Module 1. Overview of Robotics and Operating Systems

Overview

Learning Outcomes:

Upon completion of this module, students will:

- Understand robotics technologies conversationally, including key components and how they function together.
- Comprehend the role of operating systems (OS) in robotics, particularly real-time OS (RTOS).
- Develop trust in the OS's ability to manage robot tasks efficiently and in real-time.

What to Expect:

This module defines robotics, introduces robotic components, and covers robotics' multi-disciplinary nature. You'll learn how computers, software, electronics, and networks contribute to a robot's function, and explore differences between remote-controlled, fully autonomous, and hybrid robots. This module examines operating systems in robotics, focusing on RTOS, and compares general-purpose OS and embedded systems.

After each lesson, there is a self-assessment where you can see what knowledge you were able to gain from this content. This self-assessment helps the writers and developers of IRAP to see where we can do better in preparing educational material.

Towards the end of this module, there is a lab where you will put your new knowledge into practice by setting up a Raspberry Pi computer with a fresh Linux OS and practicing some basic file operations. You'll also install and explore real-time operating systems (RTOS) on an Arduino board. Most of the hands-on efforts of this course will be presented in the lab section of each module.

Finally, there is a survey that is meant for you to provide feedback on your experience going through this module. Don't worry; there's only seven questions, and four of them are click-throughs. This survey helps the writers and developers of IRAP know the value of our product and whether it is relevant to real-world applications. Completing the survey is required for you to advance to the next module.

Your Workstation:

As you progress through this module and the rest of the course, please be aware that you will be reusing most of the same equipment in all the labs. You will want to have a safe, clean, dry, and temperature-controlled space for your equipment to stay assembled.

Support Forums:

In the top, center of the IRAP website header, there is a link to the support forums for this course. Only those enrolled in the IRAP course and signed in can access the forums at this link. If you need help with anything concerning IRAP, please check these forums first. Someone may have already experienced the same issue or had the same question – and others may after you.

There are three main forum categories: "Course Content," "Troubleshooting Labs," and "Website Functionality." Within each of the categories, the topic titles are where you will discover if your question already exists. If it doesn't, we ask that you create a new topic title instead of adding a comment to an existing topic for an unrelated matter. Staff members should respond to comments on this support forum before the end of the following business day.

Materials:

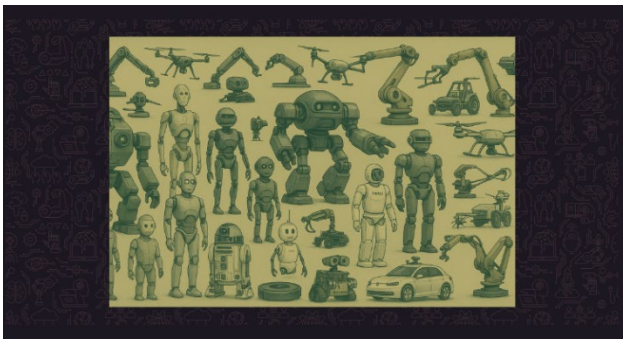
CrowPi2 Laptop, Raspberry Pi 5 8GB Computer, 64GB MicroSD Card, USB-A to µSD adapter, screwdriver kit, and Arduino Uno

Table of Contents

Introduction to Robotics

Defining Robotics

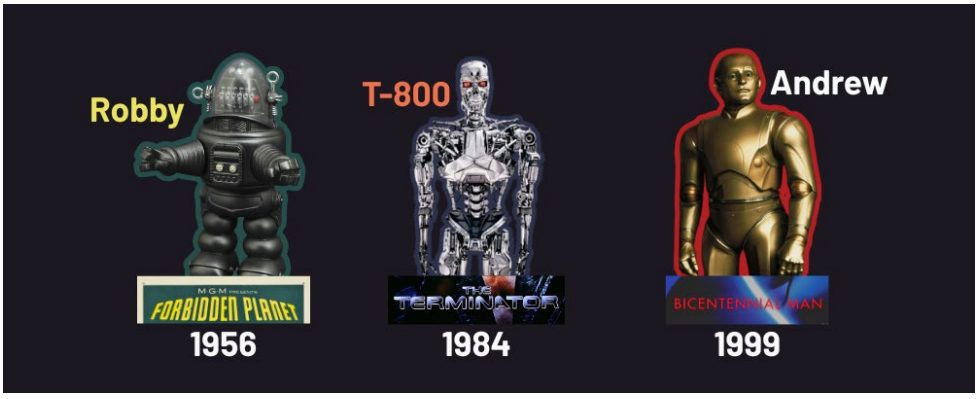
Reality vs. Fiction



Robotics is a multidisciplinary field; it's not just about mechanical parts or circuits. Robotics combines mechanical and electrical engineering with computer science to create intelligent systems. Definitions of "robotics" vary slightly. Oxford calls it a branch of technology; Britannica emphasizes task automation, and Wikipedia and Merriam-Webster focus on design and use. Despite the variations, they converge on a common idea: robotics involves designing, building, and operating machines capable of performing tasks independently or semi-independently.

Throughout this course, the term "robotics" should be interpreted broadly to include mobile robots, fixed robots, automobiles, drones, and similar systems with varying levels of autonomous behavior. Whether referring to an unmanned aerial vehicle (UAV) conducting reconnaissance, an assembly robot in a factory, or an autonomous ground vehicle transporting supplies, the principles remain the same.

Consider our first exposure to the idea of robots through science-fiction films designed for entertainment. We've seen robots portrayed as big, clunky devices, or assassins sent through time, or life-size caretakers. As with all science fiction, viewers often see these portrayals as existing only within the creativity of filmmakers—hence the word "fiction" in the genre. The reality, however, is that modern robotic systems are now approaching, if not already matching or exceeding, the capabilities of these fanciful portrayals.

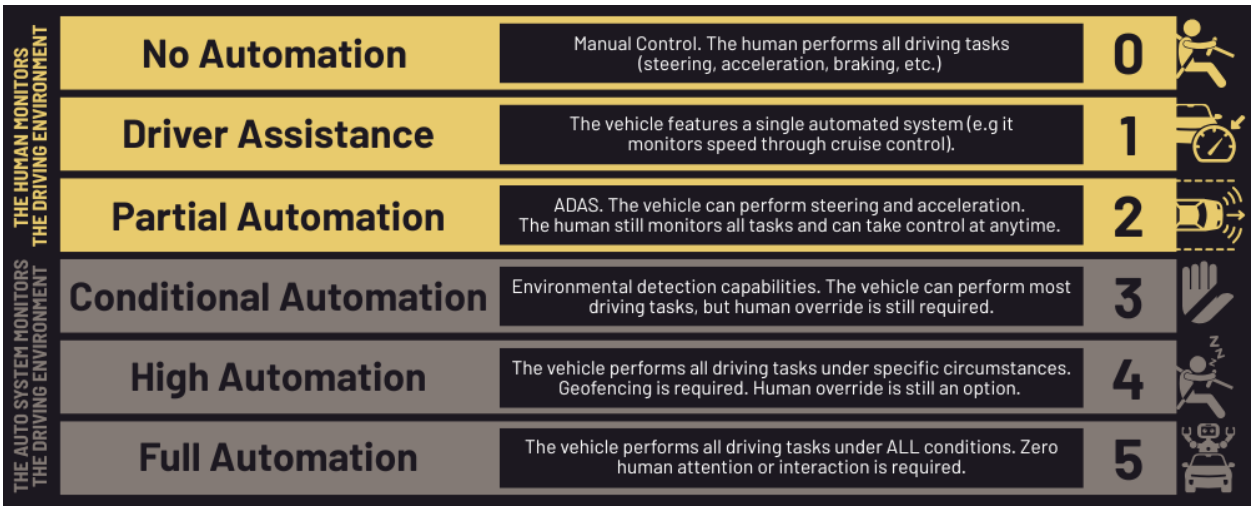


Developing a robot consists of designing and constructing hardware and software systems that work together to complete a task. For example, **computers** are the brains for the entire system, processing information and controlling a robot's physical movements. **Networks** are the connective tissue that enable robotic systems to communicate (i.e., send and receive data) between on-board components or with other robots, external systems, and human operators. **Software and artificial intelligence** (AI) enable both simple robotic autonomy and large ensembles of robotic systems to work together toward a common objective. While all robots have some level of **autonomy**, not all robots share the same autonomous complexity.

Autonomous Systems

Autonomy refers to a system's ability to perform tasks and make decisions with varying levels of direct human control. Autonomy can be as simple as a toy robot following a fixed path or as complex as a flying drone navigating through a dense forest at high speed. Complex networks of systems enable ensembles of robotics, fixed sensors, and other technologies to work together to accomplish complex missions with little human intervention.

Some industries and governments have developed formal definitions to describe various levels of autonomy. For example, the Society of Automotive Engineers has six levels of autonomy.



Autonomy

Remote controlled systems: These systems are typically teleoperated, meaning they rely entirely on human control via remote interfaces. They might include various sensors or decision-assist features that help the operator avoid errors, hazards, or actions that could compromise the system.

- **Example:** Explosive ordnance disposal robots are typically controlled by a remote operator with a joystick, monitoring the actions through a video feed.

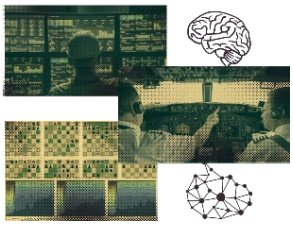
Fully Autonomous Systems: These systems operate completely on their own, using artificial intelligence (AI) to operate, navigate their environment, make decisions, and complete tasks without human input.

- **Example:** The Waymo Self-Driving Taxi is a fully autonomous vehicle with no human driver at all.

Hybrid systems: These are "human-in-the-loop" (HITL) systems where robots carry out some tasks autonomously, but human-operators make or confirm critical decisions, and assist or take over some courses of action. This paradigm is common in safety-critical systems where there is a risk of damage or death.

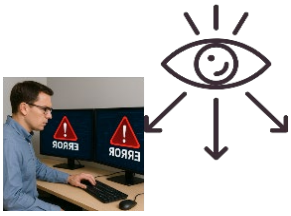
- **Example:** There are robots implemented in the military to survey dangerous areas autonomously, but a human must confirm or control direct engagement with targets.

HITL Systems in Detail



There are three common forms of HITL systems:

- **Supervisory Control:** The human operator monitors and intervenes only when necessary.
- **Shared Autonomy:** Task execution and control are dynamically shared between humans and robots.
- **Training-and-Feedback:** Humans provide feedback and training data to improve the system's performance.



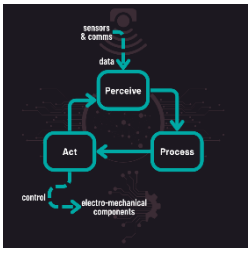
Supervisory Control operates completely autonomously until it encounters a critical situation and turns over control to an operator. Consider a military drone that collects data autonomously. If it encounters an unexpected obstacle, sees a target it doesn't understand, or some other unrecognized event, the operator is notified and can assume control to ensure safe and accurate operation.

Shared Autonomy is a collaborative control model where the human and the autonomous system continuously and simultaneously share control over the task. The system assists the human by interpreting intent, optimizing commands, or correcting errors. This paradigm is often used in robotics, avionics control systems, assistive technologies, or teleoperation. Consider a robotic wheelchair that adjusts the user's steering input to avoid obstacles while still allowing directional control.

Training and Feedback systems are where the autonomy continually learns through feedback loops between the human and the system. As humans provide feedback, the system learns and gets smarter. Consider an autonomous military vehicle in the field. If the vehicle encounters unfamiliar terrain, the soldier can train the vehicle how to traverse it, equipping the system to handle similar situations in the future.

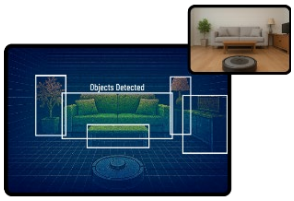
Key & Cyclical Functions

Key Functions

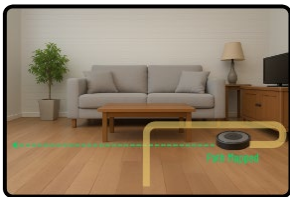


The engineering science of robotic automation focuses on a system's ability to sense, think, and complete a task on its own, with or without some level of human intervention. These abilities make up the three key cyclical functions of an autonomous robot: perceive, process, act.

Robots must have subsystems to receive data and **perceive** the environment in which they operate. This is done using sensors, such as cameras, LiDAR, sonar, and a myriad of other sensors. Sensors gather critical information about a robot's surroundings, in much the same way as we use our five senses to perceive the world around us.



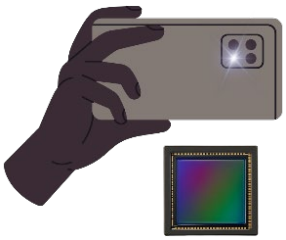
Data from a robot's sensors is often raw, meaning it consists of arbitrary values received directly from the sensors. Robots use computers and software to continuously **process** and interpret how that data affects the robot's operation and designed purpose.



Robots make decisions to **act** based on the interpreted data, such as completing instructions, identifying objects, or knowing where to move. A robot may utilize actuators to move the robot or its limbs, activate tools, or trigger further sensing and communication actions.

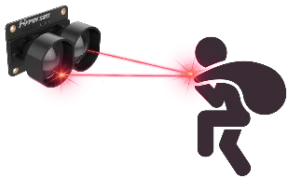
Perception Tools

Sensing: Sensors are key to enabling robotic perception. There are many types of sensors that can detect and measure physical phenomena. Robots usually employ a combination of sensors to collect data enabling them to recognize objects, avoid obstacles, navigate, and understand their operational environment.

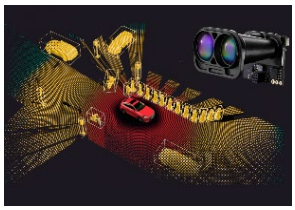


Cameras: Digital cameras (e.g., those in a phone or webcam) capture light and convert it into digital images using an image sensor comprised of millions of tiny light-sensitive elements called pixels. Each pixel measures light intensity. When light hits the pixels, they generate an electrical signal, which is processed into images or videos of the world around us.

- **Example:** The camera in your smartphone uses a CMOS sensor to take photos and videos.



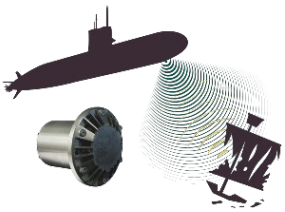
Time-of-Flight (ToF): ToF sensing is a technique used by various sensors to determine distance by measuring how long it takes for a signal of some type (e.g., light, sound, or radio) to travel from a source to an object and bounce back. By timing the round-trip journey, the sensor determines how far away an object is.



LIDAR: LIDAR stands for Light Detection and Ranging. It is a ToF sensor that works by sending out laser pulses (i.e., light waves) and measuring how long it takes for the light to bounce back from objects. This helps create detailed 3D maps of an environment by calculating distances to things like trees, walls, buildings, or the ground.

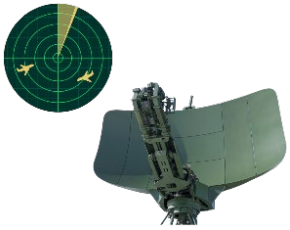
- **Example:** Self-driving cars use LIDAR to "see" the road and obstacles around them.

Sonar: Sonar is another type of ToF sensor that uses sound to measure distance much like a bat flying through the night sky. Sonars emit sound waves that travel through air or water. When they strike an object, they bounce back to the sonar where they are received. To determine distance, sonars measure the round-trip travel time of the sound waves.



- **Example:** Maritime systems like submarines and seafloor mapping systems use sonar to identify objects and distances.

Radar: **RA**dio **D**etection **A**nd **R**anging (RADAR) is another ToF-based sensing system that sends out radio energy from an emitting antenna, which strikes an object and bounces back to a receiving antenna. Radars measure the time of flight for radio waves to hit an object and return.



- **Example:** Air Traffic Control systems use radar to track aircraft position and altitude to safely manage air traffic.

Motion and Position: These sensors provide information about movement, location, and orientation.

- **Accelerometers** measure how quickly something is speeding up or slowing down.
- **Gyroscopes** measure how quickly something is rotating or turning.
- **GPS** uses signals from satellites to figure out where you are on Earth.
- **Inertial Measurement Units (IMUs)** combine multiple sensors that measure movement, acceleration, orientation, and magnetic fields.

Force, Touch, and Tactile: These sensors detect contact, pressure, and/or force. They work in different ways, such as using electricity, light, sound, or magnetism to sense if something has been touched and measure the force of contact.

- **Example:** The touchscreen on your phone can detect the difference between swiping, pressing, or holding an object displayed on the screen.

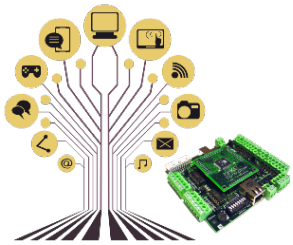
Environmental: Environmental sensors measure things like temperature, humidity, air quality, and even radiation or biological agents. They help monitor conditions in an environment for safety, comfort, or other scientific analysis.

- **Example:** Thermometers are used to control heating or air conditioning to maintain a set temperature.

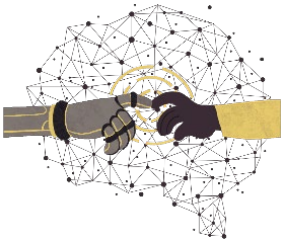
Acoustic Sensors: Acoustic sensors measure pressure variations in some medium (e.g., air, water, materials). A microphone is an example of an acoustic sensor that detects sound vibrations in the air.

- **Example:** Vibroacoustic sensors are attached to machines to monitor for excessive vibrations, which could mean something is wrong or that maintenance is needed.

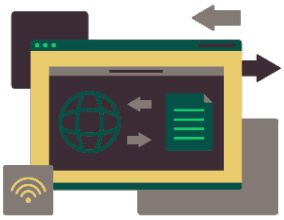
Process



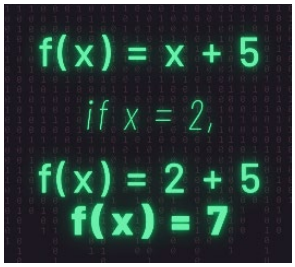
All robots have at least one on-board computer; most utilize multiple on-board computers to process data and manage the actions of a robot. Small on-board microcontrollers manage simple tasks (e.g., collecting data, controlling electromechanical systems). Special-purpose computers may be required to handle specific types of data. Sophisticated computers coordinate overall robot operations. For some applications, off-board (i.e., remote) computers are needed for high-performance computation to manage complex tasks, or to coordinate the actions of multiple robots.



To accomplish complex tasks, robots must communicate with human operators, other robots, and external systems (e.g., enterprise software or cloud services). Networks enable this communication. Robots often include both external communication networks and internal networks that allow subcomponents to coordinate their operations.



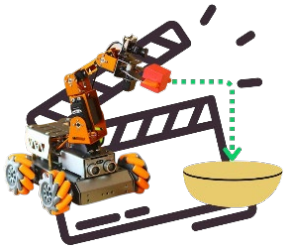
Software turns mechanical hardware and sensors into intelligent, purpose-driven machines. Even the simplest robot has on-board software that accepts and executes user commands. More advanced robots rely on complex software systems to process sensor data, maintain robot state, coordinate actions, communicate, and implement robot locomotion (e.g., movement, obstacle detection, avoidance, path finding).



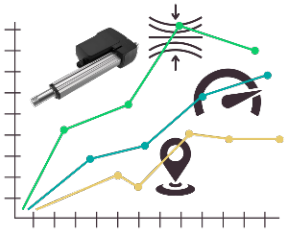
Do you remember functions from algebra? How about this one: F(x) = x + 5. If you set x=2, you get the answer, 7. This is an algorithm, and they are embedded everywhere in robotic software – actually embedded in any software. They enable computers to make decisions and complete tasks the robot was designed for.

Algorithms can be as simple as the example above, or complex machine learning algorithms with thousands of parameters.

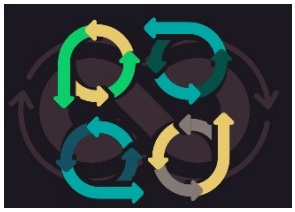
Act



Introduction: In the action phase, a robot responds physically based on its understanding of the environment from processed sensor data and communications information. This may involve moving wheels, adjusting a robotic arm, changing direction, stopping, sending or receiving messages, gathering more sensor data, actuating specialized on-board electro-mechanical equipment, or any behavior appropriate to the situation. These physical actions are the output of the perceive-process-act loop and often generate new sensor data, which restarts the cycle.



Motion and Feedback: Robots have actuators (e.g., motors, servos, pneumatics). As actuators operate, "feedback" data (e.g., position, pressure, velocity) is sent to the computers. This data is necessary for the robot to monitor a motion as it is doing it. For example, as a robotic pincer closes around an object, pressure data from sensors on the pincer is sent to the computer so that the application knows when to stop squeezing and the object is secured.



The Loop: Once the perceive, process, and act steps are complete, the process begins again, operating continuously, forming a loop. In robot systems this loop happens in real-time; thousands of loops can happen every second to ensure the robot operates accurately and in a timely manner. Complex robotic systems have many independent perceive-process-act loops operating concurrently, and many of these loops interact with one another. This makes software design for robotics very challenging.

Robotic Floor Cleaner

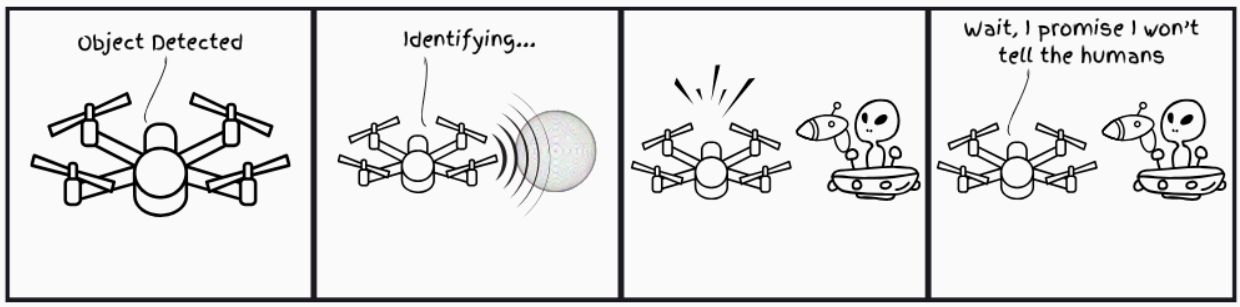
Consider an indoor robot floor cleaner. When activated, it starts moving on some pre-programmed path.

- **This robot perceives:** On-board infrared sensors measure distance; bump sensors detect obstacles; cliff sensors detect drop-offs, power sensors monitor batteries, and dirt sensors to recognize parts of the floor to clean. More advanced robot floor cleaners use cameras or LiDAR systems to map the room.
- **This robot processes**: On-board software analyzes the sensor data to perceive its environment and determine what it should do next. For example, the robot's computer may receive the following alerts:

- Alert: [drop off detected]
- Alert: [obstacle present]
- Alert: [high dirt concentration]

- **This robot acts**: Based on the results of the analysis, the robot's software will instruct and activate various components of the system. For example, the robot may respond in the following ways:
  - Response to cliff: Rotate 180 degrees and advance forward.
  - Response to excessive dirt: Vacuum and travel in a spiral path, starting from the center of a spiral, until excessive dirt is no longer sensed.
  - Response to obstacles: Rotate 45 degrees and advance forward.

This model describes how a relatively simple robot, like a floor cleaner, can function autonomously without user intervention. Yet the same foundational loop of perceive, process, act applies to far more complex systems, such as autonomous drones and industrial arms. Thanks to advances in AI, engineers are building robots that perform highly complex tasks with greater speed, accuracy, and efficiency than humans. For example, modern drones can navigate dense, obstacle-filled environments and carry out reconnaissance or attack missions without human input, relying entirely on sophisticated sensors and software. As we move through the course, we'll explore how AI continues to enhance each stage of this loop, enabling robots to perceive more intelligently, process faster, and act more effectively.

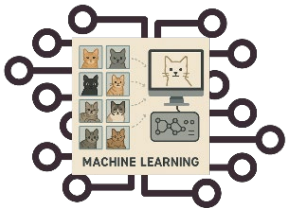


AI and the Cycle of Trust

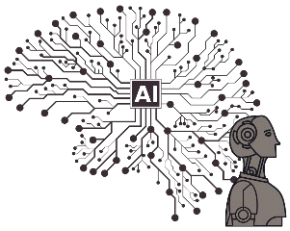
AI vs. ML

Artificial Intelligence (AI) and Machine Learning (ML) are closely related but not identical. AI refers to systems that simulate human intelligence, while ML, a subset of AI, specifically enables systems to learn from data. These technologies are the primary enablers of complex, intelligent, autonomous systems, and robotics platforms.

Together, they utilize massive statistical models to predict outcomes or actions based upon large, complex, input data sets.

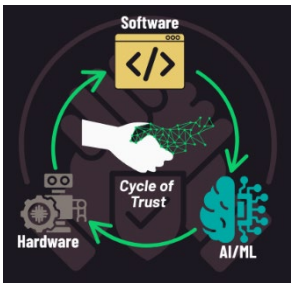


Machine Learning (ML) enables systems to learn by studying abundant data and examples rather than being explicitly programmed. For instance, instead of excessively coding rules for how to recognize a cat in a photo, ML systems learn by analyzing thousands of cat photos.



Artificial Intelligence (AI) is software that often uses massive statistical models, many of which are built with ML to make predictions or decisions based on complex data. AI gives robots the ability to "think, perceive, and learn" by analyzing data, recognizing patterns, and making independent decisions. While some level of autonomy is possible without AI, AI enables systems to adapt to complex, unstructured, and dynamic conditions in unpredictable environments.

The Cycle of Trust



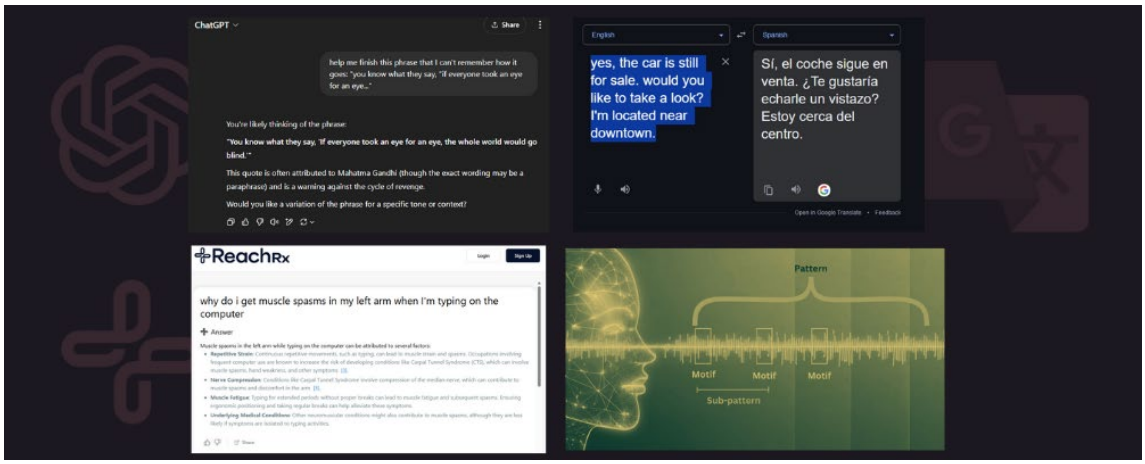
Establishing a cycle of trust between ourselves (i.e., human operators) and autonomous systems is an integral part of a robotic system having purpose. Developing that cycle of trust is a key goal of this course. As such, we will study how robots work, their various subsystems and components, the increasing levels of autonomy, and their impact on design and operation.

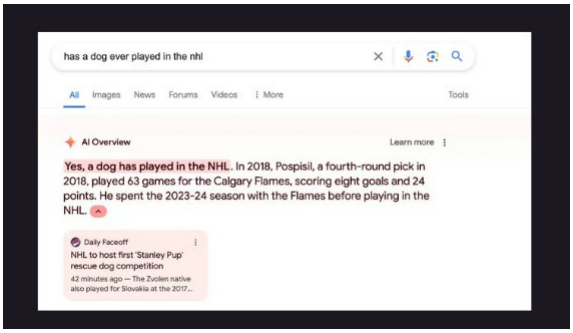
Despite what you see in the movies, AI is not intelligent in the human sense. Understanding what AI is and what it's not, is very important in forming a realistic Cycle of Trust between human operators and intelligent systems. This course aims to help you understand how AI systems work to build a firm foundation for a Cycle of Trust.

AI is a system of math, data, software, and computers able to recognize patterns at a very large scale from large volumes of data. AI must be trained like a child. The training process is akin to having billions of flashcards containing related information (e.g., text, pictures, audio) where a super-fast computer reads through them and identifies patterns faster than possible for a human. Based on the flashcards (training data), the training process builds statistical models enabling AI to make predictions at run time from operational data to do things like:

- Predict the next word in a sentence (LLMs like ChatGPT)
- Estimate the likelihood of a certain medical diagnosis (ReachRX LLM)
- Recognize patterns in images (object identification) or sounds (speech recognition)

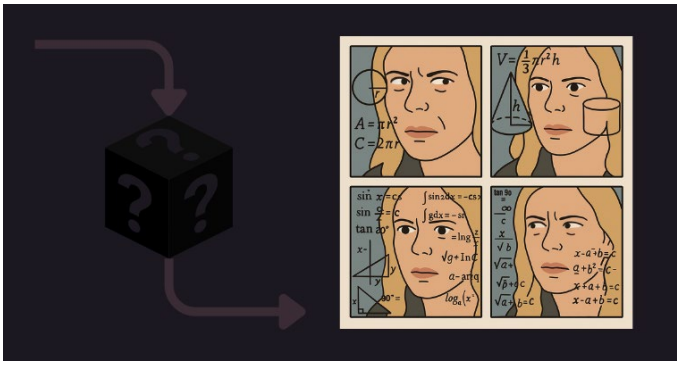
- Translate languages (Google Translate)





AI does not understand context. It does not have emotions or self-awareness. It can't think creatively or morally. AI has no goals without humans providing them, and it doesn't know why it's doing what it's doing. AI does not really "know" anything. For example, AI does not understand what a cat is. It's able to determine statistically that certain pixel patterns in an image likely mean there's a cat in the picture. These not-so-obvious limitations are important to understand in terms of building a Cycle of Trust with an intelligent system – especially in safety/mission critical situations.

AI can be wrong… and confidently so! When an AI provides an answer, it can sound confident and even provide references, but that doesn't mean it's correct. AI can mislabel images, generate false information, and make up references. These hallucinations occur when AI generates something that is statistically plausible but in reality, is incorrect. AI doesn't understand meaning or intent like humans do. It simply predicts the next most likely word, action, image pixel, etc., based on statistical patterns in the training data. If an AI model was trained on lots of texts about "Harvard," it might start a sentence with "Harvard researchers found…" — even if no such research exists. The AI is not lying — it just has no concept of truth.

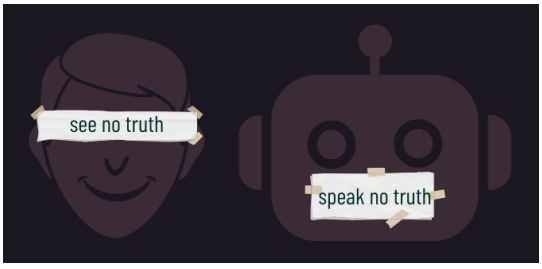


Another aspect to be aware of is that humans don't often know how an AI system arrives at the answers it formulates. This is referred to as "The Black Box Problem." Modern AI models (especially deep learning systems) are far too complex to fully interpret. These systems can have billions of parameters and layers of computation in their underlying model. Even engineers who build these systems can't explain exactly why a particular decision was made.

For these reasons, building a Cycle of Trust is essential when using AI systems, like robots and drones, to ensure they are used responsibly. A Cycle of Trust is a feedback loop where humans and AI learn to rely on each other safely and responsibly. The Cycle of Trust includes continual observation, evaluation, and validation:

- **Observation**: Testing and observation of the AI in an operational environment. This includes normal operational use cases and testing the edge cases of the system.
- **Evaluation**: Human assessment of whether the AI's response is reasonable and appropriate in complex situations. Are the behaviors correct and consistent?
- **Validation**: Where and how does the system fail? Identify where "guardrails" may need to be developed to ensure safe ethical operation and where refinement of the model and/or system is necessary.

This process begins on day one and continues until the system is decommissioned. Because both the world and operational context change, AI models must be regularly retrained. After the AI is retrained, the Cycle-of-Trust process begins anew. In any system where human operators work alongside AI, trust must be earned and not assumed.

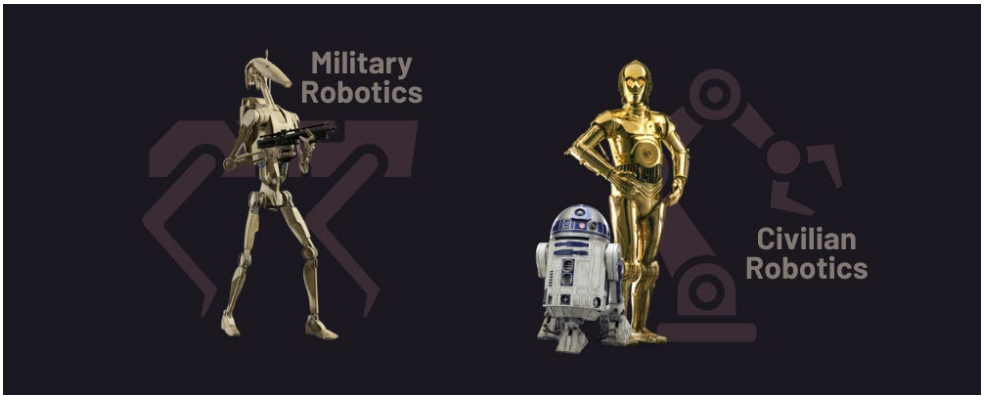


In summary, blind trust is dangerous and assuming AI is always correct leads to harmful mistakes. Total distrust is wasteful. If you ignore useful AI insights, you lose productivity, speed, and capability. A balanced, feedback-driven relationship where humans and AI support each other is the sweet spot. AI handles scale and speed, and humans provide judgment, oversight, and ethical grounding.

Military vs. Civilian Robotics

The integration of AI and ML into military systems enhances decision-making, situational awareness, and autonomous operations. These technologies enable robots to learn from their environments, adapt to new situations, learn, and make complex decisions in real time. For example, AI can help military robots identify threats, analyze terrain, and even predict and react to enemy movements. The key differences between military and civilian robotics lie in their purpose, operational environments, and levels of risk:

- **Military robotics** focuses on combat, surveillance, and logistical operations in hostile environments where human lives and military operations are at stake, and advanced autonomy is utilized to adapt to such conditions. A key goal for military robotics is for individual robots to operate together (teaming or collaborative autonomy) and interact with human operators safely, efficiently, and with trust.
- **Civilian robotics**, on the other hand, focuses on improving efficiency, productivity, and service in environments like manufacturing, homes, businesses, transportation, and hospitals. The risk factors can be lower but often involve safety



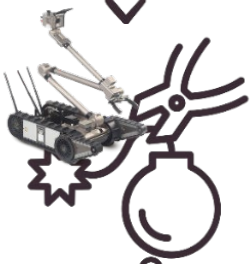
critical factors. Other factors include concerns about privacy, safety, intellectual property, and data security.

The differences are quite subtle and have significant overlap. There are civilian autonomous systems used in safety critical systems where human life is at stake, and many robotics and AI technologies developed for civilian use, have been adapted for military use, as there are capable, cost-effective, and rapidly deployable solutions in highly dynamic environments. In recent history, this has been an effective strategy against better financed, equipped, and numerically superior forces.

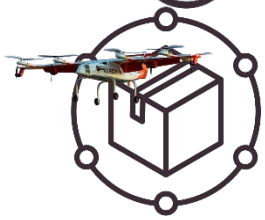
Military Applications

Ground, Aerial, and Underwater Robots: These robots serve multiple functions, from reconnaissance to combat support. For example, unmanned aerial vehicles (UAVs), like the MQ-9 Reaper drone, are used for surveillance and targeted strikes. Autonomous ground vehicles such as the Robotic Mule (R-MULE) are designed to carry heavy loads across difficult terrain, following soldiers and freeing them from carrying physical burdens.

Surveillance and Reconnaissance: Robots equipped with high-resolution cameras, infrared sensors, and LiDAR, like the ScanEagle Drone, can gather intelligence, monitor enemy movements, and provide situational awareness in real time.



Explosive Ordinance Disposal (EOD): Specialized robots such as PackBot and Andros are designed for EOD missions. These robots work with human operators to locate, disarm, and dispose of explosive devices, saving lives and preventing damage to infrastructure and equipment.



Logistics Support: Autonomous systems for transporting supplies have emerged from civilian markets for disaster relief missions. The Silent Arrow and Elroy Air Chaparral are examples of autonomous cargo delivery systems being adopted by the military. These systems are capable of autonomous deployment and flight for military resupply in contested or remote environments, post-disaster relief delivery, and humanitarian operations.

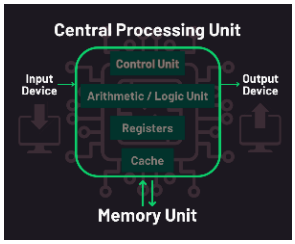
Fundamentals of Operating Systems

Basic Components

Complex Machines

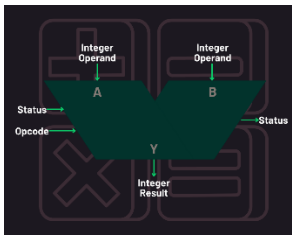
Robots are complex machines made up of multiple computer systems that range from enabling basic movement to supporting advanced capabilities like artificial intelligence (AI) and machine learning (ML). While not all computer systems require an operating system (OS), most rely on one to manage hardware resources and coordinate tasks. An OS is a computer program whose responsibility is 1) to provide access to a computer's hardware, 2) schedule applications to run, and 3) ensure applications have the resources they need to execute. Operating systems are made up of several key components that work together to manage a computer's resources and allow programs to run smoothly. An understanding of operating systems is crucial because complex autonomous systems manage many applications at once. Each computer must schedule applications, allocate resources (e.g., memory), interact with sensors, and control hardware subsystems in real time. In this lecture, we'll introduce the fundamentals of OS operations to include memory management, file systems, and concurrency.

Computer Hardware

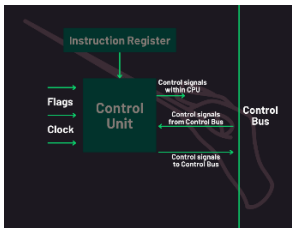


Architecture: Operating systems manage a computer's hardware resources. Before introducing operating systems in any detail, it is essential to understand basic "central processing unit" architecture. As the OS schedules applications to run, it provides access to various hardware components and services, or "resources" that define the computer architecture of the central processing unit.

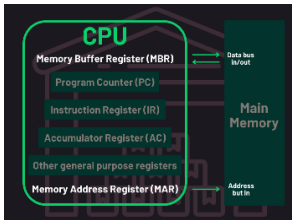
Arithmetic/Logic Unit (ALU): Responsible for all computations and logical operations performed by a computer. This includes all of the basic bit-level manipulation and binary mathematics encoded in a computer program. All of the code that comprises a computer application can be reduced to a small set of instructions carried out by the ALU.



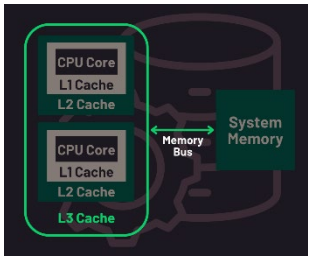
Control Unit: If you think of the components of a computer as an orchestra, the control unit is the conductor. The control unit coordinates the processing of instructions, enabling/disabling various components including the ALU, registers, memory, managing buses, and so forth.



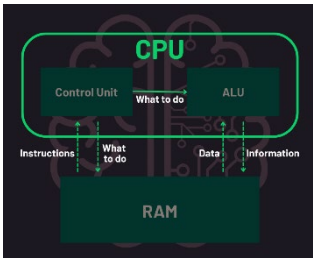
Registers: These are small, fast storage locations (memory) inside the CPU used for storing data used in computations. Registers typically store 32 or 64 bits of data. Computers typically have lots of registers, some of which are general purpose and others that serve special purposes. Programmers typically do not directly access registers unless they are using very low-level programming language (e.g., machine language).



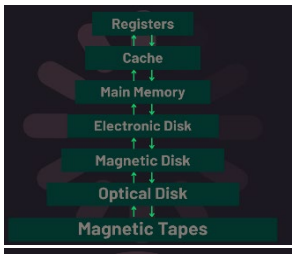
Cache: A small amount of high-speed storage (memory) bridges the gap between the computer and main memory (RAM). There are different types and sizes of cache, storing between 16K bytes to 1M bytes of data. Main memory is typically external to the CPU; whereas cache exists on the CPU and is much faster to access. Cache significantly reduces computational latency by storing frequently accessed data and instructions physically closer to where computation takes place.



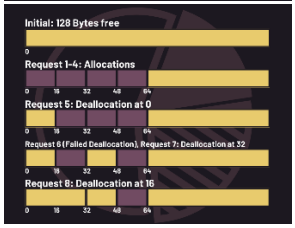
Memory: Memory is where the OS, other applications, and data are stored. Applications generate, save, and retrieve data to and from memory. Computer memory is often called random access memory (RAM) because any piece of information can be accessed directly. The term RAM is an anachronism from the early days of computing when some memory was accessed sequentially.



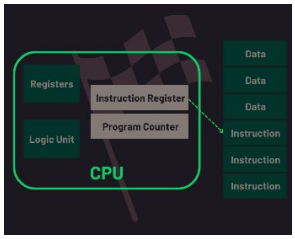
Resource Management



Memory: Memory management is the process by which an operating system allocates, controls, and coordinates access to the computer's memory. When programs are started, the OS must load the program into memory. The first program loaded into memory is the OS.



Allocation and Deallocation: As programs run, they will need to allocate memory and ensure that various programs do not interfere with each other's memory. When programs no longer need memory, it must be deallocated so that memory is available for other applications. Computers can run out of memory under certain conditions, and when they do, they may crash. In robotics, this process is essential because robot computers often have limited memory.

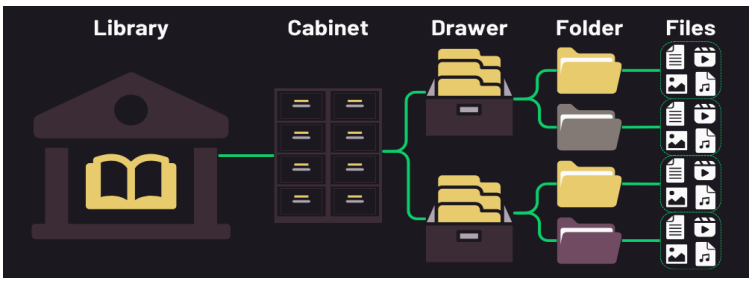


Execution: When you start a program on a computer, the OS loads the program into memory and manages the execution of resources. Once loaded, the OS determines what resources are needed, allocates them, and then sends each instruction to the ALU. The ALU decodes and executes each instruction of a program. This sounds like a lot to do to execute an instruction, but an Apple Mac Mini (M4) can execute 38 trillion operations per second.

Access to Hardware and Services: Depending upon the type of CPU, various hardware components and services are available to applications. The OS enables access to timers, interrupts, input/output ports, analog-to-digital and digital-to-analog converters, and various other hardware and services. We will introduce and discuss these in more detail throughout the course.

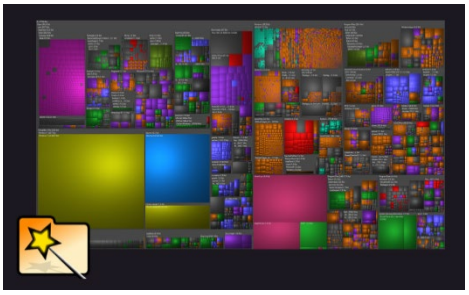
Files and Data

File System Management



A file system defines how data is organized and stored on a computer's long-term storage devices such as hard drives and USB flash drives. You can think of a file system like a filing cabinet in an office. Inside the cabinet are drawers, which contain folders, which in turn hold individual documents. Similarly, on a computer, the file system organizes data into directories (or folders) and files. Files exist inside directories, which form a hierarchical structure — like how folders and subfolders work in a physical filing cabinet. This hierarchy helps users and applications navigate and organize information in a logical way.

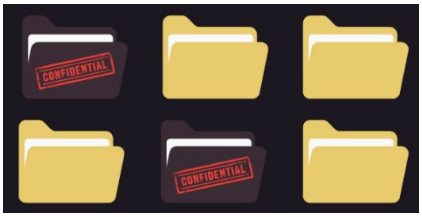
If the file system is the cabinet, then the OS acts as the office manager. It's responsible for managing access to files, when they can be read or written, and how space is allocated on the storage device. Just like an office manager coordinates office activity, the operating system coordinates access among multiple applications, preventing conflicts and data loss. It also monitors available storage space and performs background "housekeeping" tasks to keep everything running smoothly.



Internally, files are often stored in "blocks" on the storage device. The OS maintains a map — much like a table of contents — that keeps track of where every part of each file is located on the device. When a file is deleted, the OS updates the map so that space can be reused.

Beyond just organizing, the operating system controls permissions for who can access what files. Not all files are public — some are private; others can only be edited by certain users. Just like some office drawers are locked or marked "confidential," the OS uses permissions to prevent unauthorized reading, editing, or deletion of files. This is especially important in shared computers with multiple users.

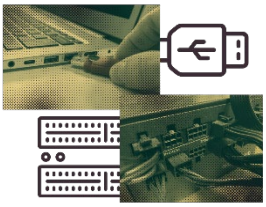
In summary, a file system allows data to be organized and retrievable, while the OS acts as a manager — coordinating access, ensuring safety, and maintaining efficiency.



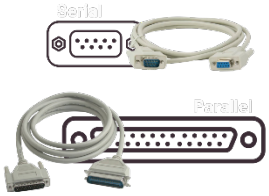
Input-Output (I/O)

Input and Output (I/O) ports enable a CPU to communicate with the outside world. In this context, "communication" is the process of receiving data (input) into the computer to be used for computation and/or storage and sending data (output) out of the computer for use by human operators or other components, devices, and machines.

- **Input Devices**: Keyboard, mouse, microphone, touchscreen
- **Output Devices**: Monitor, printer, speakers

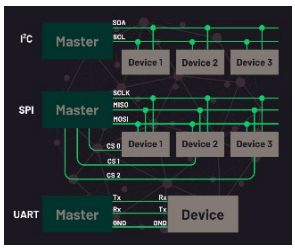


A data bus is a communication subsystem within a computer to send and receive data to and from devices. Buses follow protocol standards to ensure interoperability and enable device manufacturers to build compatible devices. Some buses provide external connectivity – the universal serial bus (USB) enables users to connect devices to the CPU. Others provide internal connection to the CPU – the Peripheral Component Interconnect (PCI) connects internal components like graphics and sound cards to the CPU.



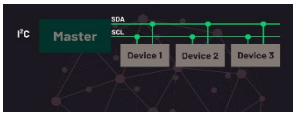
Computers send and receive serial or parallel data. The term "serial" means data is sent to and from external devices one bit at a time. "Parallel" means that multiple bits of data are sent simultaneously to and from external devices. USB is an example of serial data transfer. Today, parallel ports are obsolete for consumer devices. Parallel cables were heavy, bulky, complex, and prone to failure. Parallel ports can be found in various special purpose devices.

Device Connectivity

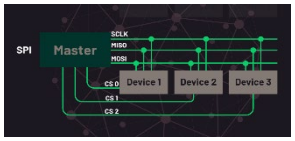


Special Purpose: In addition to commercial serial ports and protocols like USB, there are special purpose serial communication protocols to connect to other chips/devices commonly used in robotics systems.

I²C: Inter-Integrated Circuit is a protocol for short-range (~30cm), multiple-device (or chip) communication using two wires.



SPI: Serial Peripheral Interface is a protocol that provides faster data transfer over a very short range (~25cm) and is meant for high-performance applications, and uses four wires.



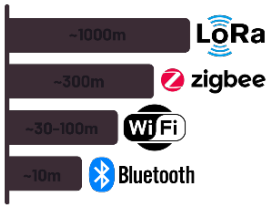
UART: Universal Asynchronous Receiver-Transmitter is used to connect devices over a long range (15m) and over 2 wire cables.

Wireless Communication

Wireless I/O enables embedded systems (like robots) to communicate with other devices without physical wires. Instead of using electrical signals over copper wires, they use radio waves, infrared, or other electromagnetic signals. This eliminates wiring, enables communication between physically separated and mobile devices, and eases integration. Examples include Bluetooth devices, smart appliances, and keyless car remotes.

Wireless Protocols

Today's Protocols: Many wireless protocols are used today, especially in the internet of things (IoT) domain.

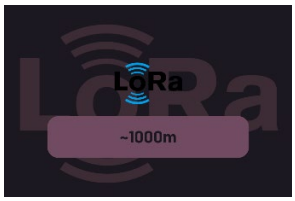


Bluetooth: This protocol can handle data transfer over only short distances (~10 m) low power for headphones, watches, keyboards smartphones.

Wi-Fi: Data transferred over Wi-Fi connections can be successful between 30–100 m, depending on environmental conditions. It's useful for connecting commercial computers devices to the internet or local area network.

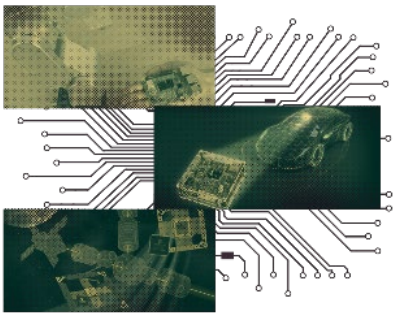
Zigbee: Perhaps a less well known protocol, Zigbee can be used to wirelessly transfer data farther distances, such as 300m. It's low power with low data rate. This type of protocol is used in peer-to-peer mesh networks in IoT applications.

LoRa: LoRa stands for Long Rang. This protocol can send data great distances (Kilometers) . The data rate is very low. This is useful for remote sensors (e.g., agriculture, weather). Power companies that have started using power meters to wirelessly send usage metrics, are getting that data over a LoRa (or similar) protocol.



Custom I/O

Unlike consumer desktops that use USB, HDMI, or Bluetooth for peripherals, embedded CPUs in robots, automobiles, spacecraft, and the like often must connect to:

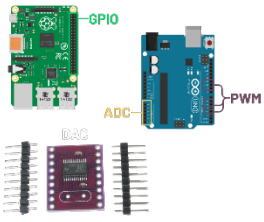


- Sensors
- Motors
- Specialized displays
- Read switches or dials
- Other special purpose, custom devices

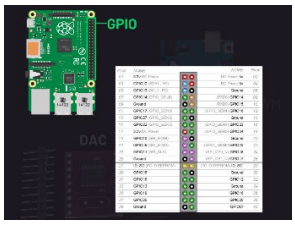
These devices require custom I/O ports and applications that use a CPU's physical electrical pins and custom-developed low-level protocols and software.

Special Purpose I/O Pins

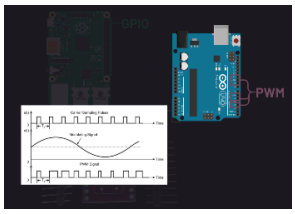
CPU Pin I/O: There are various special purpose I/O pins on CPUs commonly used for special purpose applications and devices:



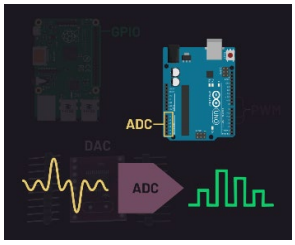
GPIO: General Purpose Input/Output (GPIO) pins are programmed as input or output to read/write one bit at a time.



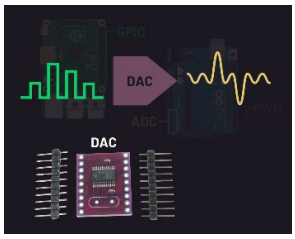
PWM: Pulse Width Modulation (PWM) is meant for output to simulate analog signals using digital pulses.



ADC: Analog to Digital (ADC) pins are designed for reading analog signals (e.g., voltage) as input.



DAC: Digital to Analog Converters (DAC) are special-purpose pins that generate analog voltages.



Linux in Robotics

Introduction to Linux

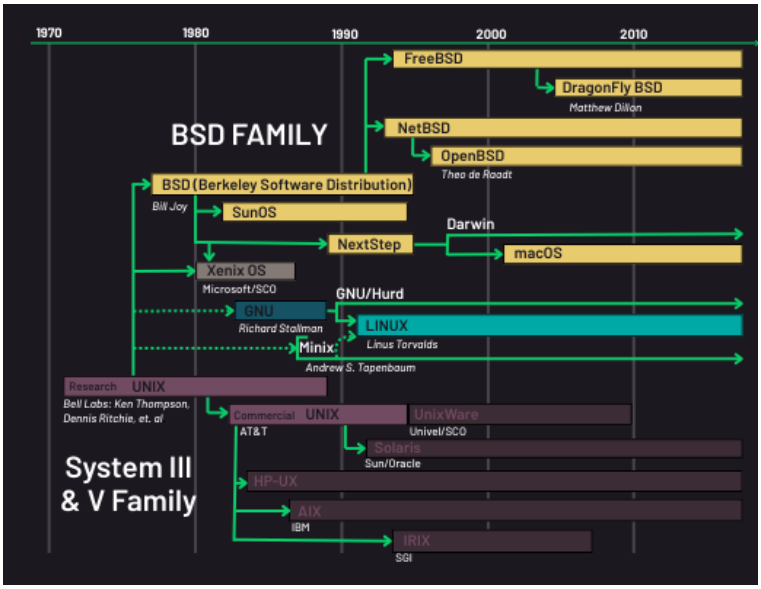
Operating Systems for Robotics

In this lecture, we will explore two types of operating systems commonly used in robotics: Linux and real-time operating systems (RTOS). Linux is widely utilized for its powerful features, flexibility, and extensive open-source community support. RTOS, on the other hand, is a special purpose operating system used for systems where precise timing and deterministic behavior are critical. Here we introduce these operating systems as examples to reinforce operating systems concepts introduced so far and their application in autonomous systems. We study these in more depth later in the course.

A Brief History

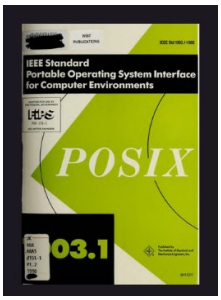
Linux was created in 1991 by Linus Torvalds as an open-source alternative to UNIX developed in 1969 by Bell Labs. Linux started as a hobby project but became popular among developers and industries worldwide. Today it powers servers, smartphones, embedded systems, robots, and supercomputers. Unlike commercial operating systems like Windows, Linux is freely available for anyone to use or modify. Linux has a long lineage and there are many variants of Linux in use today.

Linux is built around a core module called the kernel, which manages basic communication between software and hardware. The modular design of Linux allows it to be customized and scaled for different applications, whether that's running a giant server farm, or a tiny resource constrained embedded computer. This flexibility has made Linux the go-to operating system among



developers, researchers, and companies worldwide.

Distributions



The Linux kernel is compliant with the Portable Operating System Interface (POSIX) standard. This standard was established by the Institute of Electrical and Electronics Engineers (IEEE) beginning in 1988 because of the fragmentation of UNIX operating systems at that time. The POSIX standard aimed to create a common set of system calls, command line utilities, scripting language, and threading models to promote interoperability of UNIX (and later Linux) operating systems. Based on the POSIX compliant kernel, many variants of Linux have been created over the years. Variants of Linux are called "distributions." Many are designed for specific purposes. Here are a few examples:

Familiarity: Some distributions of Linux are designed for general purpose computing with easy-to-use GUIs with a familiar look-and-feel to operating systems users might already be familiar with like Windows or MacOS. These are great for users who want the power of Linux without the requirement to learn a new OS or installation hassles. Examples include:

- **Zorin:** Designed for users of Windows who want a relatively easy switch to Linux.
- **Elementary:** Mac OS is built on top of Linux but must be run on Apple hardware. Elementary OS is designed for Mac users who want a Mac OS interface on non-Apple hardware.
- **Mint:** Another Linux distribution designed for former Windows users that can mimic various versions of the Windows OS.

Minimalistic: Some distributions of Linux are designed to be very small, fast, and utilize very few resources for use in IoT, robotics, and other embedded applications on small form-factor computers with little memory and computer horsepower. These distributions typically do not have GUIs. User interaction is through a command line interface (CLI). These distributions typically require a high degree of Linux expertise to use.

- **Arch Linux:** Base installation is only 800MB.
- **Puppy Linux:** Lightweight (400MB) and runs entirely in RAM.
- **Tiny Core:** Extremely minimalistic at 20MB (you read that right). Great for use on very small, resource-limited computers.

Functionality: There are distributions of Linux that are preloaded with a suite of utilities, drivers, and other features to meet the needs of specific domains, functional environments, and applications.

- **Kali Linux:** Designed primarily for ethical hacking, digital forensics, and penetration testing in cybersecurity applications.
- **Alpine Linux:** Security focused, small (130MB), efficient Linux distribution that targets routers and other network appliances.
- **SteamOS:** Linux distribution optimized for gamers with optimized support for graphics and graphic processors.

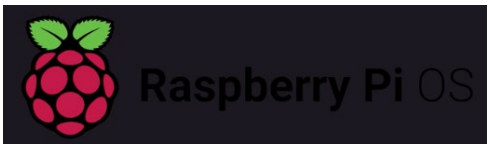
Scientific: These distributions are designed for scientific research and engineering development:

- **CAELinux:** Supports high performance computing with tools for computer aided design, computational fluid dynamics, and finite element analysis.
- **BioLinux:** Supports high-performance computing with tools for genomics and protein modeling.

General Purpose: And some distributions are designed for general purpose computing needs. Many of these are used as the basis of other operating systems or Linux variants.

- **Debian:** The Debian distribution is a general-purpose Linux distribution that can be used for desktops, laptops, enterprise servers, and high-performance computing applications. Debian is most often utilized as the basis for many other Linux distributions.
- **Ubuntu:** Most widely utilized desktop Linux distribution in use today and provides great support for general purpose computing environments. Ubuntu is built from the Debian distribution.
- **Fedora:** Another very popular Linux distribution for use in general purpose computing environments.

Raspberry Pi OS: This is the Linux distribution that we will utilize in this course and falls into several of the categories above. Because it is designed to run on the Raspberry Pi computer, it is a special purpose operating system. It is also designed to be lightweight and runs well on as little as 1GB of memory. It also comes with extensive libraries to support access (read and write) to the Raspberry Pi's pins, making it easy to interface devices (e.g., sensors, motors) for embedded applications like robotics. The Raspberry Pi OS is built from the Debian distribution. There are two primary installation options for Raspberry Pi OS depending upon your needs.



- **Raspberry Pi OS with Desktop:** This option includes a full GUI and suite of tools for everything from software development to word processing, spreadsheets, and browsers for surfing the web and general-purpose computing.
- **Raspberry Pi Lite:** This option is a lightweight installation. No GUI or tool suite. This version is often used for embedded systems like robotics, IoT, or control applications.

Raspberry Pi OS is a great operating system for diverse applications including embedded systems. You will become very familiar with setting up and using this operating system.

Command Line Interface

Flexibility of Linux

Robots are comprised of many different hardware components, many of which are custom built. The inherent flexibility of Linux makes it a natural choice for robotics applications. Engineers can tailor the Linux operating system in terms of its functionality, size (memory utilization), performance, power consumption, and to utilize custom hardware components. No other operating system offers this level of flexibility and scalability at zero cost.

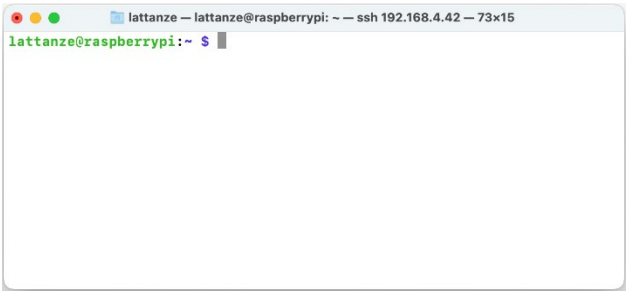
The Linux ecosystem has evolved to include an extensive, and mature set of tools, applications, and software libraries for a variety of domains – especially robotics. There are extensive tools and applications to help engineers debug, tune, and test applications and systems. Extensive open-source software libraries enable engineers to build applications without having to reinvent fundamental software functions at no cost. The developer community is huge, and help is a Google search away.

Linux Command Line Interface – First Look

It's time to bring together several concepts presented thus far. We discussed operating systems and their role in managing a computer's resources. We also discussed the Linux operating system and its widespread use in embedded and robotics domain. Now we will drill down into the Linux Command Line Interface (CLI). For our discussion here, we will use the Raspberry Pi OS distribution for our examples. This is meant as an introduction to the CLI; you will have ample opportunity to practice these commands in subsequent lab exercises.

As we discussed, you can install Raspberry Pi OS with or without a GUI interface. If you haven't installed the full desktop Raspberry Pi OS, then all you have is the CLI. If you

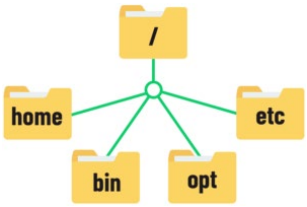
installed the full Raspberry Pi OS, then you will have to start a CLI. We will see how to do this in a subsequent lab. When the CLI is started, you will see the following on your monitor.



When you start the CLI, you are starting an application, or process called the "shell" in Linux parlance. Rather than clicking on icons as you would in a GUI based operating system, you type in commands using the keyboard. The "shell" interprets your inputs and executes the commands on your behalf. There are many different types of shells and the shell you use depends upon the Linux distribution you are using. On Raspberry Pi OS, "bash" is the default shell. An early UNIX shell was the Bourne shell, developed by Stephen Bourne in 1977 at Bell Labs. The Bourn shell was extremely advanced for its time but had limited features for the Linux OS being developed in the 1980's. The "bash shell" was developed in 1987-1989 by Brian Fox as part of an open-source project. The name "bash" stands for "Bourne Again Shell" in homage to Stephen Bourn and the religious reference to the shell's rebirth (born again).

Navigating the File System

Hierarchy



The Linux filesystem is arranged hierarchically starting with the root directory (or folder).

When you build the operating system, it establishes several directories to store files and applications used by the operating system itself. This includes root and typically those listed above. Each of these has a default purpose.

- home – where user directories are stored
- bin – application binaries
- opt – optional packages not part of standard Linux
- etc – where configuration files are stored

These are fairly standard folders in most Linux distributions and there are often others.

Print Working Directory

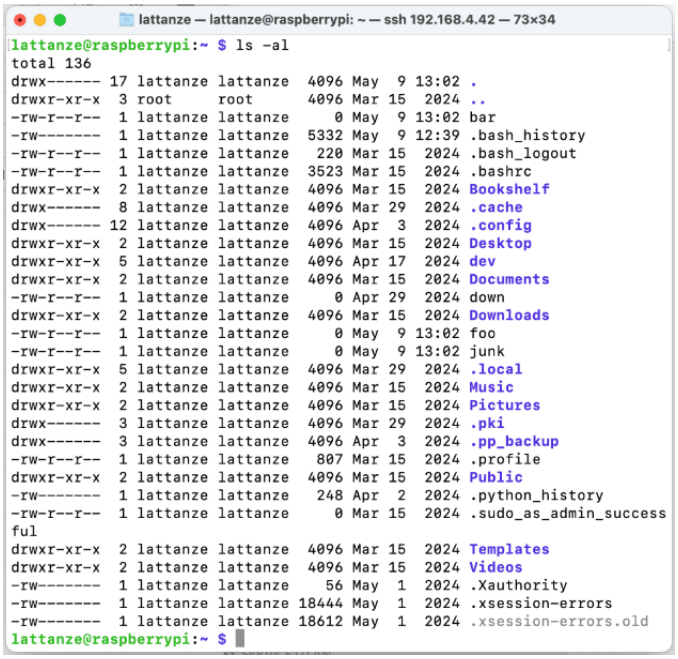
To find out where you are in the filesystem, you will use the "pwd" (print working directory) command. As you can see, the home folder for this user is: /home/lattanze. What this means is that this folder is under "root" (first /), in the "home"

directory, in the "lattanze" directory. Unless the user specifies otherwise, this is where their files are stored.

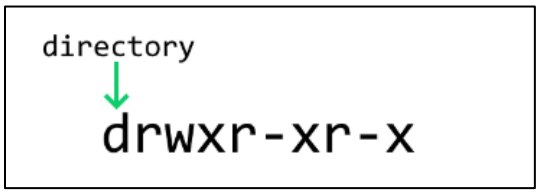
List Files

Now, let's see what files are in the user's folder using the "ls" (list files) command. There are two versions shown below.

In this example, the user typed "ls" and you see the list of items stored in this folder. Note that some of these items are files and others are folders. It can be hard to tell the difference in this view. To see details, we use the "ls -al" command as shown below. Note that some of the files start with a dot. You will not see these files if you use "ls"; you must use "ls -al" to see hidden files.

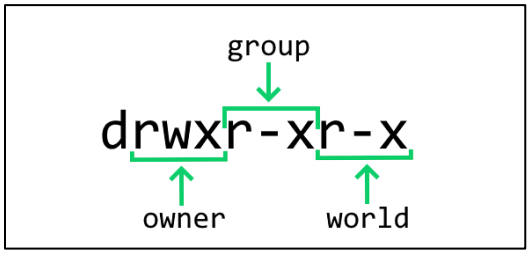


Type Identification



Here you can see all the details of each item in the user's folder including whether it's a file or directory, permissions, owner, size, creation data, and the name. The first character on the far left is either a letter or dash to indicate whether the item is a directory or a folder. If there is a "d", it's a directory. If it's a dash (-), it's a file.

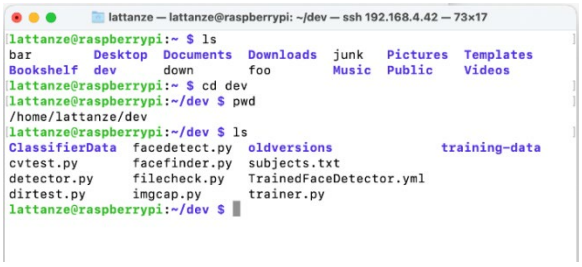
Permissions



Linux maintains permissions on files and directories that restrict or enable certain users to read, write, and execute various files. The series of letters and dashes shown on the left of one of the rows of items displays the file/directory permissions: r = read, w = write, x = execute, or a dash, which indicates that the permission for that action is not granted.

The first three, following the file/directory indicator, displays the r, w, x permissions for the file or directory owner. The next three indicate the r, w, x permissions for the group (Linux lets you assign groups, and you can assign permissions to groups). The last three indicate the r, w, and x permissions for all other users. In the picture provided, you can first see that this is a directory, and the owner has the read, write, and execute permissions. The group and world, however, only have read and execute permissions. This means the owner can make edits to the directory, but everyone else is only allowed to open it and see what's inside.

Change Directory



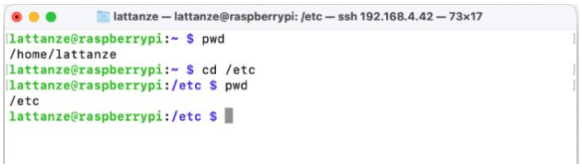
Now that we can find out what directory we are in and what's in it, let's see how to switch to another directory. To change directory, we will use the "cd" (change directory) command.

Here, the user first uses "ls" to list the files in their home directory. Then they use "cd dev" to change to the "dev" directory and use "pwd" to show their current path in the filesystem. Then they list the contents of the "dev" directory using the "ls" command.

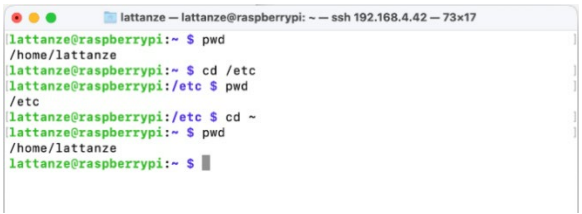
Go Back

Now let's assume that the user wants to go back up to their home directory. Rather than type the entire home path they can type "cd .." which moves their location to the next folder above the one they are in. This works because the Linux file system is hierarchical, and the "dev" directory is "below" the users home directory.

Return Home



Now assume the user wants to navigate to the system's "etc" folder. Again, the user uses the "cd" command with the full path to the "etc" folder.

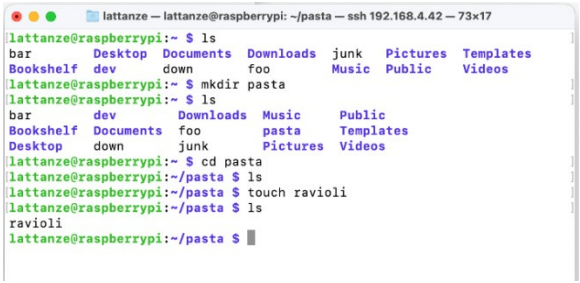


In order to get back to the home directory, the user cannot use "cd ..". Keep in mind, the Linux file system is hierarchical. The "/etc" folder is "above" the user's home directory in that hierarchy. The good news is that the user still doesn't have to provide the full path to their home directory. They can use the "cd ~" command.

No matter where you are in the filesystem, the “cd ~” command will always take you back to your home directory. Neat, eh?

Creating, Deleting & Editing in the CLI

Make Directory



To create a directory, we use the command “mkdir” (make directory). There are several ways to create files, but the first one we will use is “touch”.

Here the user lists the contents of the home directory, then creates a directory called “pasta”. Then the user changes directory to the new directory (“cd pasta”) and lists its contents (“ls”). This command doesn’t show a list of anything right now because the directory is empty. Then the user creates a file called “ravioli” in the “pasta” directory using the “touch” command (“touch ravioli”), then lists the contents of the directory again. The touch command creates an empty file, which can now be seen.

Remove File

Now let’s delete some directories and files. We will use the “rm” (remove) command to delete folders and directories. Let’s start with files.

Here the user is still in their “pasta” directory (“pwd”) and they create another file called “spaghetti” in the “pasta” folder using the “touch” command, and list the contents of the directory. Then they delete the “ravioli” file using the “rm” command and list the contents of the directory.

Remove Recursive

Let’s delete some directories using the “rm” command.

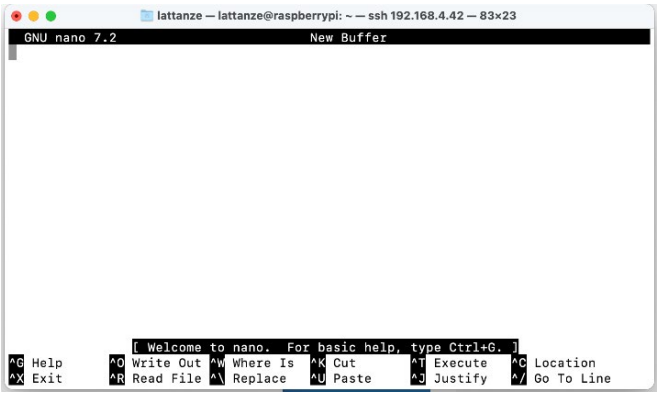
Here the user is still in their “pasta” directory (“pwd”) and creates another directory called “drinks” and lists the contents of the “pasta” directory. The user changes their mind about the new directory and decides to delete the “drinks” directory. They attempt to use the “rm” command, but the operating system says it can’t delete “drinks” because it’s a directory. Instead, the user uses the “rm -r” (remove recursive) command to remove the “drinks” directory. To remove directories, you must use the “rm -r” command.

Note that the “rm” command is very powerful and dangerous. If you have permission to delete a file, the operating system will delete it, and you can never get it back - the CLI offers no undo command. It doesn’t warn you or ask, “Are you sure?” It just deletes it. This is why you must use the “-r” option with the “rm” command to remove a directory. It serves as a warning that you are about to potentially delete a bunch of files when you attempt to delete a directory. Whenever you use the “rm” command, double check what you typed before you hit the “enter” key.

Nano

Maybe you are wondering how do you edit files from the CLI? This is especially important if you are working with a Linux distribution without a GUI. There are several editors to choose from, but we will use the “nano” editor. To start the “nano” editor, simply type “nano” at the command line or I can type “nano filename”, where “filename” is the name of a file to edit. If the file doesn’t exist, “nano” will create it.

Control Commands



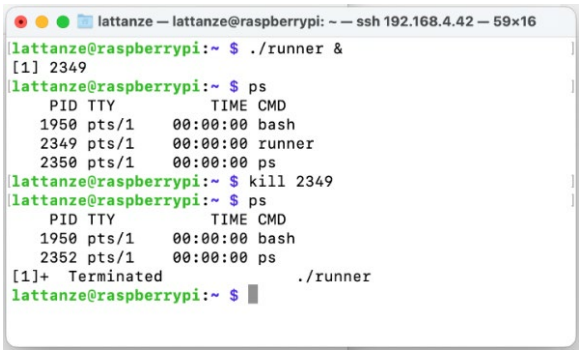
The image here shows the “nano” text editor. It’s like other editors in that you can freely type and delete text, and the cursor position is controlled with the arrow keys on your keyboard. It is not like other editors in that it does not support mouse controls. To enter control commands, you must press the control-key and a letter. These command letters are shown in the image above across the bottom of the editor. The “^” means control-key (ctrl). For example, if I want to cut a line of text I would type the control-key and the letter “k” together (ctrl+k). If I wanted to paste the line I just cut, I would type ctrl+u. To save the text in the editor to a file, I would type ctrl+x. If you get stuck, ctrl+g will give you help. There are an extensive number of control commands that are shown on the help page.

Starting and Stopping Processes

So how do you run an application on the CLI? In short, it depends on the application, but in the simplest example, you type the name of the program on the command line and hit the “enter” key.

Here the user is running the program “foo” on the command line. This program repeatedly prints “FOO!!!” on the command line. Note that

the "./" in front of the program name tells the operating system that the "foo" file is in the home folder.

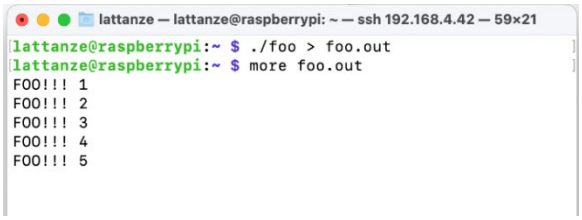


If you recall from our early discussions, operating systems manage processes and the resources they need. When a program runs, it becomes a process. To list the processes the user is running, you use the "ps" command:

Here the user started an application called "runner". The "&" means to run the program in the background, detached from the CLI. Note that the [1] 2349 is the application's process id (PID). Next the user types "ps", which lists the processes that the user is running and shows that the "runner" process is executing. It also shows "bash" which is the CLI – yes, the CLI is a process – and the "ps" command which is also a process. To stop the "runner" process, the user types "kill 2349". You can see the message that "runner" has been terminated. The "kill <PID>" stops processes from executing.

Listing File Contents and Redirecting Output

The Linux shell is a very powerful application and is an amazing interface to the powerful Linux operating system. One of the most powerful features is to redirect the output from a program into a file. Sending the output to a file so that it can be examined in detail after the program has ended is a very beneficial tool. This is great for debugging and examining logs. The ">" will redirect the output of a program into a file, creating or rewriting the content of the file.

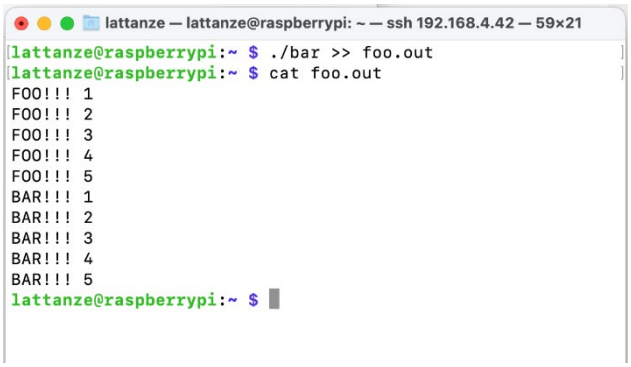


Here the user ran the program "foo" and redirected the output using ">" to a file named "foo.out". This creates the "foo.out" file. When the program ends, the user can list the contents of the file using the "more" command.

The more command displays the contents of a file one page at a time. A page, in this context, refers to the amount of content that fits within your terminal or screen window. After running the more command, you can press Enter to scroll line by line or press the spacebar to display the next full page of content. To view previously displayed content, you can typically scroll up using your terminal's scroll feature (though this depends on your terminal's capabilities).

Another way to display file contents is the cat (short for "concatenate") command. Unlike more, cat prints the entire contents of the file at once, regardless of length, which can make it harder to read if the file is very long.

The ">>" will redirect the output of a program into a file and append it to the content of an existing file, instead of rewriting the file when the command is executed. If the file doesn't exist, it will create the file.



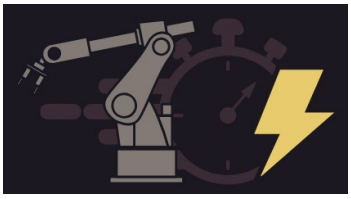
Here the user ran the "bar" program and redirected the output using ">>" to append the output to the foo.out file. The user then listed the output of "foo.out" using the "cat" command.

Summary

Believe it or not, this is an extremely brief introduction to the basic features of the Linux operating system; more features will be explored in labs throughout the course. This has also been an illustration of the key functions of an operating system in terms of managing file systems and providing access to systemic resources.

Real-Time Operating Systems

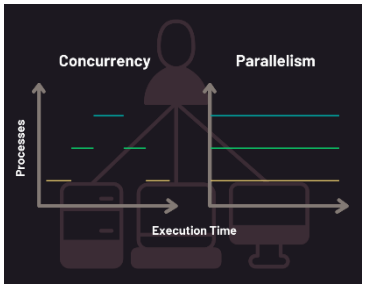
Introduction

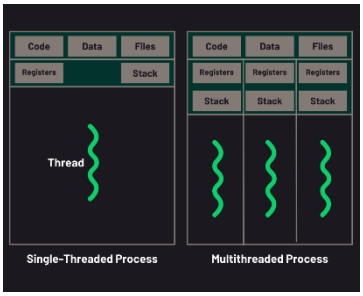


We have used the term "real-time" in our discussion so it's worth exploring its meaning. Real-time computing refers to systems that must respond to inputs in a guaranteed time. Predictability is key, not speed. For example, imagine a system that must respond to a button press within 2 milliseconds every time no matter what else the computer is doing. Real-time applications must meet specific hard deadlines every time or risk catastrophic failure.

Concurrency

Most modern operating systems are able to execute multiple applications concurrently (i.e., multiprocessing). Even though it looks like a computer is doing a bunch of tasks at the same time (e.g., email, playing music, spreadsheets, browsing), it's actually switching between tasks very quickly, working on each one a little bit at a time. The operating system is responsible for coordinating this concurrent operation.

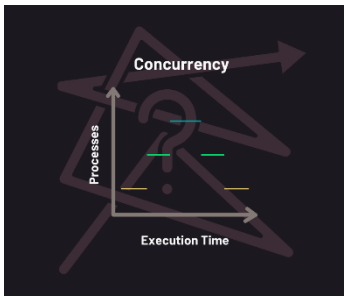


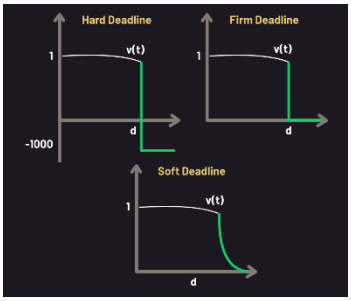


Because robots do many things at the same time, robot applications need to do many things concurrently like moving, sensing, and making decisions. Threads enable programmers to design applications to perform many operations at the same time to increase performance and efficiency. While the OS makes threading possible, the programmer is in control of what tasks run, when, and for how long. We will study this in more detail later in the course.

Scheduling & Deadlines

Standard Linux utilizes "fairness based" scheduling where each task gets equal access to the CPU for a period of time resulting in balanced performance across applications (see Concurrency section). While this provides a good user experience, task execution time is unpredictable. Depending on the number of tasks, it can take a longer or shorter period of time to complete a task. This strategy does not work when tasks must meet hard deadlines every time.



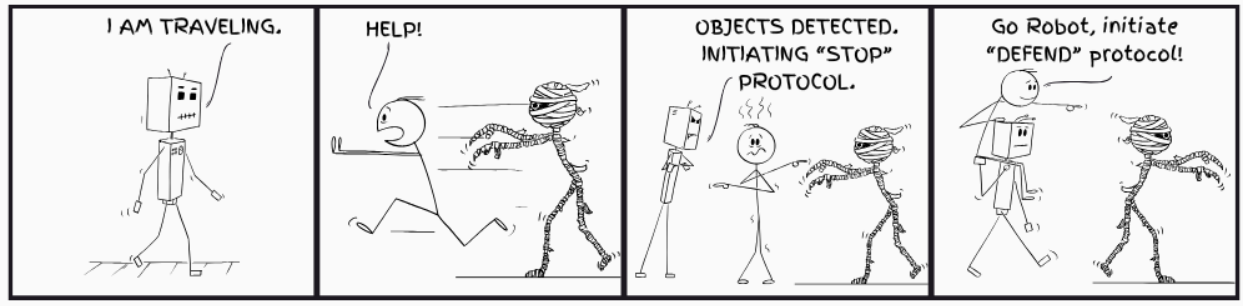


There are two main types of deadlines: hard deadlines, where missing a deadline could cause failure or danger, and soft real-time, where statistically bound deadlines are acceptable (e.g., response guaranteed in 5-10 microseconds). Understanding the system requirements for performance is key to selecting the right operating system for the robot's applications.

RTOS

Imagine a robot traveling across a warehouse when a person steps in its path. Once the sensor detects the obstruction (a human), it sends an event to the CPU. The CPU's software immediately responds with an action to stop the robot. This task cannot be interrupted; fairness is not an option. All CPU resources are assigned and prioritized to ensure this task starts and finishes, regardless of the status of other tasks. The time from start to finish must be predictable every time. This is an example of real-time scheduling.

A real-time operating system (RTOS) is designed to let developers define scheduling strategies and task priorities to meet strict deadlines. RTOSes are commonly used in robotics systems for various application. Robots may have multiple onboard computers, and some may use a combination of Linux and RTOS operating systems depending on the specific functions and responsibilities of each component.

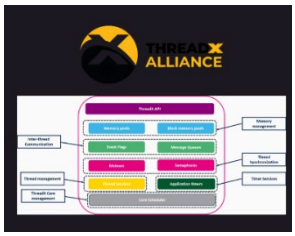


RTOS Options

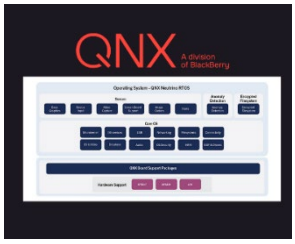
There are several RTOS options depending upon the application.

VxWorks: Paid. Commonly utilized in aerospace and military applications.

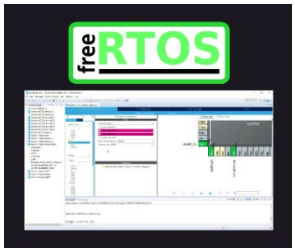
QNX: Paid. Widely used in commercial applications such as automotive, medical, and industrial controls



ThreadX: Paid. Designed for use in embedded applications (robotics).

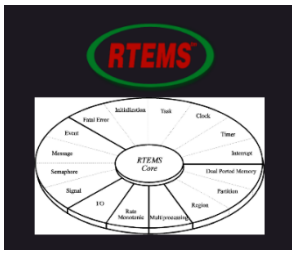


FreeRTOS: Open source. One of the most widely used RTOS (automotive, robotics, IoT, controllers).



RTEMS: Open source. Used in space systems and industrial control systems.

- We will experiment with FreeRTOS on a microcontroller in the lab at the end of this module.



Real-Time Linux

Linux is a fairness-based operating system and is not suitable for systems with hard, fixed deadlines. However, there are real-time versions of Linux. One example is PREEMPT_RT. PREEMPT_RT is built from a set of patches that can be applied to a standard Linux kernel transforming it into an RTOS. As the name implies, PREEMPT_RT enables programmers to write tasks that preempt other tasks running on the system to meet hard deadlines.

Module 1 Lab: Compare General Purpose OS & RTOS

Part 1: Setting Up the Raspberry Pi

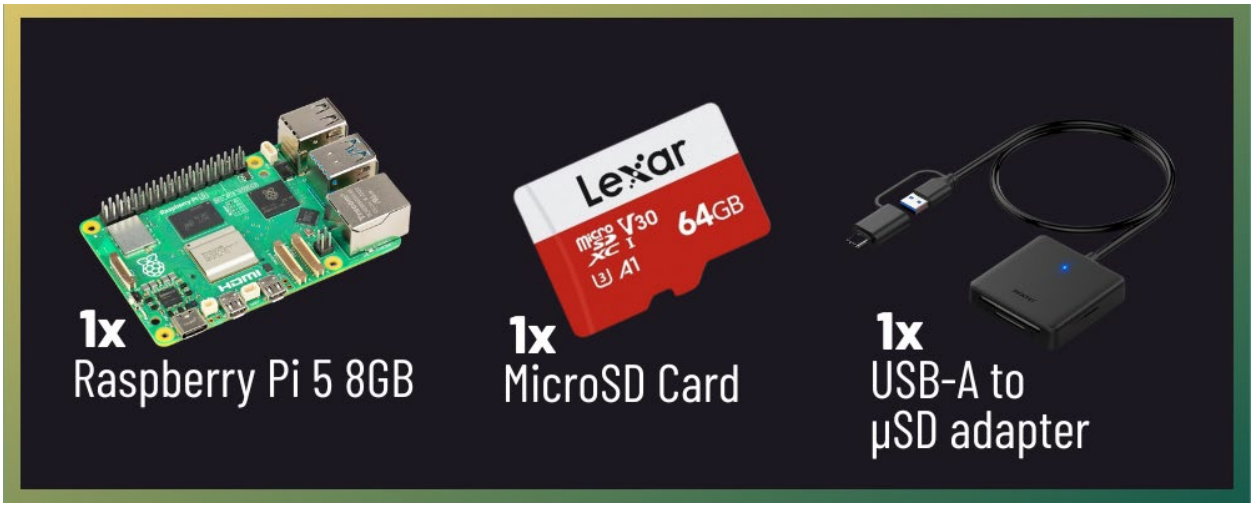
The Raspberry Pi Operating System Image

Raspberry Pi computers store their operating system (OS) on a removable MicroSD card. In your IRAP kit there is a laptop that is meant for a Raspberry Pi, but the laptop didn't come with the computer itself. This basically makes the entire laptop just a compiled machine of input and output peripherals to use with a computer (monitor, keyboard, various gadgets and circuitry).

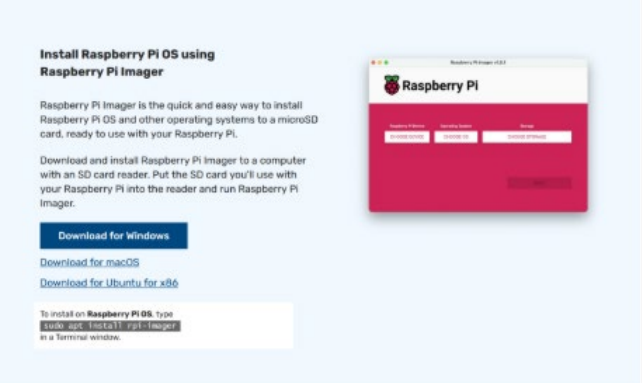
Before you can start using the CrowPi, you will first need to install an "image" of the Raspberry Pi OS onto the MicroSD card that is in your IRAP kit. This is not the same MicroSD card that came with the CrowPi laptop kit, which already has an OS on it. The MicroSD card that came individually packaged inside the IRAP kit (i.e., not inside any other included kit), is the one that you will write the image of the OS to.

Task: Image a MicroSD Card

Materials



Install the Raspberry Pi Imager



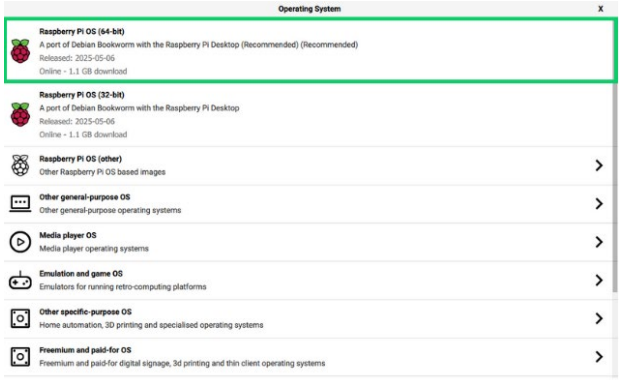
First, you need to connect the individual MicroSD card from the IRAP kit to your personal computer using either the ports on your computer, or the adapter in your kit. Next, you need to install the imaging software for Raspberry Pi. This program can be installed on Windows, Mac, or Linux. The instructions provided show how to install it on a windows machine. Click here to get to the web page for the imager software and follow the steps to download the imager onto your personal computer.

[Optional] Select your Raspberry Pi Device

When you open the imager, you will have three options available, selecting a Raspberry Pi Device, selecting Operating System, and selecting what Storage you are writing to. There are numerous available OS, but not all of them work across all versions of the Raspberry Pi, you can provide a filter to ensure you are downloading the correct one for your system. In our case, we would select the Raspberry Pi 5, as that is the device that we are mounting the MicroSD to.

Selecting your Operating System

Next, we will select the OS. Listed within the imager are numerous options for an OS. Each one provides a small blurb about what their purpose is, but for our purposes, we will be using the most recent version of the Raspberry Pi OS (64-Bit) located at the top of the list. Select it and continue with the next steps.



Select a Storage Device

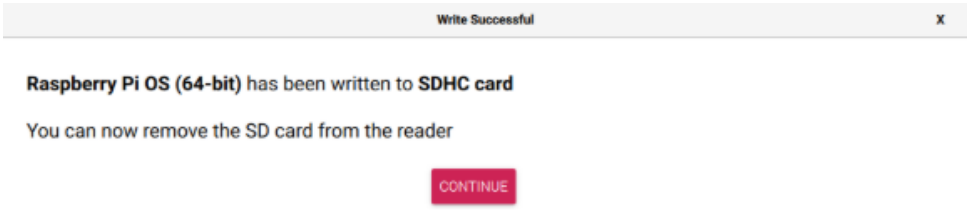
The last step before imaging the OS onto the MicroSD card, is to select the MicroSD card. Within the Storage selection, you will see ALL connected storage devices, this includes flash drives, removeable hard drives, or other MicroSD cards. Ensure you select the correct MicroSD card — flashing will DESTROY ALL information stored on the target device.

[Optional] Setting Customization Settings

When flashing your MicroSD card, you will use the default settings of the Raspberry Pi. Some of these settings can be set during the initial boot-up, but others are best to be set up prior to boot. We will not be investigating these at this time, but we implore you to explore what is available to be set. Otherwise, continue forward to the next step.

Flashing and Completing

We have no OS customization settings, we will select “No” at this time. The imager will ask one more time if you would like to continue, and to ensure you have selected the correct device as all information on the selected device will be erased. Select "YES", and once the imager finishes writing and is verified, the software will inform you that you can eject the MicroSD card and insert it into your Raspberry Pi.

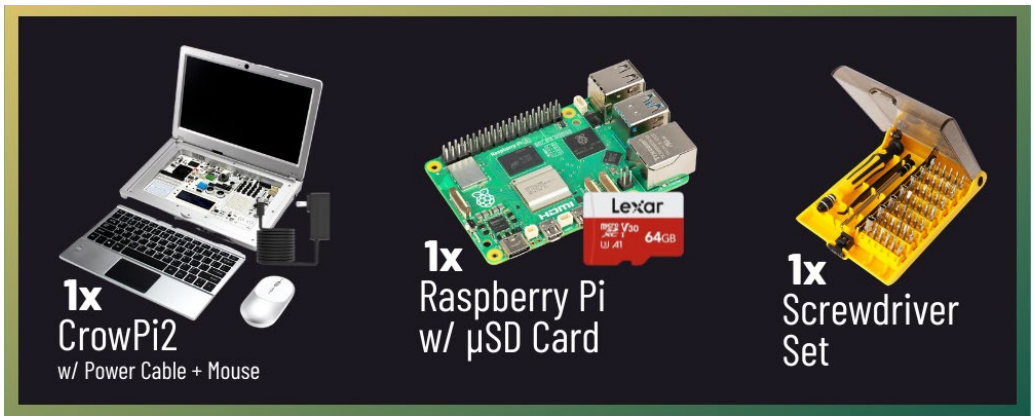


The All-in-One Device

The CrowPi is an all-in-one laptop connecting the Raspberry Pi with a monitor, keyboard and various sensors and electronics that you can program the Raspberry Pi to interact with. It also includes GPIO pins so you can tinker with external components as well. [Click here] to check out the user manual to see all the capabilities.

Task: Assemble the CrowPi and Completing Set-Up

Materials



Open the CrowPi Back Cover

On the back of the CrowPi, locate the back cover protecting the Raspberry Pi compartment. It will slide out towards the right to open the compartment. This is where your Raspberry Pi will sit.

Connect the Raspberry Pi

With the MicroSD card inserted into the Raspberry Pi, turn the Raspberry Pi face down and align the GPIO header with the female GPIO header in the compartment. Push slowly down until the Raspberry Pi fits completely.

You'll also want to screw in at least 1 or 2 corners of the Raspberry Pi to keep it secure, but don't make it too tight. You'll have situations when you need to remove this raspberry and replace it with another in one of the later labs.

Connect the Interface Board

The white interface board supplies power to the Raspberry Pi and connects it to the monitor. When joining the interface board and the Raspberry Pi, you'll notice that the interface dongles are longer than the Raspberry Pi ports. Even though the dongles may be fully seated into the Raspberry Pi ports, a small portion of each of the dongles will be left exposed. This is expected.

Connect Camera Cable and Close the Back Cover

Extend the USB cable out of the compartment and close the back cover carefully, making sure the USB cable is laying safely across the wire hole. Plug the USB cable into one of the ports on the CrowPi.

Attach the Power Cable

The CrowPi does not have a battery like other laptops. There is power cable included with the CrowPi to draw power from a nearby power outlet. Once plugged in to power, push the power button to begin working with the Raspberry Pi on the CrowPi.

Connect the Keyboard and Mouse

Open the battery compartment on the mouse and pull out the USB dongle to connect to the Raspberry Pi. If the mouse does not respond, change the USB port of the dongle. Next, insert the battery into the mouse and toggle the switch to the ON position.

Start the Raspberry Pi

When a Raspberry Pi powers-on for the first time, it launches the initial set-up process. Follow the on-screen settings to complete the start up. The username and password for your Raspberry Pi can be whatever you prefer and can remember.

Scheduling & prioritization of tasks

A Real-Time Operating System (RTOS) differs from Linux in terms of task management and scheduling priorities. RTOS is designed for deterministic, time-critical applications where tasks must be completed within strict deadlines. It typically uses preemptive, priority-based scheduling to ensure high-priority tasks are executed with minimal latency, often interrupting lower-priority ones.

In contrast, Linux is a general-purpose operating system optimized for resource utilization rather than strict timing. Linux tries to keep everything running smoothly for all programs, but it doesn't guarantee that any one task will happen exactly on time. This can lead to delays that are unacceptable in real-time systems.

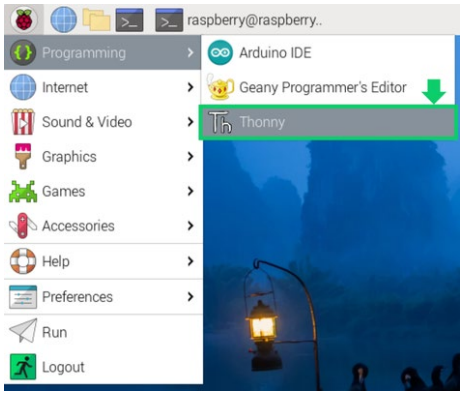
In this next task you will get the opportunity to see the differences between RTOS and Linux in how they handle processing, using an Arduino, Raspberry Pi, and some python code.

Task: Raspberry Pi Sequential Monitoring

Materials

Running Code on Thonny

The Raspberry Pi has multiple editing software for writing programs already pre-installed. Two such applications are Geany and Thonny. They are both IDEs with minor differences in features. However, Thonny was developed to support one programming language: Python, whereas Geany can also support others. In this lab we use the Thonny editor. Click on the Raspberry Pi symbol in the top left corner. Select programming, then select Thonny.



Python Code Order

When writing a program, it is good practice to keep your code organized. An example of good form is to have all your import and include statements at the top, followed by all your individual functions and methods, and ending the program with the main body of the program. Also, good practice is including sufficient comments (non-executed statements) throughout your program. In Python, any text preceded with a pound sign, turns anything to the right of that symbol into a comment.

```
# Import Libraries
{ … }
# Setting up Individual Tasks as Functions
{ … }
# Main Code Loop
```

```
{ … }
```

Libraries

Python has numerous useful libraries that make coding easier. Think of it as code someone else has made that you may leverage in your own scripts. Many libraries come pre-installed with Python. In this code we use the "time" library. This library is utilized for all things related to time, such as retrieving system time, setting delays, or measuring duration. Include the time library in your program by writing "import time" just beneath the title of the first section "import libraries."

```
# Import Libraries
import time
```

Methods

We use definitions (i.e., methods or functions) to separate any code that is repeated from the main body of the program. In this code, we'll make three definitions: one for sensing, one for movement, and one for safety.

All our methods follow a similar pattern: each takes a number as a parameter, displays a print statement to the console, and then delays the program by the number of seconds that was taken as the parameter. Review the following code and then type (or copy and paste) it into your IDE.

```
def Sensing(delay_time):
    print("Performing Sensor Tasks")
    time.sleep(delay_time)

def Movement(delay_time):
    print("Performing Movement Tasks")
    time.sleep(delay_time)

def Safety(delay_time):
    print ("Performing Safety Tasks")
    time.sleep(delay_time)
```

Main Body

To start, the first line "if __name__ == "__main__": indicates that the main function will be executed within this script and protects it from being called by another script. Notice that

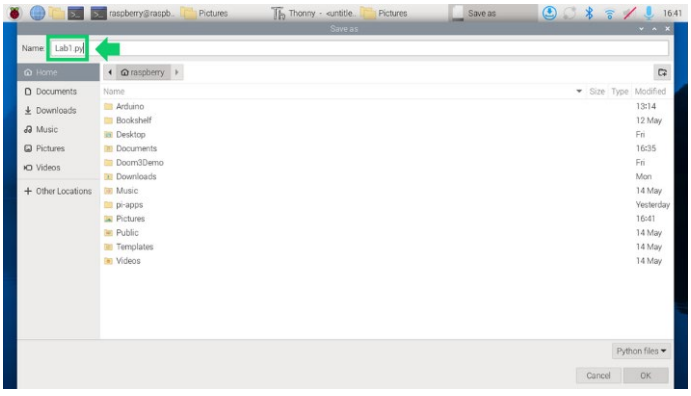
there are two underscores on each side of the words, “name” and “main.” Next, the two variables are initiated: “delay_time” and the letter “i” for index. The index will be used as a counter.

Typically, somewhere in the main body of a program, there is a loop that keeps the program running under certain conditions. In this program, the code will just repeat without end until the program is terminated. During the loop, all three methods are called for execution, then a print statement is sent to the console with the index value, which is then incremented, and then the loop starts again.

```
if __name__ == "__main__":
  delay_time = 1  # Change this value to see how increasing the robot tasking delays the counter from incrementing.
  i = 0  # initiate the counter
  print('starting')
  # Main loop that will run the tasks.
  while True:
    Sensing(delay_time)
    Movement(delay_time)
    Safety(delay_time)

    print(i)
    i = i + 1  # increment the counter
    print("")  # for readability
```

Save and Execute



When you try to run your code for the first time, you will be required to save the file prior to the run, if you haven’t already done so. Navigate to your Documents directory. In here, you may want to consider making a special directory for small programming tasks, such as this. Type the name of the program to any name, but be sure to use “.py” as the file extension. For example, you can name this file “simpleSim.py” and save it to a directory named “My Programs.”

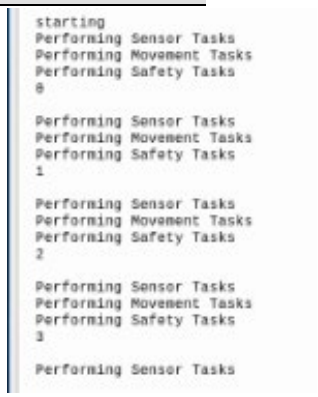
A Comment on Sequence

Python code is executed sequentially, in order from first statement at the top of the code,

then moving down. In real-life situations where safety conditions are being monitored and will need to react immediately when a safety violation is sensed, waiting until all other tasks are completed might be detrimental. One way to address this is to reorder the tasks so that the safety is checked before the other code is executed. See the while loop portion where the methods are called:

```
i   # Main loop that will run the tasks.
    while True:
        Safety(delay_time)
        Movement(delay_time)
        Sensing(delay_time)
```

While this new order of statements checks for safety first, it doesn't resolve the issue that if a safety violation has occurred before the other methods have finished executing, there can still be detrimental consequences. That's where special programming tools, such as FreeRTOS come into play.



FreeRTOS puts priority on the scheduling for such events as real time immediate responses. In the next part of this lab, we will see how FreeRTOS is able to assist the program to always put safety first.

Part 2: Using Arduino Uno

Programming with Arduino

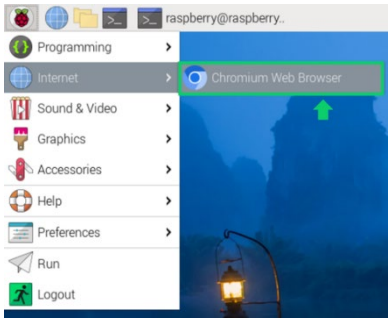
Through this lab we will be using the Arduino IDE. Before we can begin programming, we will need to install the IDE. Follow the next steps to understand how to install the IDE properly.

Task: Download and Install Arduino IDE

Materials

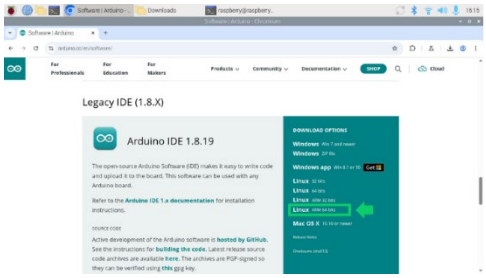
Open Chromium Web browser

First, turn on the CrowPi and ensure the mouse is connected. Open the Chromium Web Browser by pressing the menu key on the keyboard or navigating to the Raspberry Pi icon on the top left of the screen and then Internet> Chromium Web Browser.



Download the Arduino IDE

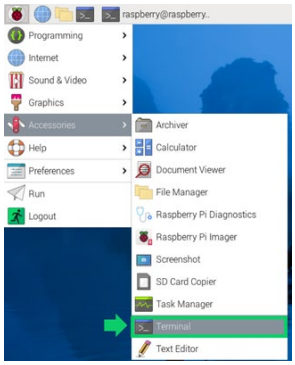
Download the Arduino IDE by going to [this website] and follow the download steps for Linux. Scroll down to the Legacy IDE section and click on the download labeled "Linux ARM 64 bits."



This should download a file that is named "arduino-####-linuxaarch64.tar.xz" where #### is the version number, in this case – arduino-1.8.19-linuxaarch.tar.xz.

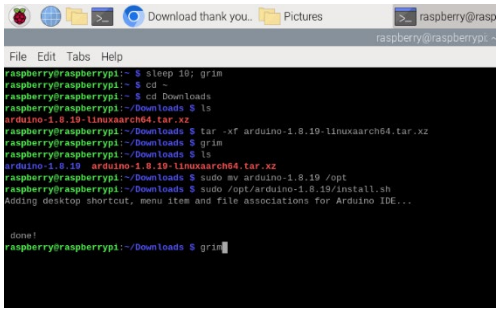
Open the Linux Terminal

Open a terminal window by clicking the Terminal icon at the top of the screen, or by selecting Accessories and then Terminal in the menu.



Home Directory

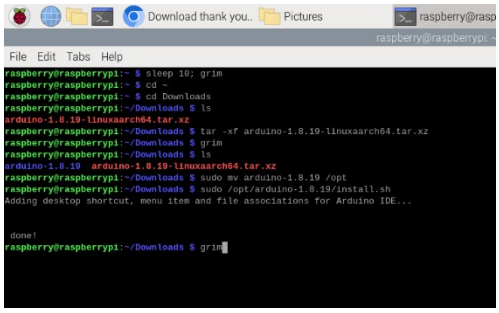
Ensure you start in the home directory, then navigate into the Downloads directory and list the files in that directory. The Arduino IDE will be listed as: ""arduino-####-linuxaarch.tar.xz" where #### is the version number.



cd ~
cd Downloads
ls

Extract using tar

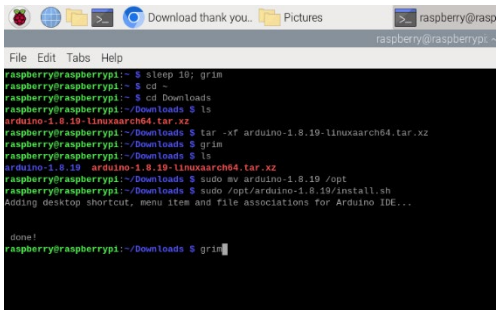
Extract the contents of the downloaded file using the tar command, and the -xf flags. X is for extracting a tar ball. F is for specifying an archive or a tarball filename. Replace the pound signs with the appropriate values. This should create a directory named "arduino-####".



tar -xf arduino-####-linuxaarch.tar.xz

Move and Install

Move the folder to /opt, then complete the installation of the Arduino application.



sudo mv arduino-#### /opt
sudo /opt/arduino-####/install.sh

Arduino Libraries

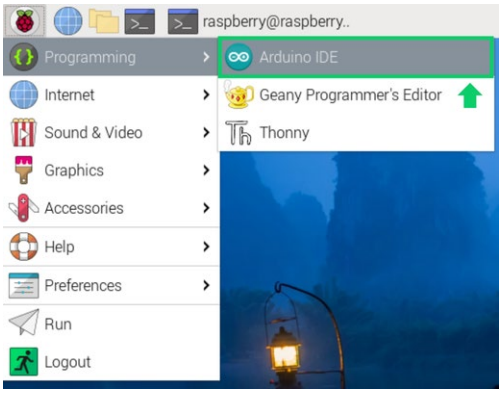
The Arduino IDE comes preinstalled with software libraries for Arduino AVR boards; however, we need to manually install the library for our Arduino Uno R4 Wi-Fi board before we flash any code on it.

Task: Install Uno R4 Wi-Fi Board Library

Materials

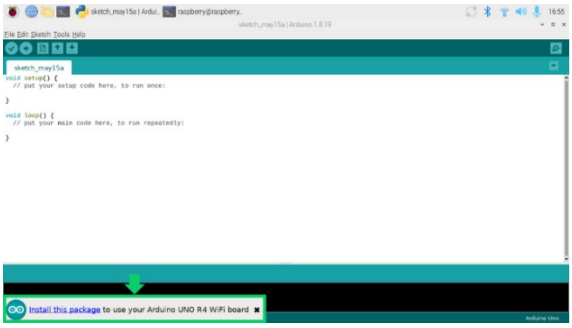
Open the Arduino IDE

The installation process adds Arduino to the Raspberry Pi Programming options in the OS menu. Plug in the Arduino to one of the USB A ports of the Crowpi, then click "Arduino IDE" to run the IDE.



Install the Wi-Fi Board

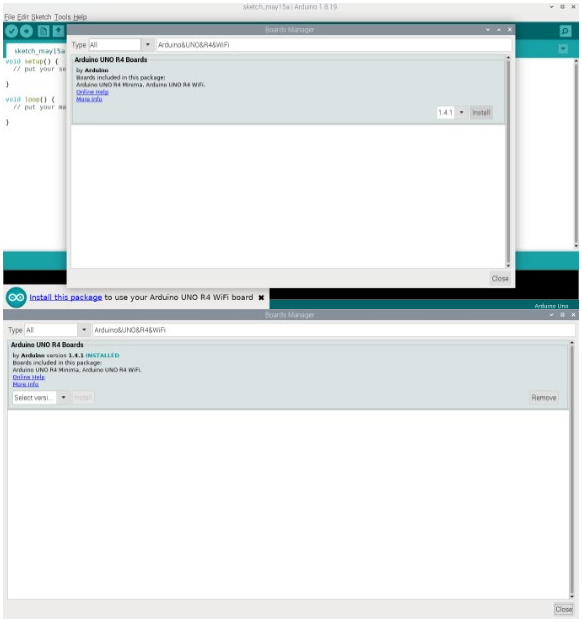
When opening the Arduino IDE for the first time, you will be prompted to "Install this package to use your Arduino UNO R4 WiFi Board" on the bottom left. Click on "Install this Package." If you don't see or missed this notice, open the Board Manager (Tools > Board > Board Manager). Search "Arduino&UNO&R4&WiFI" on the list to find it. This might take a few moments. Please be patient, Arduino is a large IDE



Select and Install

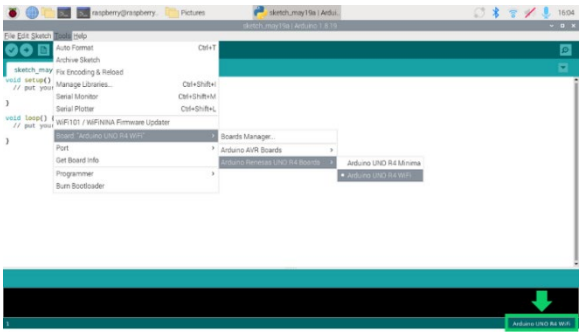
When the Board Manager opens, locate the Arduino Uno R4 board library. Select the latest version for installation.

Click on install and wait for the IDE to install the new board library. Installing may take several moments. Please be patient. Once it has finished, an Installed tag should appear next to the Arduino UNO R4 library. You can close the boards manager.



Select the Board

Finally, select the installed board library by clicking on the "Tools" menu and then Board: > Arduino Renesas UNO R4 Boards > Arduino UNO R4 WiFi. Once selected, the board information “Arduino UNO R4 WiFi” should appear on the bottom right corner of the IDE.

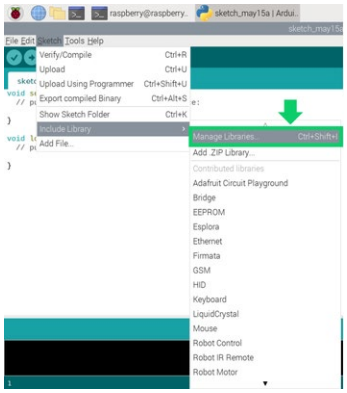


Task: Install FreeRTOS Library

Materials

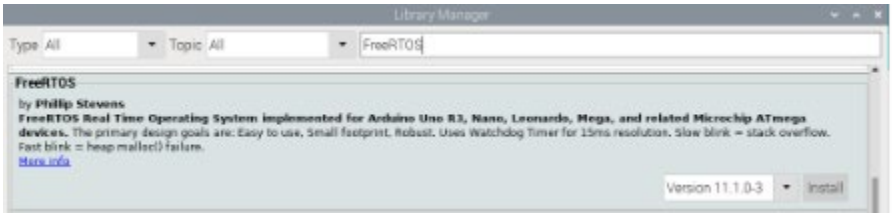
Manage Libraries

We install Arduino libraries using the in-built library manager. Open the IDE and click on the "Sketch" menu and then Include Library > Manage Libraries.



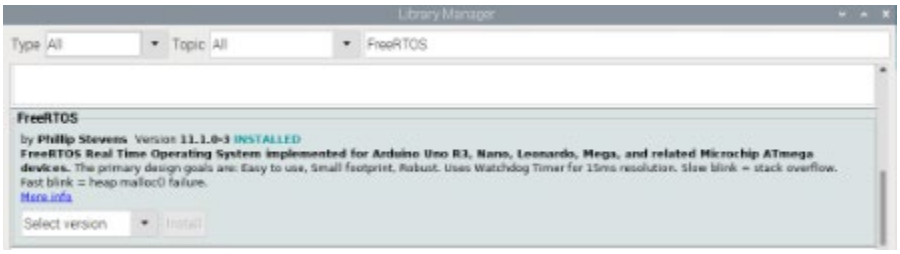
Installing FreeRTOS Using the Library Manager

The Library Manager will open, and you will find a list of libraries that are already installed or ready for installation. Search “FreeRTOS” on the list to find it, click on it, then select the version of the library you want to install.



Finalizing the FreeRTOS Library Installation

Finally click on install and wait for the IDE to install the new library. Once it has finished, an Installed tag should appear next to the FreeRTOS library. You can close the library manager.



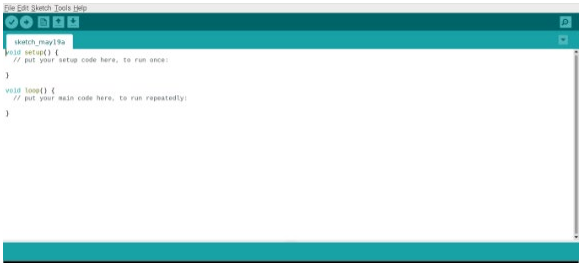
Next Steps

Now that the Arduino IDE software and libraries are installed, we can start coding. We will program the Arduino to demonstrate real-time task scheduling using FreeRTOS. FreeRTOS manages multiple tasks with different priorities. It allows tasks to run seemingly in parallel, with the scheduler determining which task runs based on priority. The following code creates two concurrent tasks: one blinks the onboard LED at regular intervals, and the other prints a counter to the Serial Monitor, showing the effectiveness of task scheduling.

Task: Programming the Arduino

Open Arduino IDE

When you open the Arduino IDE, it will create a file/sketch by default with the date created as the name. The code is empty except for two functions – void setup() and void loop() which will be explained later.



Importing Libraries

This program uses the FreeRTOS library to enable multitasking on Arduino. Unlike regular Arduino code that executes a single task at a time, this lets us run multiple tasks simultaneously with different priorities.

```
include <Arduino_FreeRTOS.h>
```

Setting Up Tasks

Before diving into the main program, we define the behavior and scheduling priorities of two tasks: one responsible for blinking an LED and another for printing messages to the serial monitor. By default, the LED task is given a higher priority to ensure it runs more urgently than the print task. We create variables “LEDPriority” and “PrintPriority” to assign their respected priorities. LEDPriority = 2 for the LED blink task and PrintPriority = 1 for the print task. While both tasks run continuously, FreeRTOS uses these priority levels to decide which task gets CPU time when both are ready to execute. By adjusting these values, we can observe how task preemption and scheduling are influenced by priority settings.

```
// Task function prototypes
void TaskBlinkLED(void *pvParameters);
void TaskPrintMessage(void *pvParameters);

// Priorities - you can tinker with these to see the effects to scheduling
uint32_t LEDPriority = 2;      //Higher Priority
uint32_t PrintPriority = 1;    // Lower Priority
```

Arduino Core Functions

There are two core functions in every Arduino program: “void setup()” - Runs once at startup to initialize hardware and “void loop()” - Runs repeatedly forever and contains main program logic. In our FreeRTOS program, loop() is left empty because tasks are managed by the RTOS scheduler, while setup() is still used to initialize hardware and create tasks. Once all tasks are created using xTaskCreate(), the vTaskStartScheduler() function is called (These functions are defined within the FreeRTOS library). This will transfer control to the FreeRTOS kernel, which begins scheduling and executing tasks according to their priority and timing. From this point on, the loop() function is never called, and all program logic is handled within the task functions.

```
void setup() {
```

```
  // Initialize serial communication
  Serial.begin(9600);
  while (!Serial);

  // Create tasks
  xTaskCreate(
    TaskBlinkLED,     // Task function
    "Blink",          // Task name
    128,              // Stack size
    NULL,             // Parameters
    LEDPriority,      // Priority
    NULL              // Task handle
  );

  xTaskCreate(
    TaskPrintMessage,
    "Print",
    128,
    NULL,
    PrintPriority,
    NULL
  );

  // Start the scheduler
  // After this, setup() and loop() won't be called anymore
  vTaskStartScheduler();
}

void loop() {
  // This will never run when using FreeRTOS
}
```

Functions

The two task functions in this program—TaskBlinkLED and TaskPrintMessage—each run in an infinite loop and carry out their respective duties with specific timing delays. These loops are sustained forever to ensure each task operates continuously throughout the runtime of the application.

The TaskBlinkLED function is responsible for toggling the onboard LED on and off every 5000 milliseconds. It does this by setting the LED pin high, waiting 5000 milliseconds using vTaskDelay(), setting it low, and then waiting another 5000 milliseconds before repeating.

Since the LED blink task has a higher priority, it will preempt the print task whenever it is ready to run. The TaskPrintMessage function maintains a counter that increments each time the task runs and prints the value to the Serial Monitor every 500 milliseconds. Because this task has a lower priority, it will only run when the higher-priority LED task is idle or waiting.

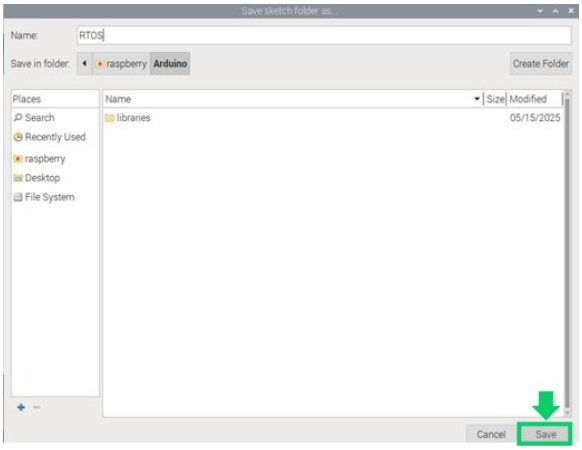
```
// Task 1: Blink an LED (if available)
void TaskBlinkLED(void *pvParameters){
 pinMode(LED_BUILTIN, OUTPUT);
 while(1){
  digitalWrite(LED_BUILTIN, HIGH);
  vTaskDelay(5000 / portTICK_PERIOD_MS); // Delay for 5000ms
  digitalWrite(LED_BUILTIN, LOW);
  vTaskDelay(5000 / portTICK_PERIOD_MS);
 }
}

// Task 2: Print a message
void TaskPrintMessage(void *pvParameters){
 int x = 0;
 while(1){
  x = x + 1;
  Serial.println(x);
  vTaskDelay(500 / portTICK_PERIOD_MS); // Delay for 500ms
 }
}
```

Save the File

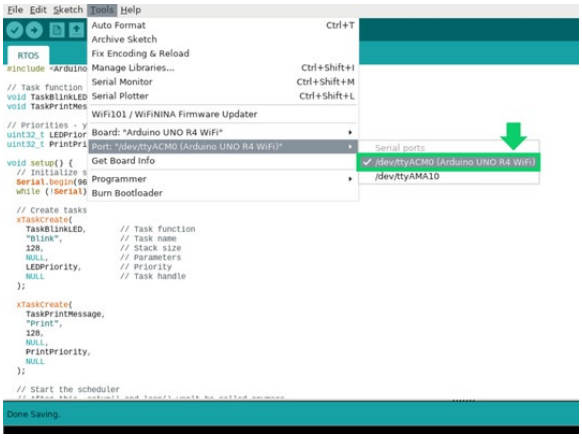
Now that the code is completed, we can save the file by clicking on the "Save" icon located below the “Tools” bar or by pressing Ctrl+S.

Once we click save, a “Save sketch folder as...” window will open. For this Lab, we will name the file RTOS. Click on the Save button on the bottom left of the window to complete saving. After saving a “Done Saving.” notification should appear at the bottom of the IDE.



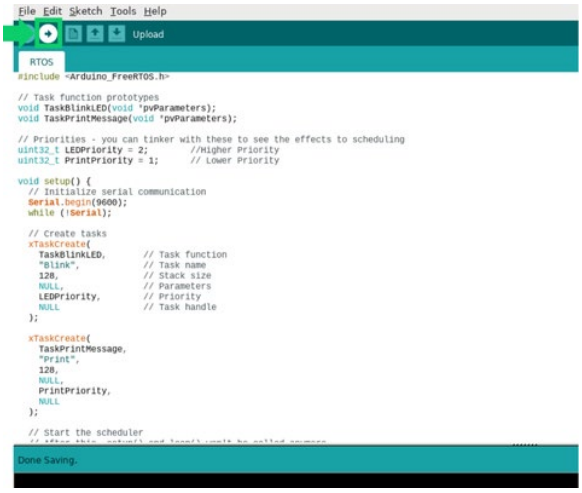
Select Serial Port

Plug in the Arduino to the USB A port of the Crowpi. We need to select the serial port (USB port the Arduino is plugged into) in the Arduino IDE before we can upload the code to the board. To do so, click on the "Tools" menu and then Port: > /dev/ttyACM0 (Arduino UNO R4 WiFi). The Port Number will vary based on where you plug the Arduino, in this case it is listed as /dev/ttyACM0



Upload the Code

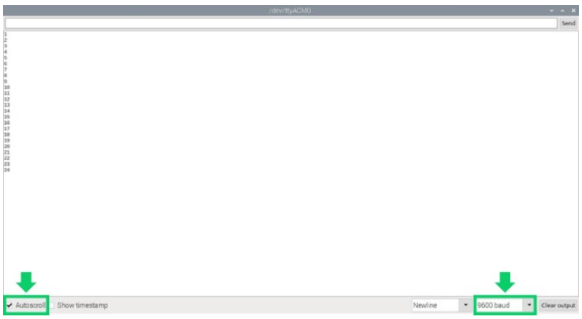
To flash the code onto the Arduino, we must select the Upload option in the toolbar below the File, Edit, Sketch, Tools and Help Menus. After uploading, the code remains stored in the Arduino's flash memory permanently, even when power is disconnected.



View Output in Serial Monitor

We can open the serial monitor by navigating to the serial monitor icon on the top left of the IDE or by clicking on "Tools" menu and then Serial Monitor.

With the Serial monitor opened, we can view the output of the program. It will display the output of the counter – starting from 1 and increasing every second. The onboard LED will toggle on and off while this task is running. Try adjusting the task priorities and delays in the code to observe how they affect the timing and behavior of the output. Ensure that the baud rate is set to 9600 baud and the Autoscroll option is toggled on.



Well Done!

This completes Module 1: Overview of Robotics and Operating Systems.

Way to stick with it! The purpose of this module was to provide you with a foundational understanding of both robotics and the operating systems that support them. You've explored how robots perceive, process, and act, learned about levels of autonomy and the role of human interaction, and examined the structure and purpose of operating systems—especially Linux and real-time operating systems—in robotic applications.

By now, you should be able to:

1. Understand robotics technologies conversationally, including key components and how they function together.
2. Comprehend the role of operating systems (OS) in robotics, particularly real-time OS (RTOS).
3. Develop trust in an OS's ability to manage robotic tasks efficiently and in real time.

Take a moment to reflect on how far you've come—these core ideas will keep surfacing throughout the course, especially during hands-on labs and your final capstone project.