software logs, cameras, microphones, radio-frequency identification (RFID) readers and wireless sensor networks. According to this paper from Science, the per-capita capacity to store information has roughly doubled every 40 months since the 1980s. Already in 2012, 2.5 exabytes (2.5×10^{18} bytes) of data were generated every day. According to IDC White Paper – #US44413318, the global data-sphere will reach 175 zettabytes (175×10^{21} bytes) by 2025.

Big data analytics is the process of collecting and analyzing large volumes of data, in order to extract hidden information. For instance, in association with sophisticated tools for business analysis, the exploitation of big data in business intelligence may disclose new insights into market conditions and

customer behaviours, thus making the decision-making process more effective and faster, and overcoming the limitations of traditional. Big data analytics includes technologies designed to discover hidden patterns and connections between data of various origin and nature.

Typical challenges in big data analytics include capturing data, data storage, analysis, search, sharing, transfer, visualization, querying, updating, information privacy and data source. Big data analytics was originally associated with three key concepts: volume, variety, and velocity. The current usage of the term tends to refer more to the use of predictive analytics and user behaviour analytics, i.e. a predictive capability to use historical and statistical data in order to model what will befall in the future or to prescribe the best conditions for a certain event to happen, and may apply to different scales ranging from a particular business to societal challenges like the prediction of economic crises, epidemics, the dissemination of opinions, the distribution of economic resources, or needs for mobility.

A peculiar feature of big data analytics is the integration of heterogeneous or unstructured data, which requires an innovative approach with respect to traditional database management systems, and calls for software architectures designed to manage large volumes of information and capable of parallel processing on cluster systems.

The main current objectives of big data analytics are:

- Lower costs: to reduce the cost of managing and analyzing large volumes of data.
- Higher speeds: to produce results in a short time, towards real time analyzes.
- Higher accuracy: to integrate large amounts of data for more accurate models.

The combination of these objectives collectively targets the possibility to anticipate the future with the knowledge of the data collected in the past, and to identify business and societal trends, such as new earning opportunities or macroeconomic issues.

Artificial intelligence

The field of artificial intelligence is very broad and interdisciplinary. The origin of its modern formulation dates back to the 1950s, and its development has gone through ups and downs ever since. At present, it is one of the most hopeful and ground-breaking pillars of the Fourth Industrial Revolution.

In a nutshell, the traditional problems of artificial intelligence include functions that belong to its natural counterpart like reasoning, knowledge representation, planning, learning, natural language processing, perception and the ability to move and manipulate objects. General intelligence is its most

ambitious and ultimate objective, although its exact definition remains a matter of philosophical debate. Current approaches include statistical methods, computational intelligence, and traditional symbolic problems drawn from the fields of computer science, information engineering, mathematics, psychology, linguistics, philosophy, and many others. Among the most important tools, we mention algorithms for search engine optimization, mathematical optimization, artificial neural networks, and methods based on logic, statistics, probability and economics.

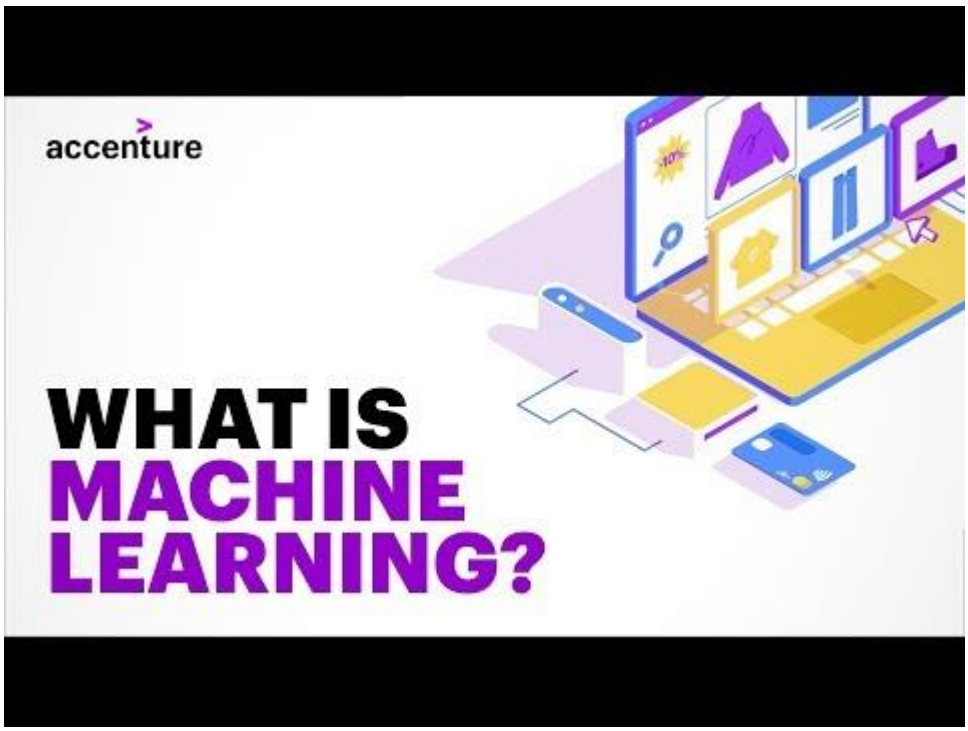
Artificial intelligence pursues the ability of an automatic system to correctly interpret external data, to learn from such data, and to use those learnings to achieve specific goals and tasks through flexible adaptation. Many algorithms are capable of learning from data, and to enhance themselves by learning heuristics derived from trials that worked well in past iterations, or even to write other algorithms. The earliest approach to artificial intelligence was formal logic based on rules like "If an animal walks on four legs, then it may be a cat". A second, more general, approach is Bayesian inference, where instructions take a form like "If the animal under study walks on four legs, then adjust the probability that it is a cat in such-and-such way". The third major approach makes use of analogizers, such as nearest-neighbour tools like "After examining the records of past animals with number of legs, size, colour, and other factors closely matching the individual under study, such a percent of those turned out to be cats". A fourth approach is artificial neural networks that take inspiration from the architecture of natural brains. These networks make use of layers of computational neurons capable to adjust their mutual connections, by identifying and iteratively reinforcing those links that provide more likelihood to output the desired response. These four main approaches can overlap with each other and with evolutionary systems or other methods. For instance, neural networks can learn to make inferences, to generalize, and to make analogies.

Among the most distinctive features of artificial intelligence is the concept of **machine learning**. Machine learning is the study of computational algorithms that automatically improve through experience towards a certain objective, and may be unsupervised, supervised or reinforced. Unsupervised learning is the ability to find patterns in a stream of inputs, such as images of cats and dogs, without requiring a human programmer to label the data first. Supervised learning includes both classification and numerical regression, which requires a human programmer to label the inputs in a training session. Classification is used to determine the category of something, such as cats and dogs, and takes place after the artificial agent has seen a number of examples from all categories of interest. Regression is the attempt to produce a function that describes the relationship between certain inputs and outputs, in order to predict the values of the outputs from those of the inputs. Both classifiers and regression learners can be viewed as function approximators trying to identify an unknown relationship. In reinforcement learning, the system is rewarded for good responses and punished for bad ones, and uses this sequence of rewards and punishments to form a strategy for operating in its problem space. In all cases, machine learning tries to replicate the dynamics of its natural counterpart. The programmer focuses on the efficiency of such processes, rather than the definition of

prescriptions for a particular problem that may require thousands of lines of code, such as classifying images from cats and dogs.

By letting the artificial agent discover the correlations between inputs and outputs on its own, this approach represents a very powerful solution to exploit data in a broad variety of high-level contexts, such as autonomous vehicles like drones and self-driving cars, medical diagnostics, creating art, proving mathematical theorems, playing games like Chess or Go, search engines, online assistants, image recognition in photographs, spam filtering, predicting flight delays, prediction of judicial decisions, targeting online advertisements, and energy storage, just to name a few.

AI 101: What is Machine Learning? | Accenture



Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before use.

Exercises

Question #1

Which of the following features is a prototype-friendly service like the Arduino IoT Cloud likely to provide?

☐ An MQTT broker for coordinating communication among peripheral nodes.
☐ Big data analytics to integrate massive amounts of heterogeneous data .
☐ Artificial intelligence for automatic speech recognition .
☐ A driver for a stepper motor.

Question #2

How are web services positioned with respect to the Internet protocol suite?

☐ They sit between the Presentation and Session Layers.
☐ They actually sit an extra layer above the Internet protocol suite, at the level of final user applications.
☐ They are Data Link Layer protocols.
☐ They offer free or paid services to the Data Link Layer.

Question #3

Certain businesses like Particle, offer complete edge-to-cloud solutions providing advanced functions like over-the-air firmware updates (see https://docs.particle.io/tutorials/device-cloud/ota-updates/). What may be the most serious challenge for a web service offering this feature?

☐ Speech recognition, which requires support for a large number of national languages.
☐ Costs, as the human resources committed to over the air firmware updates may scale with the number of target devices .
☐ Bandwidth, because over the air firmware updates may consume too many computational resources.
☐ Security, because malicious updates issued by a spoofing hacker may jeopardize a large number of connected devices in no time.

Question #4

Watch the following video by Reviews.org: The Customizable Way to Automate Your Home | IFTTT Review which provides a critical review on IFTTT. Which of the following statements may be true?

☐ IFTTT may make use of so-called webhooks, in order to connect two web applications through custom callback functions.
☐ IFTTT may be used as data logger for simple IoT applications, such as home and garden automation.
☐ IFTTT endows simple devices, such as lightbulbs and sprinklers, with wireless connectivity.
☐ IFTTT is a so-called man-in-the-middle hacker, forging the data exchanged between two web applications like Instagram and Twitter.

Co-funded by the
Erasmus+ Programme
of the European Union

Providers and examples

IoT service providers

Here, we will take a quick journey through a few of the most popular providers of IoT services, at the time of writing. We will outline their main functionalities and characteristics without the claim of completeness.

The most famous and used IoT services at the end of 2020, or, we may say, at the dawn of the IoT era, include the following:

- Google Cloud Platform
- IRI Voracity
- Particle
- Salesforce IoT Cloud
- ThingWorx
- IBM Whatson IoT
- Thinger.io
- Arduino IoT Platform
- Microsoft Azure IoT Suite
- Amazon Web Services IoT Core
- Samsung Artik Cloud
- Oracle IoT
- Cisco IoT Cloud Connect
- Altair SmartWorks
- Plotly
- Temboo
- Carriots
- NearBus
- Ubidots

Almost all of them are proprietary, commercial services. Some are more oriented to makers or hobbyists, some to business with a more general purpose or a more specific application in mind. In general, IoT services may be classified according to their principal scope as:

- Analytics and data visualization, where the main objective is to efficiently collect, analyze and visualize data;
- Prototype-friendly IoT infrastructure, where it is to easily coordinate communications among edge devices;
- End-to-end solutions, where it is to seamlessly manage a proprietary system;
- Large scale infrastructure as a service, where it is to offer highest-level functions, often at the expense of the ease of access.

Let us see some of them in some more detail in the following paragraphs.

Thinger.io

Thinger is an open source platform for the IoT.

Thinger is a FOSS (Free and Open Source Software) system distributed under MIT license, which a user can download directly from the project repository and install on a personal server while keeping everything on hand. They also offer a commercial IoT service through the Thinger.io website.

Thinger uses a third party cloud provider. At the moment of writing, the user can choose among Amazon Web Services, Digital Ocean, Google Cloud and Microsoft Azure.

Things can be connected to the platform using the provided Arduino Client Library, which is specifically designed for use with the Arduino IDE described in the next session. Therefore, the user can install it within a familiar environment and seamlessly start connecting devices within minutes.

It supports multiple network interfaces like Ethernet, Wifi, and GSM, and the user can implement it in several devices like most of the Arduino boards, but also Espressif, Texas Instruments and other chips (an exhaustive list can be found on the service website).

The main features of Thinger are:

- Free IoT platform: Thinger.io provides a lifetime freemium account with only a few limitations to start learning and prototyping. When a product becomes ready to scale, the user can deploy a so-called Premium Server with full capacities within minutes;
- Open-source: most of the platform modules, libraries and APP source code are available in a Github repository to be downloaded and modified with MIT license;
- Simple but powerful: just a couple of lines of code to connect a device and start retrieving data or controlling its functionalities with a web-based console able to connect and manage thousands of devices in a simple way;
- Hardware agnostic: any device from any manufacturer can be easily integrated with Thinger.io infrastructure;
- Extremely scalable and efficient infrastructure: thanks to its unique communication paradigm, where the IoT server subscribes device resources to retrieve data only when it is necessary, a single Thinger.io instance is able to manage thousands of IoT devices with low computational load, bandwidth and latency.

Arduino IoT Cloud

Arduino IoT Cloud is a powerful service, allowing anyone to create IoT applications with just a few simple steps. With a combination of smart technology, user-friendly interfaces and powerful features, this service is fit for students, makers, as well as professionals. We will come back to the Arduino ecosystem in the next session.

The most interesting features of Arduino IoT Cloud platform are:

- Directly linked to the Arduino Create environment to program a broad variety of boards;
- Automatically generated code for integration in Arduino and compatible boards;
- Ideal for building sensor networks;
- Ideal for real-time data monitoring;
- Wi-Fi and LoRa compatibility , among others;
- Building dashboards with a good selection of widgets;
- Creating custom apps by using the Arduino IoT API.

Arduino IoT Cloud allows many methods or protocols of interaction, including HTTP REST API, MQTT, Command-Line Tools, Javascript, and Websockets. It is a very versatile and dynamic system. For more details, check their API documentation.

INDUSTRIAL EXPERT

Dash and Plotly

Dash is a productive Python framework for building web applications.

Written on top of Flask, Plotly.js, and React.js, Dash is ideal for building data visualization apps with highly customised interfaces in pure Python. It is particularly suited for anyone who works with data in Python.

Through a couple of simple patterns, Dash abstracts away all the technologies and protocols that are required to build an interactive web-based application. Dash is simple enough that a user of Python can bind a new interface around a code in an afternoon.

Dash apps are rendered in the web browser. The user can deploy apps to servers and then share them through URLs. Since Dash apps are viewed in the web browser, Dash is inherently cross-platform and mobile ready.

There is a lot behind the framework. To learn more about how it is built and what motivated Dash, watch their talk from Plotcon or read their announcement letter.

Dash is an open source library, released under the permissive MIT license.

Plotly develops Dash and offers a platform for managing Dash apps in an enterprise environment. Plotly.js is a high-level, declarative charting library. Plotly.js ships with over 40 chart types, including 3D charts, statistical graphs, and SVG maps. Plotly is now a part of Dash distribution.

Google Cloud IoT

Google Cloud IoT offers a platform for intelligent IoT services based on Google Cloud, a suite of cloud computing services that runs on the same infrastructure that Google uses internally for its end-user products, such as Google Search, Gmail, file storage, and YouTube.

Intelligent means that they stress the data-processing capabilities of the platform. Google Cloud IoT indeed is a complete set of tools to connect, process, store, and analyze data both at the edge and in the cloud. The platform consists of scalable, fully-managed cloud services; an integrated software stack for edge/on-premises computing with machine learning capabilities for all IoT needs.

From an organizational point of view, the IoT service offered by Google Cloud can be separated in two main parts:

- Cloud IoT Core - Secure device connection and management service for the IoT;

- Cloud IoT Edge - Brings AI to the edge computing layer.

Although you can have a free test account, the service is quite expensive.

IRI Voracity

Voracity is a high-performance, all-in-one data management platform.

Like other services listed in this section, Voracity is not a specific tool for the IoT, but a data-oriented platform that is intended to provide a full stack of tools to control data in every stage of their life-cycle and extract maximum value from them.

Voracity combines data discovery, integration, migration, governance, and analytics in a managed metadata framework built on Eclipse™. Leverage the proven power of IRI CoSort, Hadoop MR2, Spark, Spark Stream, Storm, and Tez.

Frontier cases

In the real world, the border between basic and more advanced functions is sometimes blurred, and so is that among the realms of new technologies like big data analytics and artificial intelligence, and even the interplay of tasks implemented at edge level and services deployed in the cloud, within a so-called computational continuum.

Watch the following two videos.

In the first video, Dr Sara Colantonio, of the National Research Council of Italy, provides an overview over her activities in the context of predictive maintenance in healthcare, where she makes extensive use of big data analytics in association to artificial intelligence.

In the second video, Dr Davide Moroni, again of the National Research Council of Italy, explains different projects in the realm of the so-called smart city, where he combines the powers of artificial intelligence at edge and cloud level according to the specific context. Notice how the use of other emerging technologies like drones is becoming another powerful enabler for futuristic IoT applications.

INDEX expert interview: Dr Sara Colantonio

Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT INDEX

INDEX expert interview: Dr Davide Moroni

Co-funded by the
Erasmus+ Programme
of the European Union

Hands on with microcontrollers

Introduction to microcontrollers

Introduction to Microcontroller Units (MCUs)

In this unit, we begin to describe the use of microcontrollers in appliances that are suitable for integration in IoT systems. Microcontroller Units (MCUs) are electronic devices integrated in a single electronic circuit that evolve as a downgrade from microprocessors for applications in embedded systems or digital control.

While microprocessors are typically found in personal computers, MCUs are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems. By optimizing the size and cost with respect to an architecture made of a separate microprocessor, memory and input/output units, MCUs make it economical to implement digital control in a broad variety of devices and processes. Mixed signal MCUs are common to integrate analog components needed to interface to non-digital electronic systems, such as simple transducers. In the context of IoT, MCUs have become the most economical and ubiquitous means to collect data, sense and actuate the physical world at the level of edge devices.

Some MCUs may use words as small as four bits and operate at frequencies as low as 4 kHz to achieve power consumption at the level of single-digit milliwatts or even microwatts. MCUs generally provide the ability to retain functionality while waiting for an event, such as an internal timer overflow, a button press, the reception of data over some communication link, or any other kind of interrupt. During sleep mode with processor clock and most peripherals off, power consumption may be in the order of nanowatts, thus making this technology well suited for long lasting battery applications. On the other hand, other MCUs may serve performance-critical roles, and may need to act more like a Digital Signal Processor (DSP) or to execute and schedule multiple tasks, with higher clock speeds and power consumption. Indeed, current trends move towards both lower and higher performances, and a sharp distinction between MCUs and microprocessors is probably becoming ever more arbitrary. An important example of an intermediate case between MCUs and PCs is single-board computers like the uber-famous Raspberry Pi and a growing plethora of similar devices.

Table: indicative comparison between MCUs and microprocessors.

Feature	MCUs	Microprocessors

Maximum clock speed	200 MHz	4 GHz
Maximum processing capacity	200 megaFLOPs	5 gigaFLOPs
Minimum power loss	1 mW	50 W
Typical price per piece	0.5 euros	50 euros
Number of pieces sold per year	Order of 10 billion	Order of 1 billion

The typical architecture of MCUs provides a set of fixed modules, and a series of possible extensions depending on the manufacturer, price and range of application.

Common modules:

- Central processing unit (CPU);
- Program memory (ROM, FLASH, EPROM, EEPROM) used to store programs and non-volatile data;
- Data memory (RAM) used to cache volatile data;
- Configurable input/output or general-purpose input/output (GPIO) ports. GPIO pins are uncommitted digital signal terminals in an integrated circuit or electronic circuit board. Their behaviour, including whether they act as input or output, is controllable by the user at run time. GPIO pins have no predefined purpose and are unused by default;
- Interrupt management to suspend ongoing processes and react to certain events with a so-called interrupt handler or interrupt service routine (ISR), such as an internal timer overflow, completing an analog to digital conversion, a change of logic level on an input like a button being pushed, and the reception of data over a communication link. Interrupt management is a distinctive feature of MCUs especially when power consumption is important as in battery devices. For instance, interrupts may serve to wake from a low-power sleep state until a peripheral event requires a reaction;
- Direct Memory Access controller (DMAC) to access the RAM without occupying the CPU.

Additional modules:

- Counters and timers associated to the internal clock;
- An ever broader variety of communication modules, such as Universal Synchronous-Asynchronous Receiver/Transmitter (USART), Inter-Integrated Circuit I^2C, Serial Peripheral

Interface (SPI), Universal Serial Bus (USB), Ethernet, Infrared Data Association (IrDA) interfaces, Controller Area Network (CAN) interfaces, Bluetooth, Bluetooth Low Energy, Wi-Fi, Zigbee and many more. The possibility for direct connectivity represents an obvious great opportunity for M2M and IoT projects, as we shall see;

- Analog or mixed technology interfaces, such as Analog to Digital Converter (ADC), Digital to Analog Converter (DAC), Pulse-Width Modulation (PWM), analog comparators, etc.;
- Displays and other control interfaces, such as LCDs and touch sensors.

Figure: STM32F103 R6T6 MCU from STMicroelectronics, CC BY-SA by Golonlutoj

In the next units, we will illustrate the use of some of these modules by means of simple examples. However, prior to the description of practical examples, we introduce the Arduino ecosystem, which has become a convenient platform for rapid prototyping of IoT appliances.

Exercise

The LoRaFarm is a LoRaWAN-Based Smart Farming Modular IoT Architecture designed by Internet of Things (IoT) Lab of the Department of Engineering and Architecture of the University of Parma, Italy, which is described in the open-access paper entitled LoRaFarM: A LoRaWAN-Based Smart Farming Modular IoT Architecture. According to the following figure, what hardware solutions may have plausibly been implemented for the various components?

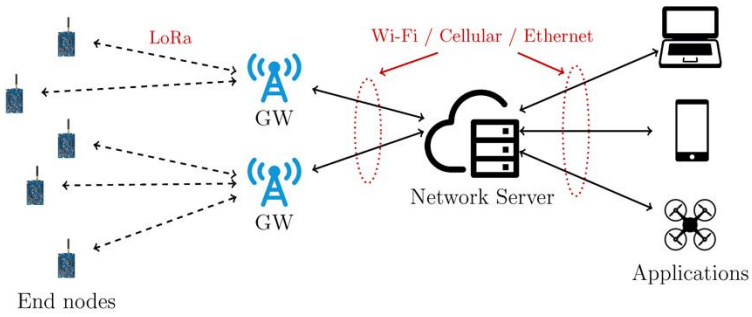


Figure: figure 1 from Codeluppi, G.; Cilfone, A.; Davoli, L.; Ferrari, G. LoRaFarM: A LoRaWAN-Based Smart Farming Modular IoT Architecture. Sensors 2020, 20, 2028., CC BY-SA by Sensors 2020, 20, 2028

Answers

Only one answer is correct.

☐ MCUs for the end nodes installed in the vineyard for their low power consumption and small footprint, mini computers for the gateways (GWs) combining LoRa and WiFi connectivity as a decent compromise between power consumption and computational power, and another MCU for the Network Server for its support to massive data storage and complex user applications.
☐ MCUs for the end nodes installed in the vineyard for their low power consumption and small footprint, mini computers for the gateways (GWs) combining LoRa and WiFi connectivity as a decent compromise between power consumption and computational power, and a standard computer for the Network Server for its support to massive data storage and complex user applications.
☐ A standard computer for the end nodes installed in the vineyard for its low power consumption, mini computers for the gateways (GW) combining LoRa and WiFi connectivity as a decent compromise between power consumption and computational power, and another standard computer for the Network Server for its support to massive data storage and complex user applications.
☐ MCUs for the end nodes installed in the vineyard for their low power consumption, mini computers for the gateways (GW) combining LoRa and WiFi connectivity as a decent compromise between power consumption and computational power, and a standard computer for the Network Server for its low cost and small footprint.

Arduino

Hardware and shields

The next units of this session will focus on the Arduino ecosystem as a convenient starting point to see first-hand the use of MCUs in IoT projects. We refer to this platform for its clear vocation for educational training and for its lively community of enthusiastic users. There exists a large number of contributed resources and projects based on the Arduino system on the Internet, but, on the flip side of the coin, we already warn the trainee that not all of those are of the same quality. However,

most of the concepts developed hereafter are applicable to a broad variety of competitive products as well.

We begin our journey into the Arduino ecosystem by introducing the overall design of its hardware and principal modules. The Arduino hardware consists of a versatile platform made up of a series of electronic boards equipped with a MCU. The project was conceived in 2005 by some members of Ivrea Interaction Design Institute in Italy as an open-source set of tools for rapid prototyping for hobby, educational and professional purposes. The name originates from that of Arduino d'Ivrea, King of Italy in 1002.

The Arduino system collectively pursues the quick and easy development of small devices integrating multiple components, such as LEDs, speed controllers for motors, light sensors, automatisms for temperature and humidity control and many more, within projects that couple sensors, actuators and communication. The hardware part combines with a simple Integrated Development Environment (IDE) for programming, which will be the focus of the next unit. All software and circuit diagrams are free.

At the time of writing, most Arduino boards consist of an Atmel 8-bit AVR MCU with varying amounts of flash memory, pins, and features. Single or double-row pins or female headers are used to facilitate connections for programming and incorporation into other circuits. Additional circuits designed to plug-in as add-on modules are termed shields. Multiple and possibly stacked shields may be individually addressable or selectable via an I^2C or SPI serial bus, which will be covered in a dedicated unit. Most boards include a 5 V linear regulator and a 16 MHz crystal oscillator or ceramic resonator.

Arduino MCUs are pre-programmed with a boot loader that simplifies uploading custom programs to the on-chip flash memory. The default bootloader of the Arduino Uno is the Optiboot bootloader. Program code is loaded via a serial connection to another computer. Some Arduino boards contain a level shifter circuit to convert between RS-232 and transistor–transistor logic levels. Current Arduino boards are programmed via a Universal Serial Bus (USB) port that is implemented using USB-to-serial adapter chips, such as the FTDI FT232.

The most iconic Arduino board is the Arduino Uno, which is based on the Microchip ATmega328P MCU. The Arduino Uno is equipped with 14 digital GPIO pins and 6 analog I/O pins also usable as additional digital pins. 6 of the digital GPIO pins support Pulse-Width Modulation (PWM) output, which will be discussed in a dedicated unit. Program code is loaded via a type B USB cable. Power is supplied via the same USB cable or an external power supply unit or a battery pack preferably providing 7 to 12 V.

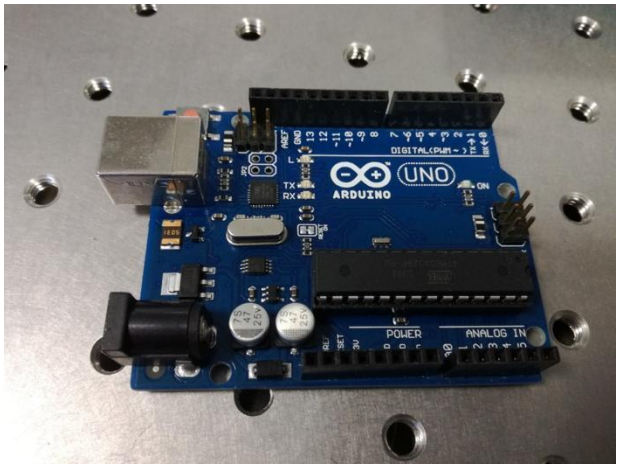


Figure: the Arduino Uno board, CC BY-SA by INDEX consortium

Technical specifications for the Arduino Uno:

- Microcontroller: ATmega328P
- Operating Voltage: 5 V
- Input Voltage: 6-20 V
- Digital I/O Pins: 14
- PWM Digital I/O Pins: 6
- Analog Input Pins: 6
- DC Current per I/O Pin: 20 mA
- Flash Memory: 32 KB (ATmega328P) of which 0.5 KB used by bootloader
- SRAM: 2 KB (ATmega328P)
- EEPROM: 1 KB (ATmega328P)
- Clock Speed: 16 MHz
- Footprint: about 69 mm × 53 mm
- Weight: 25 g

A broad variety of alternative Arduino boards are available with different CPUs, GPIO pins, footprint as well as additional features, such as sensors, a Secure Digital (SD) card slot and on-board

communication modules for Wi-Fi, Bluetooth, BLE, LoRa WAN, Sigfox, GSM and Narrowband, for instance.

The performance of Arduino boards may be specialized and enhanced by the use of printed circuit expansions called shields, which plug into the female pin headers normally supplied with the main units. Examples of shields provide direct access to high-level functionality, such as motor controls for 3D printing and other applications, GNSS (satellite navigation), Ethernet, a liquid crystal display (LCD), breadboarding (prototyping), a SD card slot, or a real-time clock (RTC) chip.

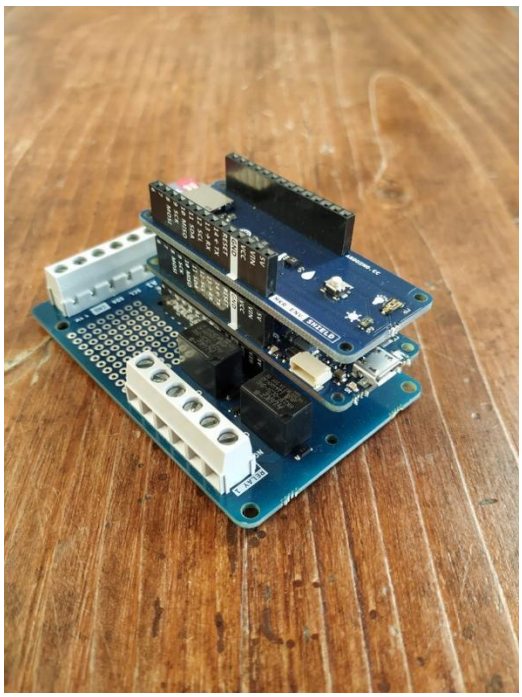


Figure: two shields stacked on an Arduino board, CC BY-SA by INDEX consortium

After introducing the principal components found in the Arduino hardware, in the next unit, we briefly outline the pathway needed to develop and upload program code that gives control over the behaviour of its GPIO pins.

Software

In the previous unit, we gave an overview of the principal hardware components available in an Arduino board, such as the most iconic Arduino Uno. Here, we introduce the issue to develop and upload program code into one such board, in order to gain access and control over the behaviour of its GPIO pins.

In principle, a program for Arduino hardware may be written in any programming language with compilers that produce binary machine code for its target processor. Atmel provides Atmel Studio, which is a development environment for their 8-bit AVR and 32-bit ARM Cortex-M based MCUs.

However, the most popular and straightforward approach to program Arduino boards is to exploit the Arduino Integrated Development Environment (IDE) made available within the Arduino project.

The Arduino IDE is available as both an offline software as well as a more recent online Arduino Web Editor.

The original offline software is a cross-platform application for Windows, macOS, and Linux that is written in Java, and originated from the IDE for other languages, such as the Processing and Wiring projects. It includes standard features for basic IDEs, such as a code editor with utilities such as text cutting and pasting, searching and replacing text, automatic indenting, brace matching, and syntax highlighting, and provides simple one-click mechanisms to compile and upload programs to an Arduino or compatible board. It also contains a message area, a text console, a toolbar with buttons for common functions and a hierarchy of operation menus, as well as a serial monitor and plotter that is a convenient interface to send and receive messages from a connected board.

The Arduino IDE supports program code written in C and C++ using special rules for structuring, and inherits software libraries from the Wiring project to perform common input and output procedures. A useful reference to the functions, kinds of variables and structures available within the programming language can be found at https://www.arduino.cc/reference/en. Furthermore, the community has developed a broad variety of additional libraries that greatly facilitate the implementation of specific and common tasks, such as M2M communication via standard serial protocols, driving various types of motors, and many more. A list of official libraries can be found at https://www.arduino.cc/en/Reference/Libraries. Homemade contributed libraries are available across many websites and are not always of the same quality. As we shall see from the next unit on, user-written code needs to contain at least two basic functions, for starting the routine and for cycling through the main program loop. These functions are compiled and linked with a main () stub into a cyclic executive program with the GNU toolchain. The Arduino IDE employs the AVRDUDE utility to convert the executable code into a hexadecimal text that is loaded into the Arduino board by a bootloader that is present in its firmware.

The online variant of the Arduino IDE is an innovative tool that is part of Arduino Create available at https://create.arduino.cc/, which is an online platform designed to enable developers to write code, access tutorials, configure boards, and share projects in a continuous workflow. The Arduino Web Editor provides all functionalities of the offline software, and saves programs in the cloud and ensures the most up-to-date version of the IDE, including all contributed libraries and support for new Arduino boards. However, at the time of writing, support is limited to official Arduino boards, and

certain limitations exist on the total number and size of saved programs and the sum of compilation time consumed per day.

Figure: Arduino Web Editor, CC BY-SA by INDEX consortium

The Arduino Web Editor comes as a workspace divided into three columns. The leftmost column shows a list of main menu items:

- Sketchbook is a collection of user-written programs ready for compiling and uploading into a board connected via a USB cable. Note that programs in the Arduino ecosystem are called sketches, and we will adhere to this convention hereafter;
- Examples are read-only sketches that demonstrate basic commands, and the behaviour of libraries;
- Libraries are packages containing functional blocks of program code that can be included in sketches to provide extra functionalities at higher levels of abstraction. We will present examples of libraries in the next units;
- Monitor is a feature that enables users to receive and send data to their boards via the same USB cable used to upload sketches;
- Help provides helpful links and a glossary about Arduino terms;
- Preferences are options to customize the look and behaviour of the editor.

The central column shows lower-level options associated with the menu items listed in the leftmost column.

Finally, the rightmost column is the so-called code area. Although it does not require much explanation, this part is the place where most users are likely to spend most time editing, verifying,

building and uploading their code into their boards, saving sketches on the cloud, and sharing their projects with the community. The entire workflow is streamlined for rapid and easy prototyping.

In the next session, we begin to show a simple example of program code that illustrates the general structure of a sketch for Arduino.

Programming Arduino for basic tasks

Sketches and the blink example

With this unit, we start a series of examples designed to illustrate the execution of basic and common tasks with an Arduino board, and to show the components needed for their implementation. Premise that the scope of this series is not a formal introduction into C/C++ nor the optimization of solutions in terms of performance, but a tutorial overview over the power and limitations of MCUs. In most cases, examples will consist of a set of hardware parts mounted on a solderless breadboard, and sketches that dictate the behaviour of one or more Arduino boards. A sketch is a program code written with the Arduino IDE and saved on the development computer or the cloud as a text file with file extension .ino.

The compiler needs the Arduino C/C++ program to contain at least two basic functions:

setup(): This function is called only once as soon as a sketch starts or restarts after power-up or reset. It is often used to initialize variables, the use of GPIO pins as input or output terminals, and libraries included in the sketch.

loop(): After the setup() function exits or ends, the loop() function executes repeatedly until the board is powered off or reset. It is as if this function was embedded in a for(;;) infinite loop.

We start to illustrate the structure of an Arduino sketch with the blink example, which is the most typical program used by beginners to test the use of their Arduino boards. In practice, it is akin to the Hello, World! example in many other programming ecosystems without a physical interface. Most Arduino boards integrate an on-board built-in LED connected through a current limiting resistor between a certain pin, i.e. digital pin 13 in the Arduino Uno, and ground. More generally , the number of the digital pin attached to the on-board LED is defined across different devices in the Arduino IDE as LED_BUILTIN. This LED is used as a convenient feature for many tests, program functions and debugging. In the blink example, the sketch repeatedly turns this LED on for a certain number of milliseconds and then off for another number of milliseconds, by the use of functions provided by internal libraries included in the Arduino IDE, such as pinMode(), digitalWrite(), and delay().

Here is a reformulation of the blink example:

```
boolean ledState = LOW;    // Define a variable to store logical level of pin attached to built-in LED, and initialize it to LOW, i.e. ground

int onTime = 1000;    // Define a variable to store time in ms for built-in LED to stay on…

int offTime = 2000;    // … And another variable to store time in ms for built-in LED to stay off


void setup()  // This function is called only once at the beginning of program execution

{

   pinMode(LED_BUILTIN, OUTPUT);  // Define digital pin number LED_BUILTIN as output

}


void loop()  // This function is repeatedly called during program execution until power off or restart

{

   digitalWrite(LED_BUILTIN, ledState);  // Set digital pin number LED_BUILTIN to logical level stored in variable ledState

   if (ledState == HIGH)  // If it is high…

      {

   delay(onTime);  // … Wait for time established for built-in LED to stay on

      }

   else  // Otherwise, here if it is low…

      {
```

```
  delay(offTime);  // … Wait for time established for built-in LED to stay off

  }

ledState  =  !ledState;  // Flip logical level stored in variable ledState, i.e. from ground to +5 V or the other way around

}
```

When this sketch is loaded on an Arduino board, its on-board built-in LED will turn on for 1 sec, then off for 2 sec, on again for 1 sec, etc., until the USB cable is removed. Upon reconnection of the USB cable or plugging of the Arduino board to a wall socket or battery supply, the program will restart from the beginning, i.e. not from the point left at the time of disconnection.

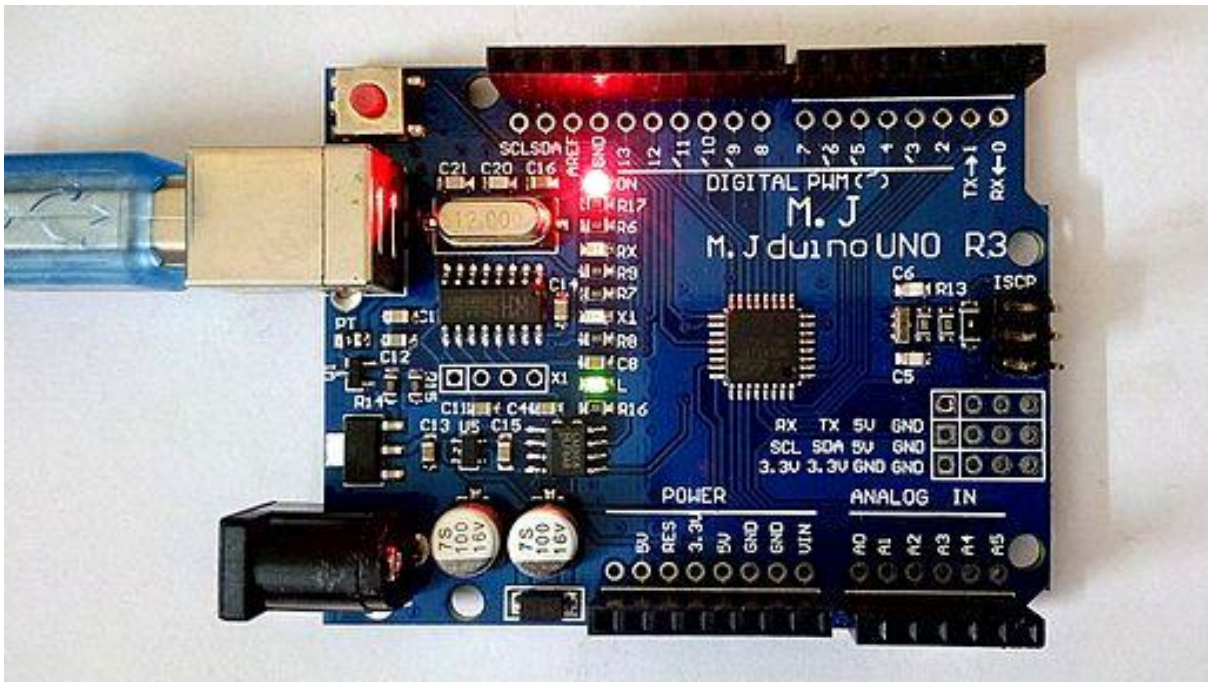


Figure: power LED (red) and programmable LED attached to pin 13 (green) on an Arduino-compatible clone, CC BY-SA by Rajib Ghosh

In the next units we will show more complex examples based on Arduino that are intended to give a general idea of the hardware and software components needed to implement basic and ubiquitous tasks in digital electronics. However, before we go on, let us open a parenthesis on the simulation of electronic circuits.

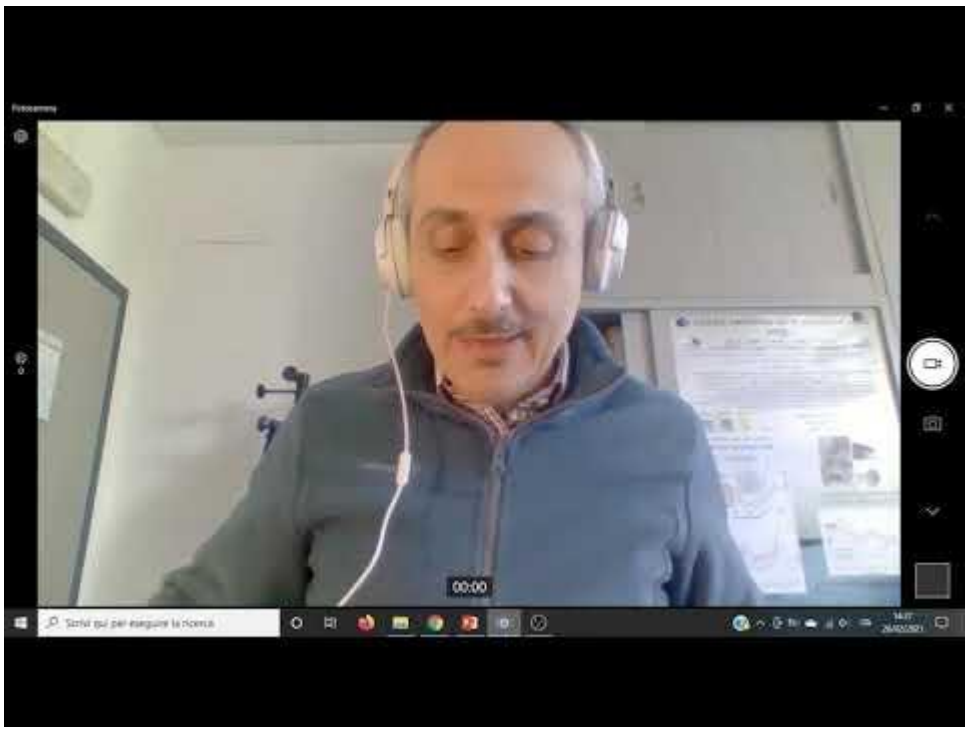
INDUSTRIAL EXPERT

Simulation of electronic circuits

Sometimes, before starting the physical construction of electronic circuits, it is convenient to make numerical simulations of their assembly and behaviour. The next video illustrates the use of an educational tool for the simulation of electronic circuits, such as Autodesk Tinkercad, which is available as a free program running on a web browser at https://www.tinkercad.com/ .

We recommend the trainee to familiarize with the use of numerical simulations for practice. Most of the examples described in the next units are suitable for implementation in an environment like Tinkercad!

Simulation of Simple Circuits with Tinkercad



Disclaimer: this is an embedded video originally posted on YouTube and complying with Standard YouTube License. If you want to re-use it, you should check the video owner's copyright terms and conditions before use.

To clarify further, here is the video script:

Hello everybody! My name is Fulvio Ratto and I am a material scientist at the National Research Council, Italy… and… well, a wannabe maker! Well, in our day-to-day work in our labs, it just happens that we need to set up small prototypes of electronic devices, such as this mini Wi-Fi camera for time-lapse movies, for instance. And, sometimes, before we start and assemble our physical devices, it is

just convenient to make numerical simulations of their construction and behaviour. So today we are talking about the numerical simulation of electronic circuits.

There are many tools out there to simulate electronic circuits. Some are more professional, some are maybe less. But today I will introduce an option that is extremely easy, surprisingly powerful… and highly educational, which is Tinkercad. Tinkercad is a free suite of online programs that run on a web browser, such as Google Chrome. It is owned by Autodesk, which is the supplier of AutoCAD… you know, the super famous software for computer-aided design. But I believe that its name is a little bit… misleading, because Tinkercad includes tools for computer-aided design, yes… but also much more. Such as the possibility to simulate electronic circuits, indeed.

So, let us get started. Let us open a web browser like Google Chrome, and let us type: www.tinkercad.com, so that will be www.tinkercad.com. First things first, you will need to create an account, which is super standard, so I am not spending time on that. Then, once you are logged in, as I am already… Here on the left hand side, you will see a list of tools. You see that my interface is in Italian. So I tend to assume that yours must be in your national language. Hey… Here is circuiti, or circuits. Let us click on circuits. And once you go to circuits, you will find a list of your past projects, which will be empty if it is your first time in Tinkercad. You see that I already have one project. And here is the place where you can rename, remove or resume your past projects whenever you come back to Tinkercad. Just try yourself.

But let us create a new project now… By clicking on create a new circuit... something like that. Every change that we will make to the new circuit will be automatically saved in the list of past projects. So this will be our workspace. And here on the right hand side, there is a list of components that you can choose, and a prompt that you can use to search. So, say I want a dc motor… That will be motore cc in Italian… Oh here it is! And you can just drag and drop your component into your workspace. Say… something more nerdy… like a shift register… oh my, what is shift register in Italian… Registro something… oh cool registro a scorrimento an 8 bit. Here it is. And of course you can move or delete your components. Try yourself. Everything is so intuitive.

So… say we want to add an MC unit like an Arduino Uno board. Micro… Cool. You see that the present choice is kind of limited. There is only one MC unit or maybe two, which is the Arduino Uno and the BBC micro:bit for the kids. So, if you need to work with an MC unit that you cannot map into an Arduino Uno, you are a little bit out of luck with Tinkercad at this time, and you may need something else. But we are using the Arduino Uno for most of our course. So let us go on. Another piece of hardware that is ubiquitous is a breadboard… Breadboard… So for those of you who are not familiar with breadboards… let me briefly come back to myself.

So here is a breadboard, which is a piece of hardware to solderlessly assemble your electronic circuits. You see that there is plenty of holes that you can use to plug your components. For instance, I have a transistor here. Or a capacitor. What you need to remember with breadboards is that the holes in

each of these semi-columns, as well as the holes in each of these longer rows are connected together. Let me show you what a breadboard looks like from the inside. Here it is. You got it, right. So, if I plug a jumper wire from this hole to this one, well now the cathode of my capacitor is connected to the gate of my transistor. I am not sure whether it makes any sense… And well it is just conventional to use these longer rows for power, and in particular these red rows for the positive leads and these blue or black ones for the negative leads. So you see that you can use two power supplies over each breadboard… like power from the Arduino Uno here… and power from a 9 V battery pack here, for instance. Like this. Alright. Back to Tinkercad.

So we have a breadboard… and what do we want to do? Say we want to have an LED and a pushbutton, and we want to make so that the LED stays on when the button is pushed. Simple. So we need an LED… here it is! A pushbutton… Pulsante in Italian… cool. Ok, what you need to know with pushbuttons is that their legs on the left end side are always connected together. Their legs on the right end side are always connected together. But their left and right halves get in contact only when the button is pushed. It is very usual to plug pushbuttons in such a way to bridge the upper and lower halves of the breadboard. Like this. Ok, now we need to wire everything up!

Guess what… It is super easy to wire things up in Tinkercad. If you need a jumper wire… Just click once in the starting hole. Once in the destination hole. And there it is! I think we can keep this one. So let us be very conventional, and let us connect the negative lead of the Arduino Uno… Or ground, to the black row of the breadboard. Let us make this wire black. The positive lead of the Arduino Uno… Or +5V, to the red row of the breadboard. This one will be red. Great! Now let me connect the right end side of the pushbutton to +5V. Red. And the cathode of the LED to ground through a resistor… Resistenza. Like that. Let us assign a standard value of 220Ω, which will ensure the ideal current limitation. You see that the stripes change accordingly. Cool.

Ok. Everything is ready. So what? You see that there is a button here labelled something like start simulation. Click it. Now we are simulating the behaviour of our circuit. And if we push the button with our mouse… the LED lights up. Pushed… Released. And we can stop and resume the simulation whenever we want. And we can even run experiments. For instance, say, let us try to modify our resistance. Say from 220Ω to… 220kΩ. You see that the LED is much dimmer. Let us try… 220mΩ, now. Wait! Now Tinkercad is telling us that we are killing our LED. So you understand why it may be useful to make numerical simulations before physical disasters! We had better come back to… 220Ω.

So now everything works fine… but we are using our Arduino Uno just as a power supply, only! Let us try to exploit the full power of our MC unit by programming its pins! Let me remove a few wires. There we go. So… Let me connect the anode of our LED to a GPIO pin… Say pin number 2, that we will use as digital output. So when we will write a LOW, or ground, to our digital output, the LED will be off. But when we will write a HIGH, or +5V, it will be on. What about the pushbutton. Let me connect the left end side to ground. Black. And the right end side to another GPIO pin… Say pin number 4… that we will use as digital input. So you see. When the button is pushed, our digital input will read a LOW.

But when it is not pushed, its right end side is actually floating, which we do not like. So let me connect the right end side of the pushbutton to +5V with a resistor… That I will call a pull-up resistor. And let me give it a high value… say… 10kΩ. So, when the button is not pushed, our digital input will read a HIGH. And when it is pushed, a little current will flow through this local loop, and almost all voltage will drop across our pull-up resistor, so our digital input will read a LOW. But we are not talking physics today.

Fine. Let us try our simulation. Well nothing happens. Well something is actually happening. As you may have noticed, there is a built-in LED here that is blinking. And it was already blinking before. Why? Because our Arduino Uno is pre-programmed with its original firmware that makes the built-in LED attached to pin number 13 to blink. So we need to replace the original firmware with our own sketch. And wait! Hey, There is a button here labelled something like code. Click it. And here is the original firmware. By default, Tinkercad shows this language made of blocks… which is Microsoft MakeCode, I guess… and is not particularly familiar to me. So let me go to text. Oh, here it is! The good old blink example! You see there is a setup function and a loop function. But we need to set our own rules, right?

Let us start with a few definitions. So… int ledPin =… what was it? 2; and… int buttonPin =… 4; right? In the setup function, let us define the directions of our GPIO pins. So pinMode(ledPin, OUTPUT); and… pinMode(buttonPin, INPUT); fine. Next, the loop function. So let me remember… we want that when the button pin reads a LOW, the LED pin sets to HIGH, and vice versa, right? So in practice we want to write a logical value to our LED pin that is the opposite of that read by the button pin. So it will be… digitalWrite(ledPin, !digitalRead(buttonPin)); and let me keep a small delay… 10 ms maybe [delay(10);] which will serve to avoid overwhelming the MC unit. Cool, let us try. Pushed. Released. Pushed. Released. Great. Like before, but now we are running a sketch.

And this can give a lot more flexibility. For instance, before I finish, let me show you a tool that is very popular in the Arduino ecosystem, which is a serial monitor. You see, when you are in the code… You can open a serial monitor. And even a serial plotter. For those of you who are not yet familiar with the Arduino ecosystem, a serial monitor is a tool to exchange serial messages… in the form of characters… between… well in this case our virtual board and… yes, our real monitor, through our virtual USB cable. So, in the setup function we need to initiate our serial communication. Serial.begin(… and we can set a standard baud rate to… 9600); Hz. And in the loop function, for instance, we can print… to the serial monitor… the current state of the LED. Serial.println(… this value [digitalRead(ledPin)]); done. Let us start our simulation. You see. There are values flowing in the monitor… and points drawn in the plotter. Pushed. Released. Pushed. Released. Pushed. Released. Great! Enough for today!

Let me just recap. I have shown you how to use Tinkercad to simulate the construction and behaviour of simple circuits starring an Arduino Uno. And I have even taken the opportunity to spend a few words about some common components, such as a breadboard or a pushbutton, that you will meet in this course. Well. Just let me finish by telling you that Tinkercad is really powerful and funny… and

you must love it if you are interested in the Internet of Things. There is a lot more, like the possibility to include some common libraries. Unfortunately you cannot load your own, at present. Or the possibility to add and program multiple boards… and simulate M2M communication, for instance… you know, things that may become quite expensive in the physical world… So, I hope you enjoyed this video and please consider to try a tool like Tinkercad for your own practice! So… Thanks for watching and good-bye!

Digital inputs and a push button example

In this unit, we introduce the concept of digital input and propose a relevant example for the sake of clarity. A digital pin configured to behave as digital input, upon interrogation, returns the boolean value HIGH or LOW whether the attached voltage is closest to logical value high, i.e. +5 V in an Arduino Uno, or ground. When the digital pin is left to float, its reading may flip at random or depend on the configuration of other pins and onboard components, the proximity of capacitive elements including the human body, etc.

In this example, digital pin 2 of an Arduino Uno connects to one terminal of a push button or a tactile switch. The other terminal of the push button connects to ground. When the push button is pressed, digital pin 2 is grounded and the reading is LOW. In order to make sure that the reading is HIGH in the opposite case, digital pin 2 is set to connect to an internal pull-up resistor. The figure below displays the hardware needed in this example.

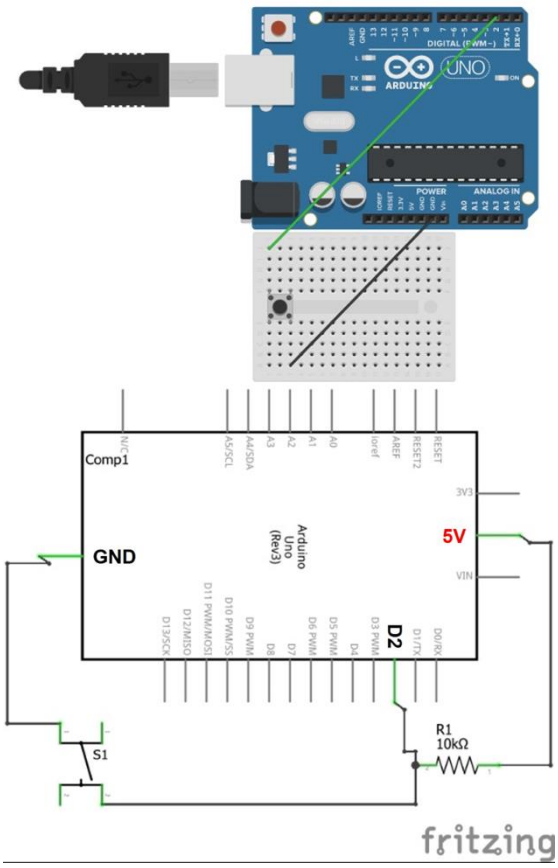


Figure: hardware for the push button example and relevant diagram explaining the role of the push-up resistor, CC BY-SA by INDEX consortium

The sketch will ensure that pressing the push button toggles the state of the built-in LED. In practice, the initial state of the built-in LED is off. A single push turns it on. A second push turns it off again etc.

This use of a push button or a tactile switch raises a common problem, because immediately after its push or release, the state of a digital pin may bounce multiple times between the logical states HIGH and LOW before stabilization. The time needed for stabilization depends on the quality of the push button and may easily extend up to around 50 ms. In the case of our push button example, the impact of bouncing may be catastrophic. A fortuitously odd number of bounces would result in the desired swap, but an even number would amount to no swap at all. As far as the swap concerns a built-in LED in an Arduino board, the user may just settle and try more times. But in other contexts, such as alarm or security systems etc., one such problem is unacceptable. Bouncing is a common issue, and efficient debouncing is a typical challenge in digital electronics.

The sketch below is meant to provide the desired functionality and presents a representative solution for debouncing. This simple sketch already contains different features that belong to C++ programming language, such as the use of functions or conditional constructs as well as referencing and dereferencing to memory addresses of variables. A thorough understanding of these features is not required for the scope of these units, which is to give an idea of the typical components and lower-level issues implied in basic projects based on Arduino. However, the interested reader is encouraged to use external resources to learn more about programming Arduino boards:

```
/*
 * Debounce sketch
 * A push button connects to pin 2 and controls the built-in LED
 */

bool ledState = LOW; // current state of the built-in LED initialized to logical level LOW
const int buttonPin = 2; // number of the digital input pin attached to push button
const int debounceDelay = 100; // delay in ms set to assess stabilization. Change as needed
bool buttonState; // variable to store consolidated state of push button
bool currentReading; // variable to store current instant reading of push button
bool previousReading; // variable to store last instant reading of push button
long int lastBounceTime; // variable to store last timepoint in ms when instant reading of push button changed

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT); // set pin LED_BUILTIN as digital output
```

```
    pinMode(buttonPin, INPUT_PULLUP); // set pin buttonPin as digital input and activate internal pull-up resistor

    previousReading = digitalRead(buttonPin); // initialize last instant reading

    lastBounceTime = millis(); // initialize last timepoint in ms when instant reading of push button changed as current timepoint

    buttonState = previousReading; // initialize consolidated state of push button as last instant reading

}


void loop()

{

    pollButton(buttonPin, debounceDelay, &ledState); // call function pollButton defined below. This function takes buttonPin, debounceDelay and the memory address of ledState as parameters, and practically dictates the behavior of the push button and what variable it toggles. A thorough discussion on the use of pointers to addresses of variables goes beyond the scope of this module. Please gloss over the exact meaning of this construct

    digitalWrite(LED_BUILTIN, ledState); // Set pin LED_BUILTIN according to state stored in ledState

}


void pollButton (int pin, int delay, bool *state)

{

    currentReading = digitalRead(pin); // record current instant reading

    if (currentReading != previousReading) // if it is different from last instant reading...
```

```
  {
    lastBounceTime = millis(); // ... reset last timepoint in ms when instant reading of push button changed
  }
  if (((millis() - lastBounceTime) > delay) && (currentReading != buttonState)) // if instant reading of push button is unchanged for long enough, and it differs from consolidated state of push button...
  {
    buttonState = currentReading; // ... set it as consolidated state of push button, and...
    if (buttonState == LOW) //... if it is low, i.e. if push button is pressed...
    {
      *state = !(*state); // ... toggle value of variable stored at specified memory address
    }
  }
  previousReading = currentReading; // set last instant reading as current reading. Here in any case at end of any loop
}
```

The experimenter may find that a lag a 100 ms may provide too slow a response and a poor experience. The optimization of the time set to assess stabilization may depend on the actual quality of the push button and the application of interest. 10 ms may be a more proportionate choice in most cases.

The instructions designed for debouncing are wrapped in a function named pollButton() that takes the number of the pin attached to the push button as parameters, the time in milliseconds needed for stabilization, and the memory address of the global variable toggled upon pushing, and is called once in the loop() function. Any sketch must at least define the setup() and loop() functions, but

nothing prevents the user to add more functions, which is oftentimes an excellent idea to better structure a complex code.

In the next unit, we introduce another key feature of Arduino boards, which is the possibility to parse analog inputs.

Exercise

Try to simulate the electronic circuit described in this unit by the use of a tool like Tinkercad. Now try to attach the push button to GPIO pin number 4 rather than 2.

Which statement is correct?

☐ The same behaviour may be obtained by replacing code line “const int buttonPin = 2;” with “const int buttonPin = 4;”, only.
☐ The same behaviour may be obtained by replacing code line “const int buttonPin = 2;” with “const int buttonPin = 4;”, and another choice of debounceDelay, because bouncing primarily depends from pin to pin.
☐ It is impossible, because LED_BUILTIN is internally attached to GPIO pin number 2
☐ It is impossible, because GPIO pin number 4 cannot serve as a digital input.

Programming Analog inputs and a voltage divider example

In this unit, we introduce the analysis of analog inputs, which are signals that present as a continuous scale of voltage typically spanning between ground and the operating level of the MCU. Typical examples of analog signals relate to simple sensors that transduce some physical quantity, such as humidity, pressure or temperature, into a scale of voltage through some responsive material.

The analysis of analog signals in digital electronics poses some challenges. Arduino boards are usually equipped with a 10-bit Analog to Digital Converter (ADC) that samples and quantizes an analog value between ground and their operating voltage into a linear scale from 0 to 1023. Since the operating voltage of an Arduino Uno is 5 V, the nominal resolution of its ADC is about 4.9 mV. This feature is accessible in a certain number of analog pins numbered as A0 to A5 in an Arduino Uno. Their analog reading returns an int between 0 and 1023.

In order to illustrate this feature, we propose a simple project that returns a nonlinear indicator of environmental light intensity as percent of output signal with respect to limits calibrated by the user. The hardware shown in the figure below makes use of a photoresistor, which is a low-cost active component made of semiconducting material that decreases in resistance with received light intensity. Examples of use may be to assess the proximity of a shadowing object, such as a hand for a quick proof, the intensity of artificial or sunlight, opening or closing of a box or a drawer, etc. In the

dark a photoresistor can have a resistance as high as several megaohms. Instead, in full light its value can drop to as low as a few hundred ohms. The typical technique used to read the resistance R1 of a variable resistor is a voltage divider, where one end of the photoresistor connects to +5 V and the other to ground through a fixed-value resistor of resistance R2. Then, the voltage at the intersection between both resistors reads V = 5 V*R2/(R1+R2) and can be sampled by connecting to an analog pin, such as A0. In the case of a photoresistor, a reasonable value for R2 is in the order of 10 kΩ, so that V approaches 0 in the darkness and +5 V in full light.

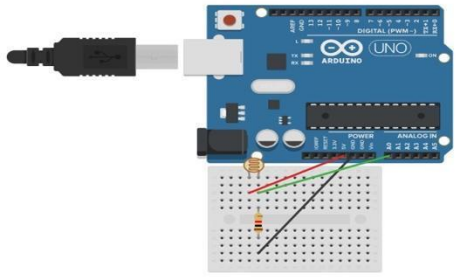


Figure: hardware for the voltage divider example, CC BY-SA by INDEX consortium

The sketch below provides for a calibration of the sensor in the setup() function, where the user is left with 5 sec to simulate the lowest and highest levels of light intensity, e.g. by shadowing the photoresistor by hand. Then the loop() function reads and converts the analog signal as percentage of the calibrated stroke, and calculates the value of the resistance of the photoresistor, and prints both values to the serial monitor via the USB cable:

```
/*
 * Photoresistor sketch
 * Maps the range of analog values from a photoresistor to scale from 0 to 100 and print the percent to the serial port
 */

const int sensorPin = A0; // select the input pin for the photoresistor
int sensorVal; // variable to store the analog reading
```

```
int lowestVal = 1023; // variable to store the lower limit of the analog reading in the calibration step. Initialize this variable to highest value and let the calibration routine to decrease it by and by

int highestVal = 0; // variable to store the higher limit of the analog reading in the calibration step. Initialize this variable to lowest value and let the calibration routine to increase it by and by

int percent; // variable to store analog signal as percentage of calibrated stroke

float R1; // value of photoresistor in kOhm

float R2 = 1.0; // value of fixed resistor in kOhm


void setup()

{

// first, initialize serial communication through the USB cable and set baud rate to 9600 Hz

  Serial.begin(9600);

// calibrate the sensor for the first 5 sec after program start

  Serial.println("Start calibration for 5 sec"); // print this message to the serial monitor

  while (millis() < 5000) // until 5 sec of program start

  {

    sensorVal = analogRead(sensorPin); // store analog reading in the appropriate variable

    if (sensorVal > highestVal) // if analog reading is larger than current higher limit, which will be the case on program start

    {
```

```
        highestVal = sensorVal; // update higher limit
      }
      if (sensorVal < lowestVal) // if analog reading is smaller than current lower limit, which will also be the case on program start
      {
        lowestVal = sensorVal; // update lower limit
      }
    }
    Serial.println("End calibration"); // here after 5 sec of program start. Print this message to the serial monitor
}

void loop() {
    sensorVal = analogRead(sensorPin); // store analog reading in the appropriate variable
    percent = map(sensorVal,lowestVal,highestVal,0,100); // percent will range from 0 to 100 according to a linear conversion of the analog reading. Should the analog reading be smaller than the lower limit set in the calibration step, e.g. due to a poor calibration, its value will be set to 0. Seemingly, should the analog reading be larger than the higher limit set in the calibration step, its value will be set to 100.
    // Next instructions print percentage to the serial monitor
    Serial.print("Reading is ");
    Serial.print(percent);
    Serial.println("% of stroke");
```

```
    R1 = (1023.0 - (float)sensorVal)/(float)sensorVal*R2;  // calculate resistance of photoresistor as a function of floating-point values

    // Next instructions print value of photoresistor in kOhm

    Serial.print("Resistance of photoresistor is ");

    Serial.print(R1);

    Serial.println(" kOhm");

    Serial.println();

    delay(100);  // wait 100 ms before another run

}
```

The experimenter may want to modify the time left for calibration according to the particular application of interest. Of course, we emphasize that a procedure based on single lowest and highest records is weak against any source of noise fluctuations, and so any actual application would certainly require some statistical analysis. Also in this case, we have exploited tools such as the map (value, fromLow, fromHigh, toLow, toHigh) function that the interested reader is encouraged to study by the use of external resources. Other features, such as the use of the serial monitor and plotter, will be shortly resumed later in this series of units.

In the next unit, we introduce the use of digital outputs in a simple example.

Exercise

Try to simulate the electronic circuit described in this unit by the use of a tool like Tinkercad. Now try to set the resistor in the breadboard to 1 MΩ rather than 1 kΩ, without any other change.

Which statement is correct?

☐ The reading for R1 is actually in MΩ instead of kΩ, and the sensor loses sensitivity especially at higher brightness.
☐ The reading for R1 is actually in MΩ instead of kΩ, and the sensor gains sensitivity especially at higher brightness.
☐ The compiler returns an error message, because the reading for R1 is actually in MΩ instead of kΩ.

☐ The simulator returns a warning, because the sensor goes out of range, and so too much current is drawn from the power pin.

Digital output and a bar graph example

Another key feature of Arduino boards is the possibility to use its digital and analog pins as voltage source that may be set at logical value LOW, i.e. ground, or HIGH, i.e. +5V in the Arduino Uno. In particular, the Arduino Uno is able to supply up to 20 mA of DC current per digital pin, which is sufficient to power a variety of interesting actuators, such as LEDs, buzzers and small servo motors, or to control external circuitry via transistors, optocouplers etc.

The next example shows a simple use of a bar graph made up of an array of LEDs that indicate the position of a potentiometer. A potentiometer is a ubiquitous case of variable resistor that works as voltage divider in many applications, where the position of its thumbwheel or slider may regulate the volume of a buzzer, the brightness of a liquid crystal display or serve as rotation transducer, for example, in a joystick. In the hardware shown in the figure below the central terminal of a potentiometer divides the voltage drop of 5 V into two parts, and the partition is read as an analog signal. Then, the sketch provides for the activation of a number of LEDs proportional to the analog reading. In particular, an array of 6 LEDs are assembled with common anode attached to +5 V. Instead, their cathodes are individually addressed via digital pins 2 to 7 through as many 220-Ω resistors, which ensure the appropriate limitation to the current flowing across each diode. Note that digital pins 0 and 1 serve the serial port that is often reserved for communication with the computer through the USB-to-serial adapter chip, and so their use for other purposes is a second choice.

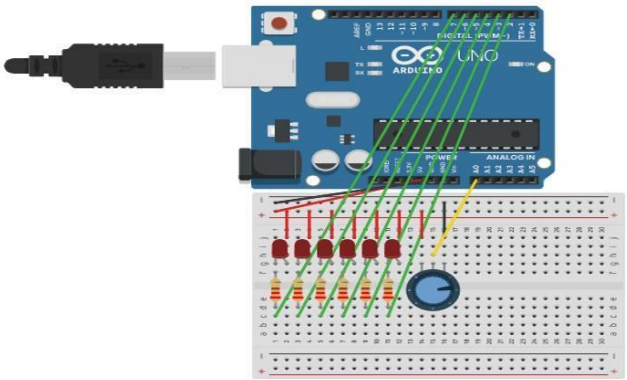


Figure: hardware for the bar graph example, CC BY-SA by INDEX consortium

/*

```
* Bargraph sketch
* Turns on a series of LEDs proportional to the value of an analog sensor.
* Six LEDs are controlled in this example
*/

const int ledPins[] = {2,3,4,5,6,7}; // array of int containing the labels to the digital pins attached to the LEDs

const int noOfLeds=sizeof(ledPins)/sizeof(ledPins[0]); // number of elements included in the array. Note that the function sizeof(variable) returns the number of bytes occupied by the array or variable, and that each int occupies two bytes in the Arduino Uno

const int potPin = A0; // analog pin attached to the potentiometer

// Swap values of the following two constants if cathodes are connected to gnd and anodes to the digital pins

const bool LED_ON = LOW; // setting the digital output to LED_ON will turn LED on

const bool LED_OFF = HIGH; // setting the digital output to LED_OFF will turn LED off

int potVal; // variable to store the analog reading

int ledLevel; // variable to store the number of LEDs to turn on


void setup() {

  for (int led = 0; led < noOfLeds; led++) // for each numeral from 0 to noOfLeds...

  {
```

```
    pinMode(ledPins[led], OUTPUT); // ... set the pin of the number stored in that position of ledPins as digital output
  }
}

void loop() {
  potVal = analogRead(potPin); // store analog reading in the appropriate variable
  ledLevel = round(potVal*noOfLeds/1023.0); // calculate the number of LEDs to turn on
  for (int led = 0; led < noOfLeds; led++) // for each numeral from 0 to noOfLeds - 1...
  {
    if (led < ledLevel ) // ... if that numeral is less than the number of LEDs to turn on...
    {
      digitalWrite(ledPins[led], LED_ON); // ... set the corresponding pin to LED_ON, which will turn the attached diode on
    }
    else // otherwise, if that numeral is equal or more than the number of LEDs to turn on...
    {
      digitalWrite(ledPins[led], LED_OFF); // ... set the corresponding pin to LED_OFF, which will turn the attached diode off
    }
  }
```

```
    delay(100);  // wait 100 ms before another run
}
```

The style of this sketch makes it super easy to modify the choice of LEDs used in the bar graph by solely editing, deleting or adding elements into the array of int containing the numbers of the relevant pins, i.e. line const int ledPins[] = {2,3,4,5,6,7}; . All subsequent instructions are self-consistent. The use of arrays is another key feature of C++ programming language that we show without additional explanation. Another variable that the experimenter may want to optimize is the number of milliseconds left from cycle to cycle according to line delay(100); which governs the responsiveness of the bar graph.

In the next unit, we will illustrate the counterpart to function analogRead(pin) in the context of outputting voltage and current, which is analogWrite(pin, value).

Exercise

Try to simulate the electronic circuit described in this unit by the use of a tool like Tinkercad. Now try to modify the code line “if (led < ledLevel);” as “if (led == ledLevel);”.

Question

Which statement is correct?

☐ Only one LED lights up at a time, or none above a certain threshold, and its brightness is the same as before.
☐ Only one LED lights up at a time, or none above a certain threshold, and, in most cases, its brightness is greater than before, because there is less load connected in series.
☐ Only one LED lights up at a time, or none above a certain threshold, and its brightness is proportional to the value set by the pot.
☐ The simulator returns a warning, because there is no LED to light up above a certain threshold.

Analog output and a DC motor example

Pretty much as in the case of inputs, many applications may require an analog output to control the functionality of an actuator, such as the intensity of a monochromatic LED, the color of a RGB LED, the speed of a motor, etc. Most Arduino boards are not natively equipped with the natural counterpart to their ADC, which is a Digital to Analog Converter (DAC). However, other techniques make it possible to obtain or simulate an analog output. One possibility may be the use of a potentiometer for manual adjustment of output voltage. But this solution is likely to be incompatible with many projects based

on digital actuation. The main method to replicate an analog output with an Arduino board is pulse-width modulation (PWM), which is the simulation of an analog level through the duty cycle of a train of square-wave digital pulses, i.e. the ratio of ON time with respect to sum of ON plus OFF times. For instance, with an operating voltage of 5 V, a duty cycle of 50% would simulate an output voltage of 2.5 V, a duty cycle of 20% one of 1 V, etc.

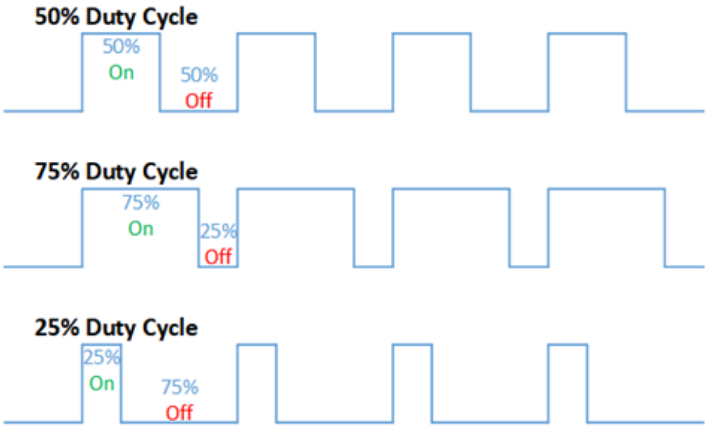


Figure: explanation of duty cycle for pulse-width modulation, CC BY-SA by INDEX consortium

In an Arduino Uno, the preset value for the frequency of PWM is about 500 Hz, which corresponds to a full cycle of ON and OFF times of 2 ms. Such an interval is much faster than the retention time of images in human retina, i.e. around 60 ms, or the inertia of typical motors, so that PWM represents an effective alternative to a true analog output in many projects. In Arduino boards, the duty cycle can be set anywhere from 0 for 0% to 255 for 100%, which is a preset resolution of the internal handler around 7.8 µs.

The next example shows the use of this technique to control a DC motor. The circuit is outlined in the figure below . We take this opportunity to mention a few issues that are very common in digital electronics, and in particular the use of moving parts: that the power drawn from an electrical load as a DC motor may exceed that available in each pin of an MCU, and that its operation may challenge the stability and integrity of the electronic circuitry. The first problem is usually solved by the use of a transistor or an optocoupler, so that the MCU controls a gate, but the electrical load draws power from an external circuitry that may include a power supply, such as a battery pack. In addition, DC motors typically require extra care, which is the use of a capacitor that biases the energy drawn when the coil sets in motion, and a diode that protects the transistor when deceleration causes dangerous counter-tensions. All these components are sometimes packaged into drivers made available for the

application of choice. For instance, the Arduino project supplies a so-called Arduino Motor Shield to plug onto an Arduino Uno and drive inductive loads such as relays, solenoids, DC and stepping motors.

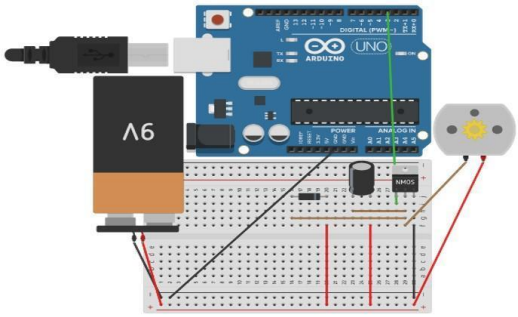


Figure: hardware for the DC motor example, CC BY-SA by INDEX consortium

The sketch gives access to the speed of the DC motor as percent of maximum through the serial console.

```
/*
 * DC motor sketch
 * Commands from the serial port control motor speed: type messages like 80 to set speed at 80% of maximum. Any other key stops motor
 */

const int motorPin = 3; // motor driver connects to digital pin 3
long percent = 0; // variable to store speed as percent of maximum
char ch; // variable to store each byte received from serial port
long dutyCycle; // variable to store duty cycle of motor driver

void setup()
```

```
{
   Serial.begin(9600);  // initialize serial communication
   Serial.println("Enter percent of duty cycle");  // print instructions to the serial monitor
}

void loop()
{
   while (Serial.available())  // until bytes are received through the serial console
   {
      ch = Serial.read();  // consume and store byte received
      if(isDigit(ch))  // if byte received is a number
      {
         percent = percent*10+ch-'0';  // accumulate byte received in the appropriate variable
      }
      delay(1);  // leave time to comply with set baud rate
      if (!Serial.available())  // if there is no more byte left. Here if final byte left was consumed by last call to the Serial.read() function
      {
         dutyCycle = min(percent*255/100, 255);  // map and store percent as duty cycle from 0 to 255 for PWM. Make sure that duty cycle never exceeds 255, for instance, should user have accidentally typed something like 120
         // Next instructions print duty cycle to the serial monitor
```

```
            Serial.print("Duty cycle = ");

            Serial.println(dutyCycle);

            analogWrite(motorPin,  dutyCycle);  // set analog output to control motor
driver...

            percent  =  0;  // ... and reset variable used to accumulate each byte received
from the serial console

        }

    }

}
```

Of course, the experimenter may try to design a different cipher to control the DC motor, such as the use of alphabet letters, etc.

After mentioning fingerprint features in the Arduino ecosystem such as functions pinMode(pin, mode), digitalRead(pin), analogRead(pin), digitalWrite(pin, value) and analogWrite(pin, value), and before we come to the topic of machine to machine communication and IoT with Arduino boards, in the next module, we open a brief parenthesis on a ubiquitous tool as libraries.

Exercise

Try to simulate the electronic circuit described in this unit by the use of a tool like Tinkercad. Now try to replace the 9 V battery with a 3 V coin cell.

Which statement is correct?

☐ The DC motor will go slower, on average, because its power supply will be less.
☐ Nothing happens, because the power supplied to the DC motor depends on the digitalWrite instruction only.
☐ Nothing happens, because the capacitor is there to compensate for the loss of power supplied to the DC motor.
☐ The DC motor will go slower, on average, unless code line “dutyCycle = min(percent*255/100, 255);” is changed into “dutyCycle = min(percent*765/100, 765);”, which ensures the adequate compensation over the entire range from 0 to 100%.

Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT

Libraries and another debouncing example

A theoretical definition of library in computer science goes well beyond the scope of this unit, which is to show what a simple library may look like and how it may be used within a sketch. Without pretension of formal rigor, a practical definition of library may be that of a chunk of program code and additional resources, such as documentation, examples, etc. made available within a community of users, which may be included in any sketch to gain access to higher-level functions, methods and behaviours.

A list of standard Libraries in the official Arduino distribution can be found at https://www.arduino.cc/en/Reference/Libraries:

- EEPROM – for reading and writing to non-volatile storage (1 kB in the Arduino Uno);
- Ethernet – for connecting to the internet using the Arduino Ethernet Shield, Arduino Ethernet Shield 2 and Arduino Leonardo ETH, at the moment of writing;
- Firmata – for communicating with applications on the computer using a standard serial protocol;
- GSM – for connecting to a GSM/GPRS network with the GSM shield;
- LiquidCrystal – for controlling liquid crystal displays;
- SD – for reading and writing SD cards;
- Servo – for controlling servo motors;
- SPI – for communicating with devices using the Serial Peripheral Interface (SPI) bus, which will be covered in the next unit;
- SoftwareSerial – for serial communication on any couple of digital pins;
- Stepper – for controlling stepper motors;
- TFT – for drawing text, images, and shapes on the Arduino Thin-Film Transistor screen;
- WiFi – for connecting to the I nternet using the Arduino WiFi shield;
- Wire – Two Wire Interface (TWI/I2C) for sending and receiving data over a net of devices or sensors, which will be covered in the next unit.

Many more libraries are available for download through the Arduino IDE and from dedicated websites. A library typically consists of a library.cpp text file containing the definition of its functions and methods written in C++, a library.h text file serving as a header listing the so-called prototypes of those functions and methods, and a folder collecting examples in the form of library.ino text files ready for upload into an Arduino board.

In the following sketch, we show how to translate the push button example seen in a previous unit in terms of program code including a library that may be readily reused in any other sketch.

Let us start with the pushButtonExample.ino text file:

```
/*
 * Debounce sketch
 * A push button connects to pin 2 and controls the built-in LED
 * Debounce logic is now included in a pushButton library
 */

#include <pushButton.h> // include the desired library code
// The rest of this sketch is almost identical to that seen in a previous unit
bool ledState = LOW;
const int buttonPin = 2;
const int debounceDelay = 100;
pushButton myButton(buttonPin, debounceDelay); // create an instance of pushButton class attached to desired digital pin and associated to required delay for debouncing, according to library definition

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
}
```

```
void loop()
{
  myButton.poll(&ledState); // use method poll with memory address of desired state that shall be toggled upon pushing attached button
  digitalWrite(LED_BUILTIN, ledState);
}
```

The pushButton.cpp text file contains the program code that sets the functions and methods of the library, and looks as follows:

```
/*
 * pushButton.cpp - Library for debouncing push buttons
 * Created by Fulvio Ratto, Aug 31, 2020.
 * Released into the public domain.
 */

#include "Arduino.h" // include standard Arduino library. This library is implicit in all .ino sketches built under the Arduino IDE
#include "pushButton.h" // this line is needed to import the declaration of all variables, functions and methods

pushButton::pushButton(int pin, int delay) // constructor of pushButton class. These lines explains what happens when a user creates an instance of this class
{
```

```
    _pin = pin; // the _pin variable is a so-called private variable used internally within the library, and may take any name. It is just conventional to start the name of private variables with symbol _
    _delay = delay;
    pinMode(_pin, INPUT_PULLUP);
    _previousReading = digitalRead(_pin);
    _lastBounceTime = millis();
    _buttonState = _previousReading;
}

void pushButton::poll(bool *state) // definition of method named poll. This program code is identical to that used in previous push button example
{
    _currentReading = digitalRead(_pin);
    if (_currentReading != _previousReading)
    {
        _lastBounceTime = millis();
    }
    if (((millis() - _lastBounceTime) > _delay) && (_currentReading != _buttonState))
    {
        _buttonState = _currentReading;
        if (_buttonState == LOW)
```

```
        {
            *state = !(*state);
        }
    }
    _previousReading = _currentReading;
}
```

Finally, the pushButton.h file is a header that contains the definition of all variables and the prototypes of all functions and methods:

```
/*
 * pushButton.h - Library for debouncing push buttons
 * Created by Fulvio Ratto, Aug 31, 2020.
 * Released into the public domain.
 */

#ifndef pushButton_h // if there is no other library installed with same name
#define pushButton_h // define this library
#include "Arduino.h" // include standard Arduino library

class pushButton // a class is a collection of variables, functions and methods
{
```

```
  public:  // public items are called from the main sketch, and include variables, constructors and methods

    pushButton(int pin, int delay);

    void poll(bool *state);

  private:  // private items are used within the library and never become visible to the user

    int _pin;

    int _delay;

    bool _buttonState;

    bool _currentReading;

    bool _previousReading;

    long int _lastBounceTime;

};


#endif
```

At first sight, this solution may seem to amount to a very intricate alternative to achieve the same functionality as in previous push button examples. However, this library is now ready for use in any sketch and even for distribution among users, who may prefer to manage debouncing at higher level without the nuisance of writing their own dedicated program code. There are thousands of libraries out there that have undergone extensive optimization, and may represent a very efficient solution to perform common tasks. Meanwhile, the experimenter may also stumble across contributed code of lower quality, and so a word of caution is in order here.

We are aware that the example developed in this unit is so simple that its justification is hardly plausible. Its presentation was meant to illustrate the structure and use of libraries within this course, rather than to create a useful case for broader distribution. In the next units, we will make extensive use of libraries. Their power will become evident as a fundamental tool to obtain quick access to very high-level functionality without the need to delve into hundreds of lines of program code and complex logics or even to understand all subtleties of underlying processes, such as those behind communication standards. In the absence of libraries, the use of MCUs in M2M communication and IoT projects would be inaccessible to the non-highly professional specialist.

M2M communication, I2C, SPI and an example

M2M communication is a common situation in the context of applications based on MCUs. Arduino boards are fit to implement different protocols for M2M communication.

Serial communication is often used for communication between an Arduino board and a PC, but M2M communication among more boards via the serial port is possible as well. All Arduino boards comprise at least one serial port, also known as Universal Asynchronous Receiver-Transmitter (UART) or Universal Synchronous-Asynchronous Receiver/Transmitter (USART). On the Arduino Uno, the pins devoted to serial communication are number 0 for reception (RX) and 1 for transmission (TX). Communication between Arduino boards and a PC occurs through an onboard USB/TTL serial converter chip and a USB cable. The Arduino IDE features a built-in serial monitor for bidirectional interaction with an Arduino board, which we have already exploited in several previous examples. The sketch below provides another simple example of communication between an Arduino Uno and a PC through the serial monitor. When the PC sends any char, the Arduino Uno checks whether it is a digit and returns the corresponding number of random integers from 1 to 6 and their sum, and so behaves as a digital dice for use, for instance, to play Chutes and Ladders:

```
/*
 * Digital dice sketch
 * Generates and print random numbers to the serial port
*/
char ch; // variable to store received character
int number; // variable to store random number
```

```
int tot = 0; // variable to store cumulative number


void setup()
{
  Serial.begin(9600); // initialize serial communication and set baud rate to 9600 Hz
}


void loop()
{
  if(Serial.available()) // if any byte is received through serial port...
  {
    ch = Serial.read(); // ... consume and store byte received in appropriate variable
    if (isDigit(ch)) // if it is a number...
    {
      for (int i = (ch-'0'); i > 0; i--) // ... for each numeral counting down from that number to 1
      {
        number = random(6)+1; // generate and store a random number from 0+1 = 1 to 5+1 = 6
        tot += number; // accumulate number and update appropriate variable
        Serial.println(number); // print random number to serial monitor and start a new line
```

```
        }
        Serial.print("The total is ");  // print message to serial monitor
        Serial.println(tot);  // print cumulative number to serial monitor and start a new line
        Serial.println();  // start another new line for clarity
        tot = 0;  // reset cumulative number
      }
    }
}
```

Serial communication between an Arduino board and an external TTL serial device requires connecting the Arduino TX pin to the device RX pin, the Arduino RX pin to the device TX pin, and the Arduino ground to the device ground. Instead, communication to a classical RS232 serial port requires a RS232/TTL converter, because the RS232 standard operates at +/- 12V, which may damage the Arduino pins.

The most common protocols for M2M communication among MCUs and TTL serial devices are the Inter Integrated Circuit (I2C) and the Serial Peripheral Interface (SPI) standards.

I2C is a two-wire serial communication system used between integrated circuits. The typical I2C bus consists of at least one master and one slave. In the most frequent situation, there is a single master and multiple slaves. However, multimaster and multi-slave architectures with bus arbitrage can be used in more complex systems. The bus was developed by Philips in 1982 and the first version of the protocol was released in 1992.

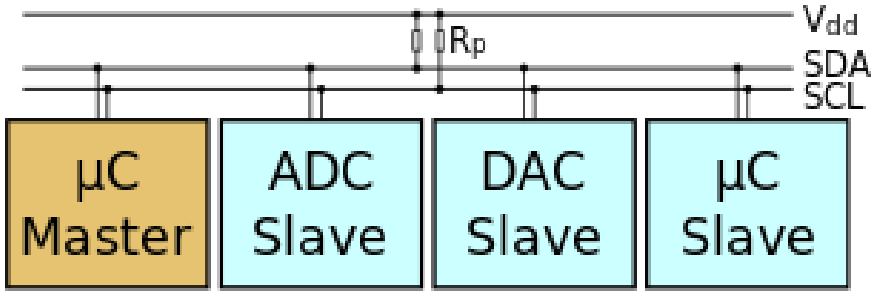


Figure: connections on a I2C bus, CC BY-SA by INDEX consortium

The I2C hardware requires two serial communication lines:

SDA (Serial DAta) for data;

SCL (Serial CLock) for clock, which makes I2C to be a synchronous bus.

A reference connection to ground and a power supply line Vdd attached to SDA and SCL through pull-up resistors are needed as well. The protocol reserves 7 bits for addressing a theoretical maximum of 112 different nodes connected on the same bus or mastering 16 special purposes. The transmission speed in standard mode is 100 kbit / s. Other variants provide as many as 10 address bits or high speed up to 3.4 Mbit / s. The maximum number of nodes is more often limited by the parasitic capacity introduced by each device, because the total capacity presented by SDA and SCL must remain below 400 pF. The principal difference between the nodes that serve as master and those that respond as slave is that the former provide the clock signal while the latter synchronize to the provided meter.

In an Arduino Uno, the I2C protocol is available through analog pins 4 for SDA and 5 for SCL.

SPI is a four-wire communication standard originally conceived by Motorola. Also, in this case, transmission takes place between a master and one or more slaves, where the former controls the bus, outputs the clock signal, and decides when to start and end each communication. The SPI bus is serial but, at variance with the I2C alternative, the communication is full-duplex, because transmission and reception occur on separate lines.

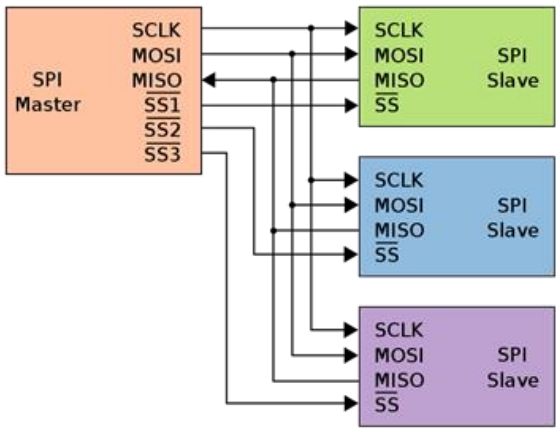


Figure: connections on a SPI bus, CC BY-SA by INDEX consortium

The SPI hardware requires four serial communication lines:

- SCLK: Serial Clock issued by the master;
- MISO: serial data Master In Slave Out, i.e. input for the master and output for the slave;
- MOSI: serial data Master Out Slave In, i.e. output for the master and input for the slave;
- SS: Slave Select issued by the master to choose which slave it wants to address.

As for the data exchange speed, the maximum limit depends on the specs of each connected device and their number, since each device adds its parasitic capacity to the communication lines. There are generally four connection lines that carry information. However, the need for a common connection to ground brings the total number of wires to five.

With respect to I2C, the main advantage of SPI is faster communication between master and individual slaves, with clock frequency that may easily exceed tens of MHz. The main disadvantage is the need to reserve an SS pin for each slave. The SS line is normally active low and, upon disabling with logic level HIGH, leaves the slave with high impedance output and so completely isolated from the bus. Therefore, the maximum number of slaves is limited only by the number of possible SS lines that can be managed by the master.

In an Arduino Uno, the SPI protocol is available through digital pins 13 for SCLK, 12 for MISO, 11 for MOSI and, most typically, 10 for SS.

INDUSTRIAL EXPERT

In the next unit, we will discuss an example that makes use of the SPI bus for fastest communication. Here, we show the use of the I2C bus to establish a serial communication between an Arduino Uno set as master and another Arduino Uno as slave. The master reads an analog thermometer and sends the reading to the slave. The slave parses the received reading and prints the temperature in centigrades to its own serial monitor. The hardware is as shown in the following figure. In practice, an analog thermometer connects to analog pin A0 of the master board. The SCL line of the master board connects to that of the slave board, and so is for the SDA line.

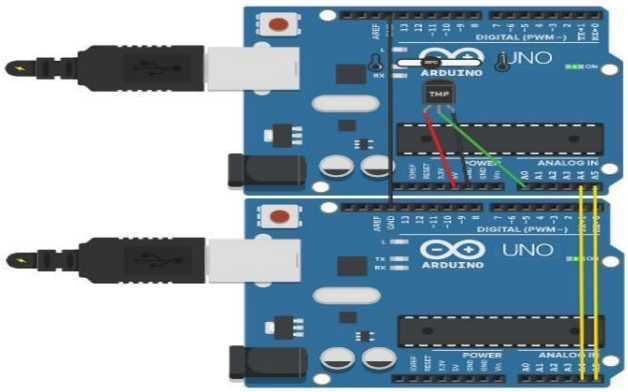


Figure: hardware required for I2C communication example, CC BY-SA by INDEX consortium

The sketch for the master board is as follows:

```
/*
 * I2C_Master
 * Sends sensor data to an I2C slave
 */

#include <Wire.h> // include library that enables I2C communication
const int address = 4; // address for communication to slave
const int sensorPin = A0; // pin attached to analog thermometer
int val; // variable to store analog reading
```

```
byte firstByte, secondByte; // variables to store values for transmission


void setup()
{
    Wire.begin(); // initialize I2C communication
}


void loop()
{
    val = analogRead(sensorPin); // store analog reading
    firstByte = highByte(val); // work out first value for transmission
    secondByte = lowByte(val); // work out second value for transmission
    Wire.beginTransmission(address); // begin I2C message
    Wire.write(firstByte); // transmit first value to slave
    Wire.write(secondByte); // transmit second value to slave
    Wire.endTransmission(); // end I2C message
    delay(1000); // wait 1 sec before another run
}
```

And this is the sketch for the slave:

```
/*
 * I2C_Slave
 * Monitors I2C requests, analyzes and prints values to its serial monitor
 */

#include <Wire.h> // include library that enables I2C communication
const int address = 4; // address for communication from master
byte firstByte, secondByte; // variables to store received values
int val; // variable to store analog reading
float temp; // variable to store temp in C

void setup()
{
  Serial.begin(9600); // initialize serial communication to serial monitor
  Wire.begin(address); // join I2C bus
  Wire.onReceive(displayTemp); // displayTemp is a callback function to execute upon reception of I2C message
}

void loop()
{
  // nothing here, all work is done in displayTemp
```

```
}


void  displayTemp(int  number)
{
   while(Wire.available()  >  0)  // while bytes are there through I2C bus
   {
      firstByte  =  Wire.read();  // receive first byte
      secondByte  =  Wire.read();  // receive second byte
   }
   val  =  word(firstByte,  secondByte);  // reconstruct analog reading
   temp  =  (val*5.0/1024.0-0.5)*100.0;  // translate analog reading into temp in C,
according to specs
   Serial.print(temp);  // print temp in C to serial monitor
   Serial.println(" C");  // print unit and start a new line
}
```

The wire.h library enables easy access to the I2C bus by the use of simple functions and methods that hide the complexity of the underlying protocol from view. The corresponding library for the SPI bus is called SPI.h and will be brought up in the next series of units.

Exercise

Try to simulate the electronic circuit described in this unit by the use of a tool like Tinkercad. Now add a third board and connect it to the original slave like this: gnd to gnd; A4 to A4; A5 to A5. The complete setup should be like that in the following figure:

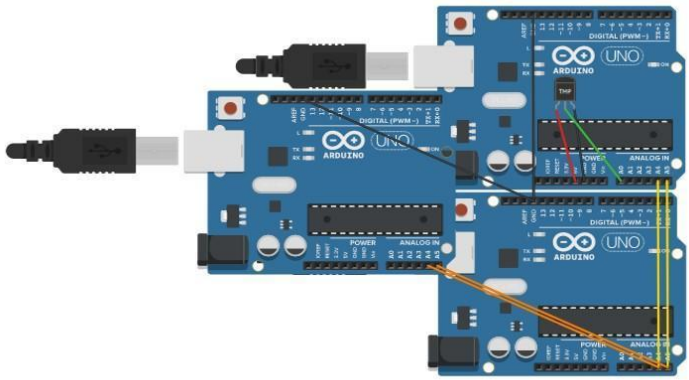


Figure: Hardware required for I2C communication exercise, CC BY-SA by by INDEX consortium

Modify the sketch of the original master like this:

```
#include <Wire.h>
const int address2 = 4, address3 = 5;
const int sensorPin = A0;
int val;
byte firstByte, secondByte;

void setup()
{
    Wire.begin();
}
```

```
void loop()
{
    val = analogRead(sensorPin);
    firstByte = highByte(val);
    secondByte = lowByte(val);
    Wire.beginTransmission(address2);
    Wire.write(firstByte);
    Wire.write(secondByte);
    Wire.endTransmission();
    Wire.beginTransmission(address3);
    Wire.write(firstByte);
    Wire.write(secondByte);
    Wire.endTransmission();
    delay(1000);
}
```

Leave the sketch of the original slave as is, and load the following sketch into the third board:

```
#include <Wire.h>
const int address = 5;
byte firstByte, secondByte;
```

```
int val;
float temp;

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  Wire.begin(address);
  Wire.onReceive(displayTemp);
}

void loop()
{
}

void displayTemp(int number)
{
  while(Wire.available() > 0)
  {
    firstByte = Wire.read();
    secondByte = Wire.read();
  }
  val = word(firstByte, secondByte);
```

```
    temp  =  (val*5.0/1024.0-0.5)*100.0;

    digitalWrite(LED_BUILTIN,  (temp>50.0));

}
```

Question

Which statement is correct?

☐ The third board is another slave attached to the original master. The original slave behaves like before. Moreover, the built-in LED of the new slave lights up when temp exceeds 50°C.
☐ The third board is another master attached to the original slave. The original slave behaves like before, plus now its built-in LED lights up when temp exceeds 50°C.
☐ The simulator returns a warning, because there are multiple slaves attached in series over the same bus. Some software may suggest to replace the I2C protocol with the SPI standard, in order to exploit its option to select one slave at the time.
☐ The compiler returns an error message, because different variables share identical names across more boards that are interconnected over the same bus.

Using Arduino in IoT projects

HTTP server: overview

In this unit, we describe the overall architecture of a simple IoT project based on an Arduino Uno. An Arduino board is connected to a LAN via Ethernet, and is configured to behave as an HTTP server that returns multiple HTML pages displaying the status of its analog and digital inputs and providing control over its digital outputs. The client may be any browser on a PC connected to the same LAN.

One HTML page will be available at the URL http://192.168.1.177/analog/ to return the rough values of analog pins A0 to A6, and will automatically refresh every 5 sec through a Meta Refresh method. An example is shown in the figure below.

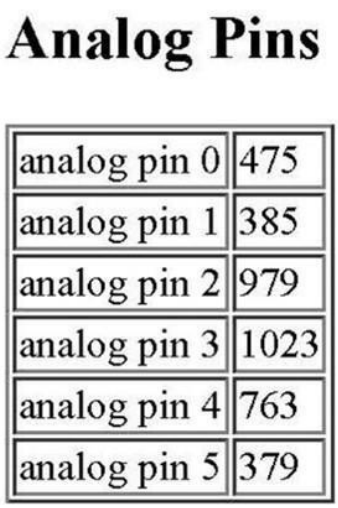


Figure: excerpt of the HTML page returned at the URL http://192.168.1.177/analog/, CC BY-SA by INDEX consortium

Another HTML page will be available at the URL http://192.168.1.177/digitalIn/ to display the logical levels of digital input pins D2 to D5 through the use of graphical images, and will also automatically refresh every 5 sec through a Meta Refresh method. By connecting to pull-up resistors, these digital pins will behave as active-low input latches. An example is shown in the following figure, where the left panel is the initial configuration and the right panel is the pattern returned after pin D3 was grounded through a jumper wire.

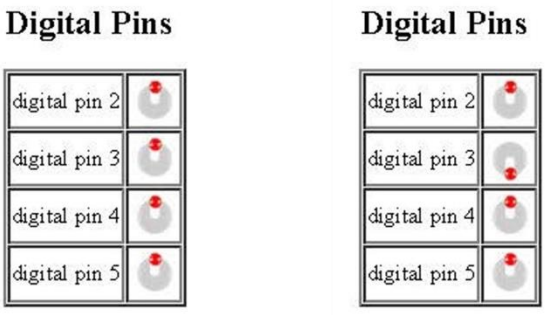


Figure: excerpt of the HTML page returned at the URL http://192.168.1.177/digitalIn/ before and after grounding pin D3, CC BY-SA by INDEX consortium

Finally, an HTML page will be available at the URL http://192.168.1.177/digitalOut/ to return the configuration of digital outputs D6 to D9, and allow the user to toggle their logical level through a submit button. An example is shown in the following figure, where the left panel shows some arbitrary configuration and the right panel is the result obtained after clicking the button corresponding to pin D8.

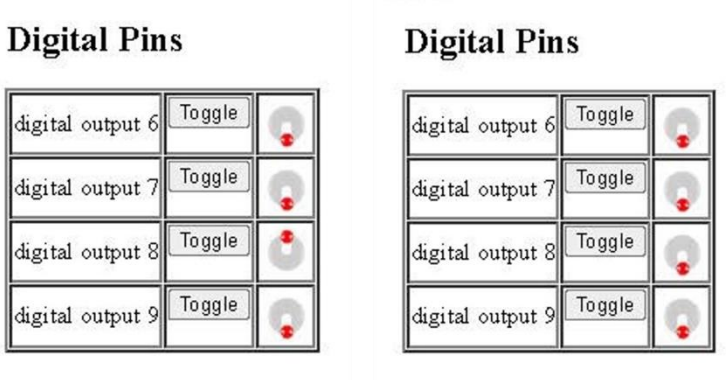


Figure: excerpt of the HTML page returned at the URL http://192.168.1.177/digitalOut/ before and after toggling digital output D8, CC BY-SA by INDEX consortium

Should the client submit a wrong request, such as http://192.168.1.177/anologOut/ , the server will generate instructions that explain the valid keys, as in the following Figure:

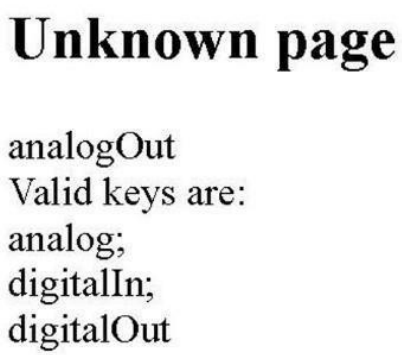


Figure: excerpt of the HTML page returned after a wrong request such as http://192.168.1.177/analogOut/, CC BY-SA by INDEX consortium

Note that digital pins D0 and D1 are reserved for possible serial communication to a PC connected through a USB cable, which may be used for debugging or monitoring all client requests and server activities. Finally, digital pins D10 to D13 are consumed for SPI connection to the Ethernet module.

Next unit describes the hardware needed to realize this project.

HTTP server: hardware

The market offers a variety of preassembled modules that make it possible for anybody to arrange the hardware required to create an HTTP server. Here, we used an Arduino Ethernet Shield 2, which is a device featuring the same standard footprint as an Arduino Uno and ready for plugging into its female headers with a corresponding set of pass-through connectors that give full access to the underlying board.

Figure: the Arduino Ethernet Shield 2, CC BY-SA by INDEX consortium

The Arduino Ethernet Shield 2 makes use of the Wiznet W5500 Ethernet chip to provide an IP stack capable of both TCP and UDP transport layer protocols, and suitable to support up to eight simultaneous socket connections. An 8P8C socket is available for connection to standard RJ45 cables with integrated line transformer and also Power over Ethernet enabled for commodity. Connection to a network hub or router requires a standard CAT5 or CAT6 Ethernet cable, while a twisted-pair cable may be necessary for indirect use with a PC. The Wiznet W5500 Ethernet chip communicates with the Arduino UNO through the SPI protocol and consumes digital pins D10 to D13. In particular, digital pin D10 serves as SS.

Note that the Arduino Ethernet Shield 2 also contains more connectors and an onboard micro-SD card slot, which may be used, for instance, to store files for serving over the network, which we will not implement in our example, though. Also, this chip communicates with the Arduino UNO through the

same SPI bus with pin D4 as SS. Therefore, it may be advisable to write a high value to this pin when the micro-SD card slot is not in use.

In order to complete the hardware required for this project, plug an Arduino Ethernet Shield 2 onto an Arduino Uno and a standard RJ45 cable. Attach the other end of this cable to a network hub or router. There is nothing more to do. Of course, it would be more meaningful to connect peripherals as analog sensors to pins A0 to A5, digital sensors to inputs D2 to D5 or actuators to D6 to D9, but their use is not needed for a functional demonstration of this project. The only other thing that may be helpful is a jumper wire to verify the reading of the various inputs, for instance, by grounding. In the next unit, we will provide the sketch for implementing the behaviour outlined in the previous unit.

HTTP server: software

The sketch described in this unit is rather basic, and the interested or experienced reader may try to delve into its lower-level details. However, the level of complexity of this code is already much higher than the cases reported in the previous session, and we will propose a description made by chunk rather than by line as we did before. Our goal is to give a feeling of the structure of a project of this kind. In particular, some constructs are more complex than a direct use of self-documenting variables and functions, such as pointers, referencing and dereferencing to locations in memory denoted by symbols as * and &, etc. We invite the beginner reader to pass over these symbols, and just try to visualize the overall architecture and behaviour of the sketch. For additional details, please consult the many resources available on the web.

The overall idea is that the Arduino server is there to listen for HTTP request messages from a generic client. When a generic client moves forward, the Arduino server receives and parses its HTTP request message, reads and modifies its configuration as appropriate, through the usual analogRead, digitalRead or digitalWrite commands, and returns an HTTP response message containing HTML code to recreate a web page suitable for a human interface.

We will make use of the following libraries, in order to exploit their higher-level functions:

- SPI – to use the Ethernet library in Arduino versions later than 0018;
- Ethernet – to connect and exchange messages to the Internet;
- avr/pgmspace – to store data in flash memory instead of SRAM. An Arduino Uno features 32 kB of flash memory and 2 kB of SRAM. The flash memory is used to store the program code, and sometimes its size is superabundant. Instead, the SRAM undergoes rapid saturation in the presence of lengthy strings. Here, we exploit the capacity of the flash memory to store most of the HTML code needed to create the web pages of interest. In particular, our HTML code will contain a small JPEG image of a size of 35 pixels × 35 pixels encoded as a base64 string that alone

consumes more than 1.4 kB of computational resources! We have chosen this example also to give an idea of the scale of the limitations met in highly constrained systems as a MCU, with respect to a PC, for instance. In more extreme cases, where the phrase extreme may probably sound to most readers as hyperbolic or ironic, resorting to files written in an SD card may be a valuable alternative, for instance.

The sketch is strongly based on a project described in ISBN: 9781449313876, which is a classical reference in the Arduino community.

In practice, the setup() function starts the HTTP server with the appropriate MAC and IP addresses ready for communication through port 80, and sets the pin modes of the various inputs and outputs, as usual.

The loop() function sorts the various requests coming from a client, by parsing incoming messages. First, it checks whether it is a GET or a POST request. GET requests are those explicitly issued by the user by typing the address of a page as http://192.168.1.177/analog/. POST requests will be automatically generated by clicking any of a series of interactive buttons available in the digitalOut page, as we shall see. Next, it parses the next chunk of message, in search of the strings analog, digitalIn or digitalOut. If any of those is found, it calls relevant functions in charge of the behaviour of the HTTP server.

In particular, a so-called showAnalog() function reads the analog pins and responds to the client with a conventional OK message and HTML code that encodes a table as displayed in the figure of the second previous unit. The HTML code is rather verbose. In order to preserve the SRAM space, all constant strings needed for its recreation are stored in the flash memory space.

Otherwise, a so-called showDigitalIn() function behaves in a similar way for the digital inputs and generates a table as shown in the figure of the second previous unit. In order to make the appearance of this page a little fancier for a human interface, digital readings are represented as a lever up or down. Lever up is stored as a 35 pixels × 35 pixels JPEG file encoded in a 1408 characters base64 string, and its transmission is commissioned to a so-called printP() function. Lever down is obtained by flipping the lever up.

Finally, a so-called showDigitalOut() function manages all requests about the digital outputs. However, now it will consist of GET, as well as POST requests. Usual GET requests are responded with a table similar to those above. The difference is the presence of an interactive button per line for toggling the status of the relevant pin, as is shown in the figure in the second previous unit. Each button is configured to automatically generate a POST request containing the string digitalOut for rebounding here, followed by an id number. In turn, the POST kind activates a chunk of code that parses the message to find the id number and swap the status of the relevant pin.

Wrong messages are forwarded to a showUnknown() function, which prints instructions to the client like those shown in the figure in the second previous unit.

The sketch also provides for the use of serial communication through a USB cable of principal interest for debugging.

Here is the complete sketch:

```
/*
 * Web server multipage sketch
 *
 * Responds to request messages in the URL to show analog and digital inputs and change digital outputs
 *
 * http://192.168.1.177/analog/ displays analog pin data
 * http://192.168.1.177/digitalIn/ displays digital pin data
 * http://192.168.1.177/digitalOut/ allows changing digital pin data
 *
 */


// preliminary operations: inclusion of libraries, and definition of global variables, i.e. MAC address of the Arduino Ethernet Shield 2 marked on chip, static IP address of the Arduino server, and buffer used to store incoming messages, creation of an instance of server object and client object. Server object will represent the Arduino server ready to exchange HTTP messages through standard port 80, and client object will be a generic client

#include <SPI.h>
#include <Ethernet.h>
```

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

byte mac[] = {0xA8, 0x61, 0x0A, 0xAE, 0x64, 0x60};

byte ip[] = {192, 168, 1, 177};

const int MAX_PAGENAME_LEN = 11;

char buffer[MAX_PAGENAME_LEN+1];

EthernetServer server(80);

EthernetClient client;


// the setup() function enables serial communication, Ethernet connection, activates
the Arduino server, and sets digital pins 2 to 5 as inputs connected to pull-up resistors
and 6 to 9 as outputs initialized as LOW. Begin to use the F() macro to print messages
stored in flash memory

void setup()

{

   Serial.begin(9600);

   Ethernet.begin(mac, ip);

   server.begin();

   delay(1000);

   Serial.println(F("Ready"));

   for(int i = 2; i < 6; i++)

   {

      pinMode(i, INPUT_PULLUP);

   }
```

```
    for(int i = 6; i < 10; i++)
  {
    pinMode(i, OUTPUT);
    digitalWrite(i, LOW);
  }
}
```

// the loop() function controls the behavior of the Arduino server. A variable isPost is set to toggle the behavior of the server between read and read/write. If a generic client connects to the Arduino server with a request message, buffer is flushed and refilled with request message until first slash. If buffer contains string GET, then variable isPost is set to false. Otherwise, if string POST is there, it is set to true, which will allow writing to the digital output, as we shall see. Then, buffer is flushed again and refilled with next part of request message until second slash. Buffer is parsed again in search for strings analog, or digitalIn, or digitalOut. If either is found, the showAnalog(), or showDigitalIn(), or showDigitalOut()function is respectively called. The function showDigitalOut() depends on variable isPost. If any other string is found instead, the showUnknown() function is called with buffer as parameter

```
void loop()
{
  client = server.available();
  if(client)
  {
    boolean isPost;
    while(client.connected())
    {
```

```
if(client.available())
{
    memset(buffer, 0, sizeof(buffer));
    if(client.readBytesUntil('/', buffer, MAX_PAGENAME_LEN))
    {
        if(strcmp(buffer, "GET ") == 0)
            isPost = 0;
        else if(strcmp(buffer,"POST ") == 0)
            isPost = 1;
        memset(buffer, 0, sizeof(buffer));
        if(client.readBytesUntil('/', buffer, MAX_PAGENAME_LEN))
        {
            if(strcasecmp(buffer, "analog") == 0)
                showAnalog();
            else if(strcasecmp(buffer, "digitalIn") == 0)
                showDigitalIn();
            else if(strcmp(buffer, "digitalOut")== 0)
                showDigitalOut(isPost);
            else
                showUnknown(buffer);
        }
    }
```


```
            break;
          }
        }
        delay(1);
        client.stop();
    }
}
```

// the showAnalog() function prints text to the serial port and a HTTP message to the client. HTTP message consists of a standard header and a HTML code to create a web page. A meta refresh instruction imparts automatic refresh every 5 sec. Title of web page is Multi-page example-Analog. Its first line consists of headline Analog pins. Then, there begins a table. For each analog input pin i, first column states Analog pin i, and second column reports the analog reading for pin i. Then, a new line begins for next pin

```
void  showAnalog()
{
    Serial.println(F("Analog"));
    client.println(F("HTTP/1.1 200 OK"));
    client.println(F("Content-Type: text/html"));
    client.println();
    client.println("<meta http-equiv=\"refresh\" content=\"5\">");
    client.print(F("<html><head><title>"));
    client.println(F("Multi-page example-Analog"));
    client.println(F("</title><body>"));
```

```
    client.println(F("<h2>Analog Pins</h2>"));
    client.println(F("<table border='1'>"));
    for(int i = 0; i < 6; i++)
    {
        client.print(F("<tr><td>analog pin "));
        client.print(i);
        client.print(F("</td><td>"));
        client.print(analogRead(i));
        client.println(F("</td></tr>"));
    }
    client.println(F("</table></body></html>"));
}
```

// base64 code to create an image representing the state of digital pins as a lever up or turned upside down. Data are stored in flash memory

```
static const unsigned char leverUp[] PROGMEM =
"/9j/4AAQSkZJRgABAQEAlgCWAAD/2wBDAAMCAgMCAgMDAwMEAwMEBQgFBQQEBQoHBwYIDAoMDAsK"

"CwsNDhIQDQ4RDgsLEBYQERMUFRUVDA8XGBYUGBIUFRT/2wBDAQMEBAUEBQkFBQkUDQsNFBQUFBQU"

"FBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBT/wAARCAAjACMDASIA"

"AhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFBAQA
```

"AAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkKFhcYGRolJico
KSo0NTY3"

"ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl
5iZmqKjpKWm"

"p6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8Q
AHwEA"

"AwEBAQEBAQEBAQAAAAAAAAECAwQFBgcICQoL/8QAtREAAgECBAQDBAcFBAQAAQ
J3AAECAxEEBSEx"

"BhJBUQdhcRMiMoEIFEKRobHBCSMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5Ok
NERUZHSElK"

"U1RVVldYWVpjZGVmZ2hpanN0dXZ3eHl6goOEhYaHiImKkpOUlZaXmJmaoqOkpaanqK
mqsrO0tba3"

"uLm6wsPExcbHyMnK0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRA
xEAPwD9U64v"

"4oePpPBXhu5k0u1Gq+IJAIrHT153yscAuAQdq8seQdqn6jovEWpNpWkzzx/63G1OOjHv
+HX8K8Tu"

"vCth8SppND1pZLiy1DcspVyrg4JDhuzAgEH2rOopODUN+h3YGVCGKpyxSvTUlzddL66X
V/S6v3W5"

"57+zP+1h46+I3ii6tvFukaamhC5+xNqFqRC9tcFWZU2FiXBCNnA46k4FfX6sJFDKQysMg
g5BFfK3"

"hz9knwt8C7o6zpl/qGp3UsuyIXzJth+VssAqjLYJGfc8entfw11qV99hId0WC0ef4T3H0PX
8PeuD"

"L6eJp0bYp3l63PquMMVk2MzN1cihyUbLaLir9dG3+np1ff0UUV6Z8OYXjSB5tBmKAnyy
HOPTkE/h"

"nP4V5DY3k2l30VxC2yWJsqa95dRIrKw3KwwQe4rgNe+GjyStLpsi7Sc+TIcEfQ9/xoGjlte
8VXni"

"COKO4EapGchY1xk+p5re+Gdu76g0gHyopZj9Rgf1qtZ/DXVJpgJhHbpnlmcN+QFeiaHodtoNmILc"

"ZJ5eRvvOfegZo0UUUEhRRRQAUUUUAFFFFAH/2Q==";

// the showDigitalIn() function is similar to the showAnalog() counterpart. Title of web page now is Multi-page example-Digital in. Its first line consists of headline Digital Pins. Then, there begins a table. For each digital input pin i, first column of table states Digital pin i, and second column contains an image with payload made of the base64 code for lever up, which is left as is or turned upside down according to digital reading of pin i. The base64 code is segmented and transmitted through the prntP() function defined at the end of the sketch. Then, a new line begins for next pin

```
void  showDigitalIn()

{

   Serial.println(F("Digital in"));

   client.println(F("HTTP/1.1 200 OK"));

   client.println(F("Content-Type: text/html"));

   client.println();

   client.println("<meta http-equiv=\"refresh\" content=\"5\">");

   client.print(F("<html><head><title>"));

   client.println(F("Multi-page example-Digital in"));

   client.println(F("</title><body>"));

   client.println(F("<h2>Digital Pins</h2>"));

   client.println(F("<table border='1'>"));

   for(int  i  =  2;  i  <  6;  i++)

   {
```

```
        client.print(F("<tr><td>digital pin "));
        client.print(i);
        client.print(F("</td><td>"));
        client.print(F("<img src=\""));
        printP(leverUp);
        if(digitalRead(i)  ==  LOW)
            client.print(F("\" style=\"transform:scaleY(-1);\">"));
        else
            client.print(F("\">"));
        client.println(F("</td></tr>"));
    }
    client.println(F("</table></body></html>"));
}
```

// the showDigitalOut() function is more complex and depends on the kind of request message, i.e. whether it was a GET or a POST message. In case of POST message, there is extra code. Request message is parsed to find string pinD. Following number is stored in variable pin, and corresponding output is toggled.

// Next part is same for GET and POST cases, and also resembles the showDigitalIn() function, i.e. instructions to send HTML code and create a web page. Title of the web page is Multi-page example-Digital out. Its first line consists of headline Digital Pins. Then, there begins a table. For each digital output pin i, first column states Digital pin i, second column hosts an HTML button labelled Toggle to generate a POST request containing string pinD plus i, and third column hosts same icon as above. Then, a new line begins for next pin

```
void showDigitalOut(boolean isPost)
{
  Serial.println(F("Digital out"));
  if(isPost)
  {
    Serial.println("POST request");
    client.find("\n\r");
    Serial.print(F("Pin to toggle: D"));
    while(client.findUntil("pinD","\n\r"))
    {
      int pin = client.parseInt();
      Serial.println(pin);
      digitalWrite(pin, !digitalRead(pin));
    }
  }
  client.println(F("HTTP/1.1 200 OK"));
  client.println(F("Content-Type: text/html"));
  client.println();
  client.print(F("<html><head><title>"));
  client.println(F("Multi-page example-Digital out"));
  client.println(F("</title><body>"));
  client.println(F("<h2>Digital Pins</h2>"));
```

```
    client.println(F("<table border='1'>"));
    for(int i = 6; i < 10; i++)
    {
        client.print(F("<tr><td>digital output "));
        client.print(i);
        client.print(F("</td><td>"));
        client.print(F("<form action='/digitalOut/' method='POST'><p><input
type='hidden' name='pinD"));
        client.print(i);
        client.print(F("'><input type='submit' value='Toggle'/></form>"));
        client.print(F("</td><td>"));
        client.print(F("<img src=\""));
        printP(leverUp);
        if(digitalRead(i) == LOW)
            client.print(F("\" style=\"transform:scaleY(-1);\">"));
        else
            client.print(F("\">"));
        client.println(F("</td></tr>"));
    }
    client.println(F("</table></body></html>"));
}
```

// the showUnknown() function takes a string as parameter. Remember that, according to the showDigitalOut() function, the string passed is a wrong page name. This function sends HTML code to create a web page with title Multi-page example-Unknown Page, headline Unknown page, a line with the wrong page name, and more lines explaining valid keys

```
void  showUnknown(char  *page)

{

    Serial.print(F("Unknown: "));

    Serial.println(page);

    client.println(F("HTTP/1.1 200 OK"));

    client.println(F("Content-Type: text/html"));

    client.println();

    client.print(F("<html><head><title>"));

    client.println(F("Multi-page example-Unknown Page"));

    client.println(F("</title><body>"));

    client.println(F("<h2>Unknown page</h2>"));

    client.println(page);

    client.println(F("<br />Valid keys are:<br />analog;<br />digitalIn;<br />digitalOut"));

    client.println(F("</body></html>"));

}
```

// the printP() function takes one string as parameter. In practice, this function copies segments of a string stored in flash memory into a buffer of 32 bytes for passing to the client

```
void printP(const unsigned char *str)
{
  // from webduino library Copyright 2009 Ben Combee, Ran Talbott
  uint8_t buffer[32];
  size_t bufferEnd = 0;
  while(buffer[bufferEnd++] = pgm_read_byte(str++))
  {
    if(bufferEnd == 32)
    {
      client.write(buffer, 32);
      bufferEnd = 0;
    }
  }
  if(bufferEnd > 1)
    client.write(buffer, bufferEnd - 1);
}
```

Uploading this sketch into an Arduino Uno and using a browser to send request messages as http://192.168.1.177/analog/ , http://192.168.1.177/digitalIn/ and http://192.168.1.177/digitalOut/ recreates the web pages shown in the first unit of this series.

Co-funded by the
Erasmus+ Programme
of the European Union

INDUSTRIAL EXPERT

HTTP server: conclusions

In this series of units, we have covered the creation of an infrastructure for an Arduino server, but we have not addressed its connection with actual sensors or actuators. Possibilities are innumerable, and may include, for instance, the hardware required to gain full control over a greenhouse, a production line, a safety system, etc.

In addition, of course, the Arduino board may be easily programmed not only to cover the human interface by serving web pages, but also, for instance, to make computations at edge level, or to actuate automatic actions within a local area network, such as to get safe conditions in case of anomalies.

Meanwhile, this project casts light on typical limitations of a simple HTTP server, such as the need for frequent polling to identify critical events that may deserve a timely reaction on the human side, which may be a very inefficient way to discover rare changes in analog or digital input signals.

In the next series of units, we present another example of a project that provides a very complementary overview over the use of Arduino boards in IoT systems.

Exercise

Suppose someone wants to split the Analog Pins page into two separate pages, in order to monitor sensors attached to analog pins 0 to 2 in one page, and 3 to 5 in another one, for whatever reason. What main modifications are needed?

Only one answer is correct.

☐ To modify the sketch in a few parts: to work up the innermost conditional in the loop() function, to update code line for(int i = 0; i less-than 6; i++) in the showAnalog() function, and to create another item similar to the showAnalog() function.
☐ To modify the sketch in a few parts: to update code lines for(int i = 2; i less-than 6; i++) in the setup() function and for(int i = 0; i less-than 6; i++) in the showAnalog() function, and to create another item similar to the showAnalog() function {No, the setup() function is fine as is}.
☐ To write a new sketch from scratch, because the design of the current code is inherently not scalable {No, the current sketch is fit to accommodate more pages without major modification}.
☐ To stack another Arduino Ethernet Shield 2, and to modify the sketch in a few parts: to define more MAC and IP addresses, and to duplicate the showAnalog() function {No, the hardware is fine as is}.

Voice-controlled TV: overview

In this series of units we present a project that differs from the previous case in many respects.

Instead of a general-purpose platform, its scope targets a narrow set of high-level tasks. And instead of a self-sufficient approach, its implementation makes extensive use of external services to include off-the-shelf tools made available in top-tier infrastructures like Amazon Web Service and Arduino IoT Cloud Service. At the time of writing, the resources implemented in this project are offered free of charge, but an extensive use of external services implies compromising on dimensions such as autonomy and control. In addition, each service may require committing to its specific standards, and the concatenation of more services may entail interfacing different APIs and communication protocols, such as HTTP2 and MQTT. The upside is that most of the underlying complexity may be overlooked and the focus be kept on the particular behaviour of the edge device. The code dictating such behaviour may be sometimes secluded within so-called callback functions triggered by predefined communication events.

We have chosen this project because we believe that it represents well a typical scenario where an engineer may opt to optimize on resources by leaning on external services, and so to act within a preset framework to perform a specific task of great added value. We put together many of the concepts seen in previous sections of this module, such as the use of different data link layer technologies, i.e. WiFi and IR light, different application layer protocols, i.e. HTTP2 and MQTT, and services of a different profile, i.e. large-scale Infrastructure as a Service such as Amazon Web Service and prototype-friendly solutions like Arduino IoT Cloud Service.

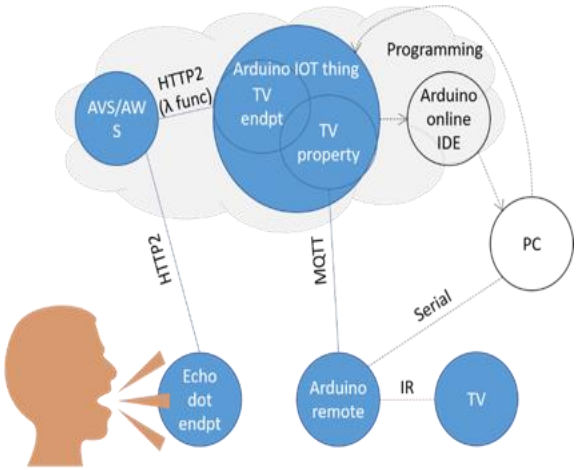


Figure: voice-controlled TV: topology, CC BY-SA by INDEX consortium

In a nutshell, this project makes use of Amazon Alexa virtual assistant and an Arduino Nano 33 IoT board to gain full control over a standard TV via speech recognition.

The overall system is such that a smart speaker such as an Amazon Echo Dot recognizes the keyword "TV", and connects as an endpoint to the Amazon Voice Service of Amazon Web Service via HTTP2. Next, the Amazon Web Service connects through a so-called lambda function to the Arduino IoT Cloud Service as another endpoint, and modifies a virtual set of TV properties via HTTP2. At that moment, the Arduino IoT Cloud Service behaves as a broker and forwards such events to an Arduino client via MQTT. The Arduino client is an edge device hosting an IR LED and programmed to emulate the ordinary control. In other words, it flashes according to a consumer protocol hacked in a preliminary step. The code governing the Arduino remote is secluded in a callback function embedded in a general template that was passed to the Arduino online IDE from the Arduino IoT Cloud Service at the time of its configuration.

This project takes much inspiration from that published at create.arduino.cc-projecthub with modifications. An interesting overview over the use of the Alexa Voice Service is available at developer.amazon.com-alexa-voice-service . An introduction to the Arduino IoT Cloud Service is available at https://www.arduino.cc/en/IoT/HomePage , and we encourage the experimenter to explore relevant tutorials, such as that illustrated at https://create.arduino.cc/projecthub/133030/iot-cloud-getting-started-c93255 .

Voice-controlled TV: IR remote decoder: hardware

This project is made up of two parts.

The first part is devoted to the development of an IR remote decoder intended to hack and decipher the signals sent from an ordinary remote, by the use of an IR receiver. Once decoded, we will assemble an Arduino remote designed to emulate these signals by flashing the same pattern of IR pulses for each command, such as volume up, or channel 9, etc. This approach is probably the simplest path to control a standard TV with an Arduino board, thanks to a contributed library named IRremote, as we shall see.

The hardware needed for the IR remote decoder is very simple.

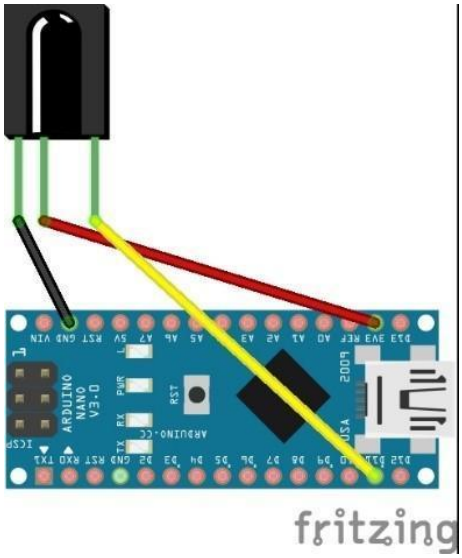


Figure: hardware required for the IR remote decoder, CC BY-SA by INDEX consortium

We make use of an Arduino Nano 33 IoT board. This board features a low power Arm Cortex-M0 32-bit SAMD21 MCU, a low power u-blox NINA-W10 chipset operating in the 2.4-GHz range for WiFi and Bluetooth connectivity, and a Microchip ECC608 crypto chip for secure communication. Note that the operating voltage of the Arduino Nano 33 IoT board is 3.3 V, instead of 5 V as the Arduino Uno. In this part of the project, we don't need WiFi or Bluetooth connectivity. Such functionality will come in handy when mounting the Arduino remote, though. A generic IR receiver connects to digital pin 11 of the Arduino Nano 33 IoT board. Common IR receivers consist of a photodiode, a preamplifier and an IR filter assembled in a miniature frame, and return a demodulated output from a PWM signal with standard carrier frequency of 38 kHz.

Voice-controlled TV: IR remote decoder: software

The sketch used to manage the IR remote decoder makes extensive use of the IRremote library that may be found at https://github.com/z3t0/Arduino-IRremote. In particular, the sketch reported below returns the consumer IR code as raw data, as we will briefly explain soon below:

```
// preliminary operations: inclusion of the IRremote library, definition of digital pin attached to the IR receiver, and instanciation of an IRrecv object

#include <IRremote.h>

int recvPin = 11;
```

```
IRrecv  irrecv(recvPin);


// the setup() function enables serial communication and starts the IR receiver
void  setup()
{
   Serial.begin(9600);
   irrecv.enableIRIn();
}


// the loop() function creates a structure to store data. If an IR code is grabbed, data are parsed through the function dumpCode(), which will output results as source code. A blank line is printed to the serial monitor. And the IR receiver is resumed
void  loop()
{
   decode_results  results;
   if(irrecv.decode(&results))
   {
      dumpCode(&results);
      Serial.println("");
      irrecv.resume();
   }
}
```

// the dumpCode() function prints to the serial monitor an array of unsigned int named rowData that contains a representation of the IR code. The comma after entries of odd index in the final array is followed by a space

```
void dumpCode (decode_results *results)
{
  Serial.print("unsigned int rawData[");
  Serial.print(results->rawlen - 1, DEC);
  Serial.print("] = {");
  for(int i = 1; i < results->rawlen; i++)
  {
    Serial.print(results->rawbuf[i] * USECPERTICK, DEC);
    if(i < results->rawlen-1)
      Serial.print(",");
    if(!(i & 1))
      Serial.print(" ");
  }
  Serial.print("};");    //
}
```

After the sketch is loaded into the Arduino Nano 33 IoT board, and the serial monitor is opened, pointing the ordinary remote to the IR receiver and pushing, for instance, the power button would generate a printout similar to:

```
unsigned int rawData[67] = {9150,4500, 600,550, 600,550, 600,1700, 600,550,
600,500, 650,500, 650,500, 650,500, 600,1700, 600,1700, 600,550, 600,1650,
600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,550, 600,550, 600,1650,
600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650, 650,1650, 600,550,
600,1650, 650,1650, 600,1700, 600,1650, 600};
```

This array suggests that the particular remote used in this example makes use of NEC protocol, which encodes logical ones as a train of PWM pulses of a duration of about 560 µs at a carrier frequency of 38 kHz and typical repetition rate between 1/4 and 1/3, followed by a pause of 1690 µs, and logical zeros as a train of PWM pulses around 560 µs and a pause of 560 µs.

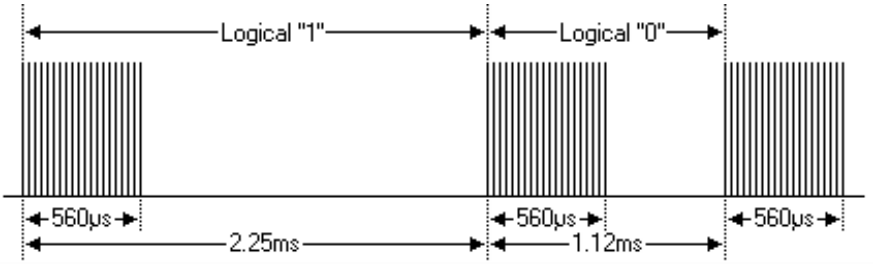


Figure: logical ones and zeros according to NEC protocol, CC BY-SA by INDEX consortium

The full sequence begins with a train of PWM pulses around 9.0 ms, a pause of 4.5 ms, and then four bytes carrying instructions, and a final train of PWM pulses of 560 µs. Out of the four bytes carrying instructions, only the first and the third actually convey information, and respectively encode the address of the intended receiver and the desired command. The second and fourth bytes are the logical inverse of the first and the second, respectively, and serve both to identify errors and to standardize the duration of the entire sequence. There exists additional code for special conditions, such as repeated commands, etc. that we overlook at this time.

Figure: example of command according to NEC protocol, CC BY-SA by INDEX consortium

The IRremote library returns results as a buffer containing the duration in µs of all trains of PWM pulses and pauses, in the order as received. So, the binary representation of the four bytes carrying information in the example tested above would read 00100000, 11011111, 00010000, 11101111.

Repeating the same operation for buttons power, channels from 0 to 9, mute, volume up, volume down, channel up, channel down; and copying and pasting each printout into a text file with appropriate headings would generate a content similar to:

Power

```
unsigned int rawData[67] = {9150,4500, 0600,550, 600,550, 600,1700,
600,550, 600,500, 650,500, 650,500, 650,500, 600,1700, 600,1700, 600,550,
600,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,550, 600,550,
600,1650, 600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650, 650,1650,
600,550, 600,1650, 650,1650, 600,1700, 600,1650, 600};
```

1

```
unsigned int rawData[67] = {9150,4500, 600,550, 600,550, 600,1650, 650,500,
650,500, 600,550, 650,500, 600,550, 600,1700, 600,1650, 650,500, 650,1650,
600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,550, 600,550, 600,550,
600,1700, 600,550, 600,550, 600,500, 650,500, 650,1650, 600,1700, 600,1650,
600,550, 650,1650, 600,1700, 600,1650, 600};
```

2

```
unsigned int rawData[67] = {9150,4500, 600,550, 600,550, 600,1700, 600,550,
600,500, 650,500, 650,500, 650,500, 650,1650, 600,1700, 600,550, 600,1650,
600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,1650, 600,550, 600,550,
600,1700, 600,550, 600,550, 600,550, 600,1650, 600,550, 600,1700, 600,1700,
600,500, 650,1650, 600,1700, 600,1650, 600};
```

3

```
unsigned int rawData[67] = {9150,4550, 600,500, 650,500, 650,1650, 600,550,
600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,1700,
600,1650, 650,1650, 600,1700, 600,1650, 600,1700, 600,1650, 650,500, 600,550,
650,1650, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650,
600,550, 600,1700, 600,1650, 600,1700, 600};
```

4

```
unsigned int rawData[67] = {9150,4500, 600,550, 600,550, 600,1650, 650,500,
650,500, 650,500, 650,500, 600,550, 600,1700, 600,1650, 650,500, 650,1650,
600,1700, 600,1650, 600,1700, 600,1700, 600,550, 600,550, 600,1650, 600,550,
600,1700, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,550, 600,1650,
650,500, 650,1650, 600,1650, 650,1650, 600};
```

5

```
unsigned int rawData[67] = {9150,4550, 600,550, 600,550, 600,1650, 600,550,
600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650, 600,550, 600,1700,
600,1650, 650,1650, 600,1700, 600,1650, 600,1700, 600,550, 600,1700, 600,500,
650,1650, 600,550, 600,550, 600,550, 600,550, 600,1700, 600,550, 600,1650,
600,550, 600,1700, 600,1650, 650,1650, 600};
```

6

```
unsigned int rawData[67] = {9150,4550, 600,550, 600,500, 650,1650, 600,550,
600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,1700,
600,1650, 650,1650, 600,1700, 600,1650, 600,550, 600,1700, 600,1650, 650,500,
650,1650, 600,550, 600,550, 600,550, 600,1650, 650,550, 600,500, 650,1650,
600,600, 550,1700, 600,1650, 600,1700, 600};
```

7

unsigned int rawData[67] = {9150,4500, 600,550, 600,550, 600,1700, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,550, 600,1650, 650,1650, 600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,1700, 600,550, 600,1650, 650,500, 650,500, 600,550, 650,500, 600,550, 600,550, 600,1700, 600,550, 600,1650, 600,1700, 600,1700, 600};

8

unsigned int rawData[67] = {9150,4500, 600,550, 600,550, 600,1650, 650,500, 650,550, 600,500, 650,500, 600,550, 600,1700, 600,1700, 600,550, 600,1650, 600,1700, 600,1650, 600,1700, 600,1700, 600,550, 600,550, 600,500, 650,1650, 600,1700, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,1650, 650,550, 600,500, 650,1650, 600,1700, 600,1650, 600};

9

unsigned int rawData[67] = {9150,4550, 600,550, 600,500, 650,1650, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,1700, 600,550, 600,550, 600,1650, 600,1700, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,550, 600,1700, 600,1650, 600,1700, 600};

0

unsigned int rawData[67] = {9150,4500, 600,550, 600,550, 600,1700, 600,550, 600,500, 650,500, 650,500, 650,500, 650,1650, 600,1700, 600,550, 600,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,550, 600,550, 600,550, 600,1650, 600,550, 600,550, 600,550, 600,1700, 600,1650, 650,1650, 600,1700, 600,500, 650,1650, 600,1700, 600,1650, 650};

Mute

```
unsigned int rawData[67] = {9150,4500, 650,500, 600,550, 650,1650, 600,550,
600,550, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,550, 600,1650,
650,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,550, 600,1650,
600,550, 600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650, 600,550,
600,1700, 600,1650, 600,1700, 600,1700, 600};
```

Vol +

```
unsigned int rawData[67] = {9150,4550, 600,550, 600,550, 600,1650, 650,500,
600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650, 650,500, 600,1700,
600,1700, 600,1650, 600,1700, 600,1650, 650,500, 650,1650, 600,550, 600,550,
600,550, 600,550, 600,550, 600,550, 600,1650, 600,550, 650,1650, 600,1700,
600,1650, 600,1700, 600,1650, 650,1650, 600};
```

Vol -

```
unsigned int rawData[67] = {9150,4500, 600,550, 600,550, 600,1650, 650,500,
650,500, 600,550, 650,500, 600,550, 600,1700, 600,1650, 650,500, 650,1650,
600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,1700, 600,550, 600,550,
600,550, 600,550, 600,550, 600,550, 600,550, 600,500, 650,1650, 600,1700,
600,1650, 650,1650, 600,1700, 600,1650, 600};
```

Pr +

```
unsigned int rawData[67] = {9150,4500, 650,500, 650,500, 600,1700, 600,550,
600,550, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,550, 600,1700,
600,1650, 600,1700, 600,1650, 650,1650, 600,550, 600,550, 600,550, 600,550,
600,550, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,1700, 600,1650,
600,1700, 600,1700, 600,1650, 600,1700, 600};
```

Pr -

```
unsigned int  rawData[67] = {9150,4500, 650,500, 600,550, 650,1650, 600,550,
600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,1700,
600,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,550, 600,550,
600,500, 650,500, 650,500, 650,500, 600,550, 650,1650, 600,1700, 600,1650,
600,1700, 600,1650, 650,1650, 600,1700, 600};
```

With this information, we are ready to assemble and instruct the Arduino remote.

Voice-controlled TV: Voice-controlled remote: hardware

After we have decoded the ordinary remote, we re-utilize the same board to assemble the Arduino remote. The overall hardware essentially consists of an IR LED connected to a MCU enabled with WiFi and Bluetooth connectivity. In spite of this complexity, there is not much circuitry to put together.

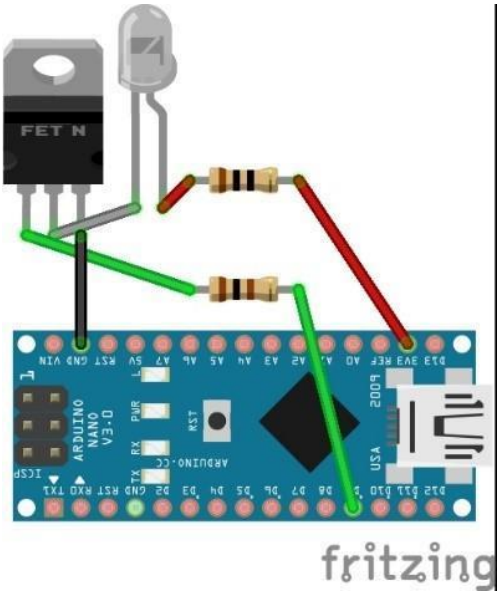


Figure: hardware required for the IR remote decoder, CC BY-SA by INDEX consortium

In order to use the IRremote library without modification to create an IR transmitter, the cathode of the IR LED should connect to digital pin 9 of the Arduino Nano 33 IoT board. Here, we have used a transistor and attached digital pin 9 to its gate instead, because each GPIO pin of the Arduino Nano 33 IoT is able to source a maximum DC current of 7 mA, but a typical IR LED may draw around 100 mA. Note that we have taken as external circuitry a path between the onboard 3.3 V output and ground, which is fit to support a DC current up to around 500 mA in a typical power supply configuration.

Resistors are used to ensure the appropriate current limitations. In particular, the choice of the resistor in series with the IR LED depends on its forward voltage (V_{fwd}), the desired current (I) and the supply voltage (V_{out}), according to $R = (V_{out} - V_{fwd}) / I$. For instance, with a supply voltage of 3.3 V, a desired current of 100 mA and a typical forward voltage around 1.6 V for a low-cost IR LED, the ideal resistor should exhibit R = 17 Ω.

We are now ready to move to the software part of the project.

Using Arduino in IoT projects

Voice-controlled TV: Voice-controlled remote: Arduino IoT Cloud Service and software

This part of the project is the most complex and covers the configuration of the appropriate topic in the Arduino IoT Cloud Service as a MQTT broker, the connection of the Arduino remote to a WiFi network of choice and subscription to that topic as a MQTT client, and programming the behaviour of the Arduino remote in response to new messages by flashing according to the decoded consumer IR protocol.

The whole process is fully guided from the Arduino IoT Cloud Service. In practice, setting up a new topic and selecting a particular board generates an automatic sketch that covers the connection to a certain WiFi network, and the configuration of the MQTT client, and also features a template for a callback function triggered in response to new messages.

The first step consists of setting the appropriate framework in the Arduino IoT Cloud Service by registering the desired board, creating a so-called thing, which is a digital representation of the actual edge device, and defining its set of properties, which are topics available for subscription that may typically trigger the behaviour of the MCU. In this example, there will be a unique property representing the entire TV set, mounted as “TV” property type under “Smart Home” category, given “Read & Write” permission, and scheduled for update “When the value changes”. In practice, the MQTT broker is ready to notify any modification in the TV topic to the MQTT client installed in the chosen board. It will be the duty of that board to react to such notification. Note that the TV property is also visible from a browser logging into a so-called dashboard in the Arduino IoT Cloud Service as another client. In this particular example, the only widget available to interface to the TV property is an on-off toggle. Other property types enable a larger selection of tools, including charts, gauges, sliders etc. to read or write values.

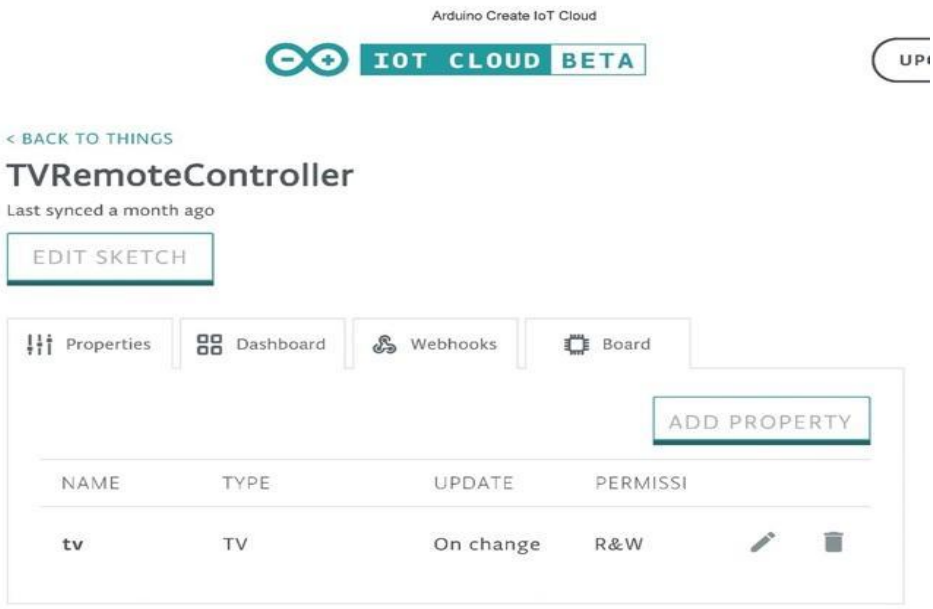


Figure: overview of the TV property created in the Arduino IoT Cloud Service, CC BY-SA by INDEX consortium

With this minimal information, the Arduino IoT Cloud Service automatically creates a sketch template ready for implementation and uploading into the selected board. This sketch template is accessible through a link to the Arduino online IDE and comprises different items. One item is a so-called Secret tab, where the user can enter the SSID and password of a WiFi network of choice without any risk to expose sensitive data to a broad community of makers in the case of voluntary sharing of the project. Another item is a thingProperties.h file containing high-level instructions to ensure connection to the web, initiation of the MQTT client and subscription to the appropriate topic, and should not require editing. Besides a useful ReadMe document, the last item is a .ino file that the user can edit with the usual setup and loop functions, as well as a callback function named “onTVchange”, which will come into play whenever the TV property changes, and will dictate the response of the Arduino remote. In the next unit, we will make sure that the TV property changes when the Amazon Web Service inputs new values for the channel, volume, etc according to valid voice commands. Here, we propose an example for the .ino file:

```
/*
  Sketch generated by the Arduino IoT Cloud Thing "TVRemoteController"
  https://create.arduino.cc/cloud/things/994d9efc-9c38-4a4c-9d3c-454106da00f6
```

Arduino IoT Cloud Properties description

The following variables are automatically generated and updated when changes are made to the Thing properties

CloudTelevision tv;

Properties which are marked as READ/WRITE in the Cloud Thing will also have functions

which are called when their values are changed from the Dashboard.

These functions are generated with the Thing and added at the end of this sketch.

*/

// preliminary instructions: inclusion of files and libraries; initialization of arrays of unsigned int representing the IR codes for the various commands cracked in the first part of this project. Note that the IR codes for channels 0 to 9 are wrapped in a super array; creation of an instance of IRsend object to manage IR data transmission; creation of global variables to store property values

```
#include "thingProperties.h"

#include <IRremote.h>

const unsigned int chan[10][67] = {

  {9150,4500, 600,550, 600,550, 600,1700, 600,550, 600,500, 650,500, 650,500,
650,500, 650,1650, 600,1700, 600,550, 600,1650, 600,1700, 600,1650, 650,1650,
```

600,1700, 600,550, 600,550, 600,550, 600,550, 600,1650, 600,550, 600,550, 600,550, 600,1700, 600,1650, 650,1650, 600,1700, 600,500, 650,1650, 600,1700, 600,1650, 650},

{9150,4500, 600,550, 600,550, 600,1650, 650,500, 650,500, 600,550, 650,500, 600,550, 600,1700, 600,1650, 650,500, 650,1650, 600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,550, 600,550, 600,550, 600,1700, 600,550, 600,550, 600,500, 650,500, 650,1650, 600,1700, 600,1650, 600,550, 650,1650, 600,1700, 600,1650, 600},

{9150,4500, 600,550, 600,550, 600,1700, 600,550, 600,500, 650,500, 650,500, 650,500, 650,1650, 600,1700, 600,550, 600,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,1650, 600,550, 600,550, 600,1700, 600,550, 600,550, 600,550, 600,1650, 600,550, 600,1700, 600,1700, 600,500, 650,1650, 600,1700, 600,1650, 600},

{9150,4550, 600,500, 650,500, 650,1650, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,1700, 600,1650, 650,1650, 600,1700, 600,1650, 600,1700, 600,1650, 600,1700, 600,1650, 650,500, 600,550, 650,1650, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,1700, 600,1650, 600,1700, 600},

{9150,4500, 600,550, 600,550, 600,1650, 650,500, 650,500, 650,500, 650,500, 600,550, 600,1700, 600,1650, 650,500, 650,1650, 600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,550, 600,550, 600,550, 600,1700, 600,550, 600,550, 600,550, 600,1650, 600,550, 600,1700, 600,1650, 650,500, 650,1650, 600,1650, 650,1650, 600},

{9150,4550, 600,550, 600,550, 600,1650, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650, 600,550, 600,1700, 600,1650, 650,1650, 600,1700, 600,1650, 600,1700, 600,550, 600,1700, 600,500, 650,1650, 600,550, 600,550, 600,550, 600,550, 600,1700, 600,550, 600,1650, 600,550, 600,1700, 600,1650, 650,1650, 600},

{9150,4550, 600,550, 600,500, 650,1650, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,1700, 600,1650, 650,1650, 600,1700, 600,1650, 600,550, 600,1700, 600,1650, 650,500, 650,1650, 600,550, 600,550,

600,550, 600,1650, 650,550, 600,500, 650,1650, 600,600, 550,1700, 600,1650, 600,1700, 600},

{9150,4500, 600,550, 600,550, 600,1700, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,550, 600,1650, 650,1650, 600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,1700, 600,550, 600,1650, 650,500, 650,500, 600,550, 650,500, 600,550, 600,550, 600,1700, 600,550, 600,1650, 600,1700, 600,1700, 600},

{9150,4500, 600,550, 600,550, 600,1650, 650,500, 650,550, 600,500, 650,500, 600,550, 600,1700, 600,1700, 600,550, 600,1650, 600,1700, 600,1650, 600,1700, 600,1700, 600,550, 600,550, 600,500, 650,1650, 600,1700, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,1650, 650,550, 600,500, 650,1650, 600,1700, 600,1650, 600},

{9150,4550, 600,550, 600,500, 650,1650, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,1700, 600,550, 600,550, 600,1650, 600,1700, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550, 600,550, 600,1700, 600,1650, 600,1700, 600}

};

const unsigned int volUp[67] = {9150,4550, 600,550, 600,550, 600,1650, 650,500, 600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650, 650,500, 600,1700, 600,1700, 600,1650, 600,1700, 600,1650, 650,500, 650,1650, 600,550, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 600,550, 650,1650, 600,1700, 600,1650, 600,1700, 600,1650, 650,1650, 600};

const unsigned int volDown[67] = {9150,4500, 600,550, 600,550, 600,1650, 650,500, 650,500, 600,550, 650,500, 600,550, 600,1700, 600,1650, 650,500, 650,1650, 600,1700, 600,1650, 600,1700, 600,1700, 600,1650, 600,1700, 600,550, 600,550, 600,550, 600,550, 600,550, 600,550, 600,550, 600,500, 650,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,1650, 600};

const unsigned int chanUp[67] = {9150,4500, 650,500, 650,500, 600,1700, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,550, 600,1700, 600,1650, 600,1700, 600,1650, 650,1650, 600,550, 600,550, 600,550,

```
600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,1700,
600,1650, 600,1700, 600,1700, 600,1650, 600,1700, 600};

const unsigned int chanDown[67] = {9150,4500, 650,500, 600,550, 650,1650,
600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 650,1650, 600,550,
600,1700, 600,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,550,
600,550, 600,500, 650,500, 650,500, 650,500, 600,550, 650,1650, 600,1700,
600,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600};

const unsigned int onoff[67] = {9150,4500, 600,550, 600,550, 600,1700,
600,550, 600,500, 650,500, 650,500, 650,500, 600,1700, 600,1700, 600,550,
600,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,550, 600,550,
600,1650, 600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650, 650,1650,
600,550, 600,1650, 650,1650, 600,1700, 600,1650, 600};

const unsigned int mute[67] = {9150,4500, 650,500, 600,550, 650,1650,
600,550, 600,550, 600,550, 600,550, 600,550, 600,1650, 600,1700, 600,550,
600,1650, 650,1650, 600,1700, 600,1650, 650,1650, 600,1700, 600,550, 600,550,
600,1650, 600,550, 600,550, 600,550, 600,550, 600,550, 600,1700, 600,1650,
600,550, 600,1700, 600,1650, 600,1700, 600,1700, 600};

IRsend irsend;

const int freq = 38;

bool first;

int prevChannel;

int prevVolume;

bool prevSwitch;

bool prevMute;


// the setup() function enables serial communication, calls functions defined in the
thingProperties.h file to connect to the web and the Arduino IoT Cloud Service, and to
```

subscribe to the topic of interest; initializes a variable used for synchronization; and sets the LED_BUILTIN pin as output.

```
void setup() {
  Serial.begin(9600);
  delay(1500);
  initProperties();
  ArduinoCloud.begin(ArduinoIoTPreferredConnection);
  setDebugMessageLevel(2);
  ArduinoCloud.printDebugInfo();
  first = true;
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop() function calls a method that ensures the appropriate connection
void loop() {
  ArduinoCloud.update();
}

// the sendIR() function takes an array of unsigned int as parameter, and exploits the IRsend library to transmit a corresponding pattern of IR pulses through digital pin 9. Meanwhile the built-in LED is set on
void sendIR(const unsigned int buf[]) {
  digitalWrite(LED_BUILTIN, HIGH);
```

```
    irsend.sendRaw(buf, 67, freq);
    delay(300);
    digitalWrite(LED_BUILTIN, LOW);
}


// the onTVChange() function is the callback function called whenever the TV property changes
void onTvChange() {
    Serial.println("==================");
    Serial.println("Switch: "+String(tv.getSwitch()));
    Serial.println("Volume: "+String(tv.getVolume()));
    Serial.println("Channel: "+String(tv.getChannel()));
    Serial.println("Mute: "+String(tv.getMute()));
    Serial.println("==================");


// initial synchronization after powering up or resetting
    if(first){
        prevSwitch = tv.getSwitch();
        prevVolume = tv.getVolume();
        prevChannel = tv.getChannel();
        prevMute = tv.getMute();
        first = false;
```

```
        return;
    }


// if volume has changed, update mute state, send IR signals through the sendIR() function for volume up or down repeatedly until needed, and update volume state
    if(tv.getVolume() > prevVolume)
    {
        tv.setMute(false);
        prevMute = false;
        for(int k = prevVolume + 1; k <= tv.getVolume(); k++)
        {
            sendIR(volUp);
            Serial.println("Volume requested: "+String(tv.getVolume())+", set: "+String(k));
        }
        prevVolume = tv.getVolume();
    }
    else if(tv.getVolume() < prevVolume)
    {
        tv.setMute(false);
        prevMute = false;
        for(int k = prevVolume - 1; k >= tv.getVolume(); k--)
```

```
    {
        sendIR(volDown);
        Serial.println("Volume requested: "+String(tv.getVolume())+", set: "+String(k));
    }
    prevVolume = tv.getVolume();
  }

// if mute has changed, update mute state, and send IR signals through the sendIR() function for mute
  if(tv.getMute() != prevMute) {
    prevMute = tv.getMute();
    sendIR(mute);
    Serial.println("Mute changed: "+String(tv.getMute()));
  }

// if channel has changed, send the appropriate combination of IR signals through the sendIR() function for the various digits, and update channel state
  if(tv.getChannel() != prevChannel) {
    int newChannel = tv.getChannel();
    if(newChannel > 0 && newChannel < 10)
    {
        sendIR(chan[newChannel]);
```

```
    }
    else if(newChannel >= 10 && newChannel < 100)
    {
        sendIR(chan[newChannel / 10]);
        sendIR(chan[newChannel % 10]);
    }
    else if(newChannel >= 100 && newChannel < 1000)
    {
        sendIR(chan[newChannel / 100]);
        sendIR(chan[(newChannel % 100) / 10]);
        sendIR(chan[newChannel % 10]);
    }
    prevChannel = newChannel;
    Serial.println("Channel changed: "+String(tv.getChannel()));
}

// if on/off switch has changed, update on/off switch state, and send IR signals through the sendIR() function for on/off
  if(tv.getSwitch() != prevSwitch) {
    prevSwitch = tv.getSwitch();
    sendIR(onoff);
    Serial.println("Switch changed: "+String(tv.getSwitch()));
```


```
    }


}
```

In practice, the Arduino remote is set to grab the desired configuration through the MQTT client, make a comparison with previous configuration, in order to decide what set of commands to transmit, and emulate the ordinary remote in all and for all.

In the next unit, we will cover the connection of the Arduino IoT Cloud Service to the Amazon Web Service, in order to change the TV property by relevant voice commands.

Voice-controlled TV: Voice-controlled remote: Arduino Alexa Skill

This unit briefly covers the creation of a link between the Amazon Voice Service and the Arduino IoT Cloud Service.

We assume that a link is already in place between a smart speaker such as an Amazon Echo Dot as an endpoint and the Amazon Voice Service over a WiFi network.

The task to establish the final link between the Amazon Voice Service and the Arduino IoT Cloud Service is committed to a so-called webhook implemented through an AWS lambda function. In practice, that is a callback function written in Node.js, Python or Java that is triggered by a predefined event, and authenticates and sends data to a desired URI. In our case, the predefined event is a valid command to reconfigure a TV, and the desired URI is the relevant topic in the Arduino IoT Cloud Service. The latter is pinpointed as an endpoint, i.e. a device. In a webhook, data are typically formatted in a JSON structure and wrapped in a HTTP POST request.

The procedure is fully guided at the highest level of abstraction, and the user does not need to worry about the underlying machinery. By using a smartphone to install and configure the Arduino Alexa Skill as a smart home utility in the Amazon Alexa App, the procedure will allow the user to create a link between the predefined Amazon Alexa and Arduino IoT Cloud Service accounts, and to discover and enable the TV as a new device. Of course, such a device is the virtual representation of the TV

properties combined in the appropriate topic in the Arduino IoT Cloud Service. The Amazon Web Service is already set up to automatically create the due AWS lambda function.

At this point, everything is set to replace the ordinary remote with its voice-controlled upgrade.

Voice-controlled TV: Conclusions

The voice-controlled TV is now configured to respond to such commands as “Alexa turn TV on”, “Alexa mute TV”, etc.. Simply power and place the Arduino remote within sight of the TV and start to talk to the smart speaker in natural language.

The experimenter may probably need to replace the set of IR codes to reflect the buttons available in the ordinary remote of the target TV, and even try to hack other IR controlled devices. We warn the trainee that the use of contributed libraries like IRremote may entail some pitfalls, as the definition of variables, functions and methods may change from version to version. Therefore, we cannot promise that the code presented in this series of units will work without modification.

With respect to the previous series of units, the scope of this project is rather narrow. As we have explained, we have chosen this case to exemplify a representative situation, where the pursuit of a specific task of high added value may benefit from the integration of higher and lower-level functionalities made available through web services that may provide resources as complex as speech recognition or as flexible as generic brokerage and webhooks. There is such an ever-wider range of offers and solutions as to satisfy most conceivable needs with relatively minor adjustments and add-ons. Even though the development of industrial platforms requires the intervention of qualified experts and specialized companies, repositories as the Arduino Project Hub at https://create.arduino.cc/projecthub or the Adafruit Learning System at https://learn.adafruit.com/ may be a source of inspiration for hobbyists, but also serve as excellent blueprints to understand the potential of IoT at all levels.

Exercise

Suppose someone has installed several Arduino remotes for friends and relatives, but later the Amazon Web Service has cut off its built-in support to the Arduino Alexa Skill. What would be the most straightforward strategy to start with?

☐ To replace the Amazon Web Service with another large-scale infrastructure like Google Cloud Platform.
☐ To implement the automatic speech recognition directly through the Arduino IoT Cloud Service .
☐ To browse the web in search of a new firmware for the Arduino remote, without many other changes.

☐ To consider the implementation of a webhook from a third-party go-between like If This Then That, if possible, in order to create a custom connection between the Amazon Web Service and the Arduino IoT Cloud Service

Final Exam

Internet protocol suite 2

Which layer of OSI model do functions like authentication and authorization belong to?

☐ Layer 5 (Session)
☐ Layer 7 (Application)
☐ Layer 6 (Presentation)
☐ Layer 4 (Transport)

Connections 6

Why does broadband cellular connectivity still play a fundamental role for IoT?

☐ For its global reach, scalability, and high bandwidth capabilities
☐ For its global reach and low costs.
☐ For its high bandwidth capabilities and low power consumption.
☐ For its scalability, low costs and low power consumption.

Connections 3

Over which frequency band does Bluetooth transmit data?

☐ 5 GHz.
☐ 2.4 GHz.
☐ 1 GHz.
☐ 3.4 GHz.

Application layer protocols 3

In RESTful architectures:

☐ The verb GET allows a client to replace an item within a collection.
☐ The verb GET is safe or nullipotent, because it does not alter anything on the server side.
☐ The verb GET allows a client to create a new item within a collection.

INDUSTRIAL EXPERT

☐ The verb GET returns the state of a server, when it makes use of so-called cookies.

Services 4

What is not a typical use case for artificial intelligence?

☐ Solving linear equations.
☐ Image recognition in photographs.
☐ Spam filtering.
☐ Predictive maintenance.

Services 2

What is not a typical function of a web service?

☐ Routing and forwarding data packets through intermediate routers.
☐ Providing a human interface, such as a dashboard.
☐ Implementing communication among end-nodes in a network.
☐ Managing configuration changes, such as over-the-air updates.

Hands on with microcontrollers 10

What is the main difference between I2C and SPI communication protocols?

☐ I2C bus requires 2 wires and SPI bus at least 4, plus ground connections in both cases.
☐ I2C protocol is simplex and the SPI protocol is half-duplex.
☐ I2C protocol runs over UDP and SPI protocol over TCP.
☐ I2C protocol is implementable in most Arduino boards, while SPI protocol is unsuitable for constrained contexts.

Hands on with microcontrollers 4

What basic functions does an Arduino C/C++ sketch need to include for compiling?

☐ A setup() function and a loop() function.
☐ A setup() function and an #include statement.
☐ A loop() function and a For loop.
☐ There is no fixed structure in an Arduino C/C++ sketch.

Hands on with microcontrollers 2

Co-funded by the
Erasmus+ Programme
of the European Union

Which of the following modules does not belong to a typical MCU architecture?

☐ Graphics processing unit.
☐ General-purpose input/output ports.
☐ Program memory.
☐ Data memory.

Hands on with microcontrollers 1

Which of the following features is not downscaled from typical microprocessors to MCUs?

☐ Connectivity options.
☐ Maximum clock speed.
☐ Maximum processing capacity.
☐ Price per piece.