Getting Started with

Entity Event-Driven Embedded Systems

Volume 2: Pulse Width Modulation and DC Motors

Raspberry Pi Pico W and Pico 2W

Cassandra
STEAM Press

Byron Mattingly

Copyrighted Material

Getting Started with Entity Event-Driven Embedded Systems

Vol 2. Pulse Width Modulation and DC Motors: Pico W and Pico 2W

www.CassandraSTEAMPress.com

Copyrighted Material

Getting Started with Entity Event-Driven Embedded Systems
Vol 2. Pulse Width Modulation and DC Motors: Pico W and Pico 2W
by Byron Mattingly
ISBN: 979-8-90046-788-7
Copyright © 2025 Byron Mattingly.
Publisher: Cassandra STEAM Press, www.CassandraSTEAMPress.com
October 2025: First Edition

www.CassandraSTEAMPress.com

Except where otherwise noted, this book is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) which is available at:
https://creativecommons.org/licenses/by-nc-sa/3.0/

All included source code examples are available for download from GitHub under the MIT license:
https://github.com/cassandrasteampress/event-driven-pico-code-vol2

The information in this book is distributed on an “As Is” basis, without expressed or implied warranty of any kind. While every precaution has been taken in the preparation of this book, the publisher and contributors assume no responsibility for errors or omissions, or for the use of the information contained in this book or in respect of any errors or omissions relating to goods, products or services referred to or advertised in it. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained in this book or with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained or programs in it.

Cassandra STEAM Press and the Cassandra Software, Inc. logo are trademarks of Cassandra Software, Inc. Other designations, product and company names mentioned in this book may be claimed as trademarks by their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, the inclusion of the trademarked name is used for editorial purposes only and to the benefit of the trademark owner, with no intent of trademark infringement.

Copyrighted Material

for Lisa
sine qua non

www.CassandraSTEAMPress.com

Copyrighted Material

About the author

Byron Mattingly has been an embedded systems software engineer and hands-on technical manager for over 20 years in regulated development in the medical devices, pharmaceutical, and avionics industries. Currently his work is focused on designing and integrating complex IT systems and training and deploying AI/ML models interacting with embedded systems. An early adopter of the Raspberry Pi platform, he is an open source contributor and avid proponent of STEAM education.

www.CassandraSTEAMPress.com

Copyrighted Material

Table of Contents

Preface vii

1. Entity Event-Driven Embedded Pico Systems 1
Learn how you can transform your Pico into an Entity Event-Driven Embedded System

2. Turn digital to analog with Pulse Width Modulation (PWM)............21
Use PWM to simulate an analog output signal on your Pico

3. The PWM whisperer: an introduction to controlling servo motors ..75
Discover how to position an SG90 servo precisely with a PWM signal

4. Rev up! Slow down! Forward! Reverse! What's driving the DC motor revolution? 117
Learn how to drive a TT DC Gearbox Motor with an H-bridge and PWM

5. Driving and controlling a stepper motor (and that's not the half of it) 163
Start full and half stepping a 28BYJ-48 stepper motor with a Darlington Array

APPENDICES

A. Raspberry Pi Pico W pinout guide 235

B. Raspberry Pi Pico W Features..237

C. Raspberry Pi Pico 2W pinout guide 239

D. Raspberry Pi Pico 2W Features..241

E. Raspberry Pi fixes Erratum E9 bug affecting the Pico 2W 243

F. Attributions and License Information 245

www.CassandraSTEAMPress.com

Copyrighted Material

www.CassandraSTEAMPress.com

Copyrighted Material

Preface

Motors make things move.

There are many types of motors. Countless books, blogs, articles and websites—as well as entire careers—are dedicated to the study of motors. Just to name a few topics that make up this field: energy efficiency, power consumption, torque, prolonging a motor's lifespan, recycling options (and what to do with the e-waste after discarding a motor's components).

This book offers a streamlined introduction to the complex world of DC motors in Entity Event-Driven Embedded Systems. Remarkably, the heart of such systems can be a Raspberry Pi Pico, a fast, versatile, and energy-efficient microcontroller board that debuted in 2022 for $4 USD and which, at the time of this writing, now costs around $7 to $8 USD in its latest versions.

The Raspberry Pi Pico is communications-savvy. Using its GPIO pins and Pulse Width Modulation (PWM) capabilities, the Pico W or Pico 2W (either Pico variant works with the examples in this book) can communicate with some of the most common but low-cost DC motors found in hobbyist, maker and educational projects available today. Going through the book's examples shows how to integrate these inexpensive DC motors into Entity Event-Driven Embedded Pico Systems using MicroPython's asyncio library and the Microdot webserver.

Building on the knowledge from *Volume 1: GPIO and Asynchronous Programming with Microdot* of this series on *Getting Started with Entity Event-Driven Embedded Systems*, we will be taking a step-by-step approach on how to provide microservices and other data consumers / AI agents important "real-time" information about the operating characteristics of DC motors. Using PWM, you'll be learning how to brighten and fade LEDs and how to control servo and stepper motors. You'll be able to experiment with single phase on and two phase on as well as half-step sequencing of a stepper motor.

With just a handful of MicroPython code and either the Pico W or Pico 2W, we'll see how easy it is to control a:

- **Servomotor.** SG90 Micro Position Servomotor
- **TT Motor.** TT DC Gearbox Motor + DRV8833 Dual H-Bridge Motor Driver
- **Stepper Motor.** 28BYJ-48 5V Stepper Motor + ULN2003 Motor Driver

www.CassandraSTEAMPress.com

Copyrighted Material

Some motor assemblies, like the SG90 servo, include a built-in motor control circuit that interprets PWM signals in a closed-loop feedback system, making it possible to precisely control the servo with minimal code.

Others, like the TT motor require a separate motor drive controller. The DRV8833 H-bridge motor driver enables speed control with PWM and is able to change a motor's direction by reversing polarity. It also features a "braking" mode that can help slow a motor down by converting the motor's kinetic energy into electrical energy.

The 28BYJ-48 stepper motor is often sold in combination with the ULN2003 motor driver which includes a Darlington Array that allows the stepper motor to be turned in precise angular increments by setting the Pico's GPIO pins high or low according to a pre-defined step sequence.

All of these motor drive controllers have simple and straightforward interfaces in common. They also include circuitry and algorithms that improve a motor's energy efficiency while extending the motor's useful life.

To begin, you'll need a Raspberry Pi Pico W or Pico 2W, a few inexpensive electronic components (like a breadboard, some LEDs, the DC motors listed above, etc.) and to know a little Python ahead of time. Fortunately, there's a tremendous amount of comprehensive, free educational resources to help out in this last regard. The Raspberry Pi foundation (https://www.raspberrypi.org/), for instance.

The introductory chapter addresses the question *Why Entity Event-Driven Embedded Pico Systems?* It sets up the theory behind the later chapters. (It's repeated in some of my other books because it forms the basis for them, too.) Feel free to skip it and come back to it later once you've started building some Entity Event-Driven Embedded Pico Systems of your own that include DC motors.

Then we start experimenting with and learning about your Pico W or Pico 2W, writing and explaining each MicroPython program along the way so that by the time you work through all of the examples and reach the end of the book, you will have a very good idea of the basics of controlling DC motors in your own *Entity Event-Driven Embedded Pico Systems.*

www.CassandraSTEAMPress.com

Copyrighted Material

Example Code

The code examples used in this book can be downloaded from GitHub:

https://github.com/cassandrasteampress/event-driven-pico-code-vol2

To download all the files as a ZIP archive, click on the *Code* button and then *Download ZIP*. After the download has completed, you can extract the contents to a convenient location on your computer.

Electronics Hardware

Information for the electronics used in this book can be found at:

https://github.com/cassandrasteampress/event-driven-pico-kits

Acknowledgements

I used Fritzing (fritzing.org) for the breadboard layouts and component diagrams, Inkscape (inkscape.org) and GIMP (https://www.gimp.org/) for the other illustrations.

www.CassandraSTEAMPress.com

Chapter 1

1. What is an Event-Driven Architecture?

2. Why Entity Event-Driven Embedded Systems?
    a. What are Entity Event-Driven Embedded Systems?
    b. What are Entity Event-Driven Embedded Pico Systems?

3. Developing Entity Event-Driven Embedded Pico Systems
    a. Host / Target Setup
    b. A System of Systems

4. Defining Embedded Systems
    a. What is an Embedded System?
    b. Intended Use and User Requirements
    c. Working with Constrained Resources
    d. When Less is More

5. Defining Embedded System Entity Events
    a. What are Entity Events?
    b. Benefits of Entity Events

6. Setting Context Boundaries
    a. Communication Interfaces
    b. Bounded Context of a Domain Model
    c. Modelling Bounded Contexts
    d. A Simpler Notation: Hexagons with interfaces

7. Some Notes before Proceeding
    a. Avoiding Technical Jargon, Definitions and Generalizations

8. Chapter Summary
    a. What is the purpose of your Entity Event-Driven Embedded Pico System?

9. References and Further Exploration

www.CassandraSTEAMPress.com

Copyrighted Material

Entity Event-Driven Embedded Pico Systems

Learn how you can transform your Pico into an Entity Event-Driven Embedded System

www.CassandraSTEAMPress.com

Since the late 1980s and early 1990s, *Data-Driven Architectures* have become widely adopted for data-driven approaches to complex decision-making. This architectural paradigm focuses on integrating data across various layers of an organization, data processing and analysis, and leveraging data analytics and insights as core elements that shape informed decision-making and system functionality. Data-Driven Architectures are particularly effective in environments like data warehouses which prioritize efficiently storing, processing, and analyzing data to facilitate better decision-making.

In the late 1990s and early 2000s, as systems became more distributed and required real-time processing, *Event-Driven Architectures* have gained prominence, fueled by the need for a more flexible and responsive architecture. Due to their ability to respond to events with low latency and handle high volumes of data, this pattern emerged for creating high-performance and scalable designs.

Event-Driven Architectures are ideal for real-time, reactive systems that require rapid processing and immediate responses to events as they occur. These designs are usually made up of strongly decoupled applications such as microservices that asynchronously receive and process events produced by other applications—financial trading systems, real-time monitoring systems, and embedded systems, for example. They are designed to be highly adaptable, robust, and efficient.

This book is on how to start transforming your Pico W or Pico 2W into an *Entity Event-Driven Embedded System* that can actively participate in Event-Driven Architectures.

Copyrighted Material

Why Entity Event-Driven Embedded Systems?

The event-based model differs from a request-based model which reacts to requests from a customer web page—for instance—to retrieve an order history. Such requests can be data-driven and deterministic, bounded by a specific context. An event-based model, on the other hand, responds asynchronously to particular situations the system must react to and not to a specific request, *per se*.

Events are records of occurrences over time. Not only have modern Event-Driven Architectures enabled the real-time consumption of such events in very large datasets, these events can persist at scale and be made available to any service interested in them as often as needed.

While the need for producing events is not new, the need for systems that communicate events that contain sufficient information for applications like microservices and multiple data consumers to process those events according to their specific use cases *is* definitely new. In particular, embedded systems must now interface with microservices and other data consumers in a variety of different contexts. Providing these applications with a stream of quality data in the form of uniquely identifiable Events and in a format that can be readily understood is ultimately the goal of Entity Event-Driven Embedded Systems.

Recently, AI agents are starting to go beyond “request-and-respond” frameworks and are transitioning to autonomous systems capable of reasoning, planning, and orchestrating complex dataflows and workflows based on high-level goals. AI agents in real-time Event-Driven Architectures continually process inputs, sifting through and making sense of Entity Events that can originate from endpoints like embedded systems. AI agents not only consume data, they can play an active role in determining Events in Event-Driven Architectures. For example, AI agents coordinating a city’s traffic light system could dynamically re-route traffic patterns by adjusting light sequences on individual traffic lights based on sensory input (e.g. radar, lidar, and button presses at pedestrian crossings) gathered at traffic intersections.

www.CassandraSTEAMPress.com

Copyrighted Material

What are Entity Event-Driven Embedded Systems?

An *Entity Event* is something of interest to an application in an Event-Driven Architecture that took place which is directly associated with a specific entity. Entities can be a system component, an object or a user which generally have attributes describing their current state. An Entity Event provides sufficient contextual information for a microservice or other data consumer / AI agent to understand what happened and what or who was involved. An Entity Event is always linked to a single, distinct entity in an Event-Driven Architecture.

Entity Event-Driven Embedded Systems produce events of interest to microservices and other data consumers / AI agents. Building Entity Event-Driven systems is about creating systems that asynchronously *generate* events in real-time in Event-Driven Architectures.

In physical computing, we define an Entity Event for an embedded system as something significant and uniquely identifiable (e.g. with a UUID, system ID and universal timestamp) that occurs to which its application responds, either by taking a measurement of the physical world (e.g. a sensor reading like temperature) or taking an action in it (e.g. controlling a component like a motor). The system reacts to a *situation* as a kind of stimulus and is able to communicate what happened and its response.

What are Entity Event-Driven Embedded Pico Systems?

Entity Event-Driven Embedded Pico Systems are small and purpose-built. Their internal applications connect to electronic components that interact with the physical world, sensing and controlling it. These applications also produce output event streams. They communicate data in machine-to-machine (M2M) exchange formats (like JSON) to event brokers through their APIs.

For the examples in this book, you'll include just enough data in an Entity Event relevant to acting on the event. This includes a unique Entity Event identifier (a UUID tag) and information that addresses the questions *Who* (which Pico system)? *What* (the data payload)? and *When* (a universal timestamp)?

Entity Event-Driven Embedded Pico Systems that provide this minimal information through their network interfaces can thereby provide output event streams for microservices and other data consumers / AI agents whose needs and goals for brokering and orchestrating events in modern Event-Driven Architectures may range from simple to complex.

www.CassandraSTHAMPress.com

Copyrighted Material

Entity Event data payloads capture structured information. *What* to include in the data payload, however, depends entirely on the applications that utilize that information. The "thicker" the Event, the more extensive the payload, the more time it takes to generate and to process the Event. Overly thick events can slow the overall responsiveness of an Event Driven Architecture. A balance must always be found between not enough and too much information. An Entity Event must only contain the essential details about an event; nothing extra. Microservices and other data consumers / AI agents should be able to process the Entity Event directly without relying on other systems for core event information.

Developing Entity Event-Driven Embedded Pico Systems

Host / Target Setup

Your computer (the host)

We'll use a Windows 11 PC and / or Raspberry Pi OS on a Raspberry Pi 4 / 400 host computer for developing and writing MicroPython code that runs on the Pico W or Pico 2W as the target system. The example projects work equally well under both—you may use either Pico variant. Since a new generation macOS computer (e.g. running macOS Sequoia (15.x) or macOS Tahoe (26.x)) behaves similarly to the Raspberry Pi OS computer (their underlying operating systems are both Unix-like), you can use it instead as the host system, if you prefer.

www.CassandraSTEAMPress.com

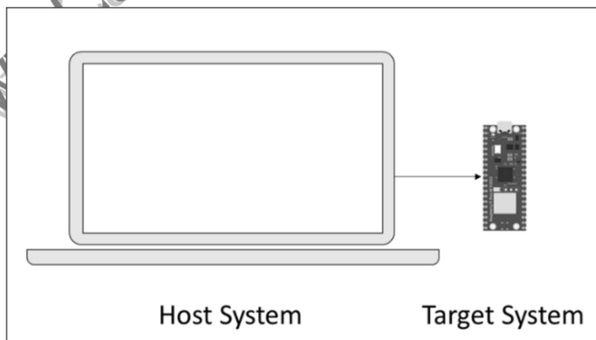


Fig 0-x Host System to Target Pico

Copyrighted Material

Your Raspberry Pi Pico (the target)

In 2020, as a way of promoting learning about embedded systems programming in schools and the maker / hobbyist community, the Raspberry Pi Foundation introduced the Pico for $4 USD, a microcontroller board about the size of a stick of chewing gum that incorporates an RP2040 microcontroller.

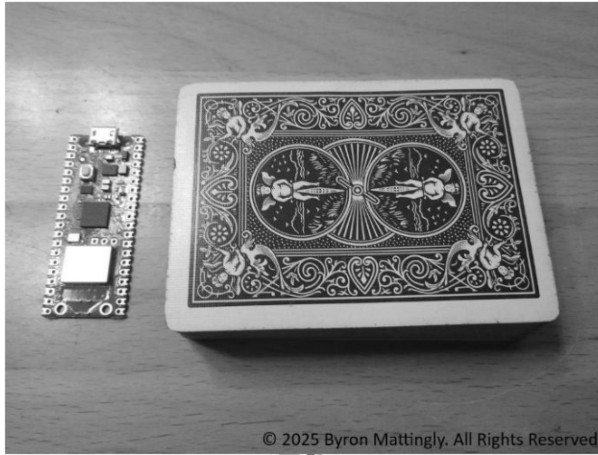


Fig 1-x. Pico W and Playing Cards

In 2022, the Raspberry Pi Foundation introduced the Pico W for $6 USD which adds an Infineon CYW43439 single-chip combo device to the original Pico. This addition enables Wi-Fi 4 (802.11n) and BLE 5.2 (Bluetooth Low Energy) connectivity.

According to Raspberry Pi CEO Eben Upton, as of February 29, 2024, approximately 4 million Picos had been sold up to that point:

- https://www.tomshardware.com/raspberry-pi/raspberry-pi-celebrates-12-years-as-sales-break-61-million-units

In November of 2024, the Raspberry Pi Foundation introduced an upgraded version of the Pico W as the Pico 2W for $7 USD.

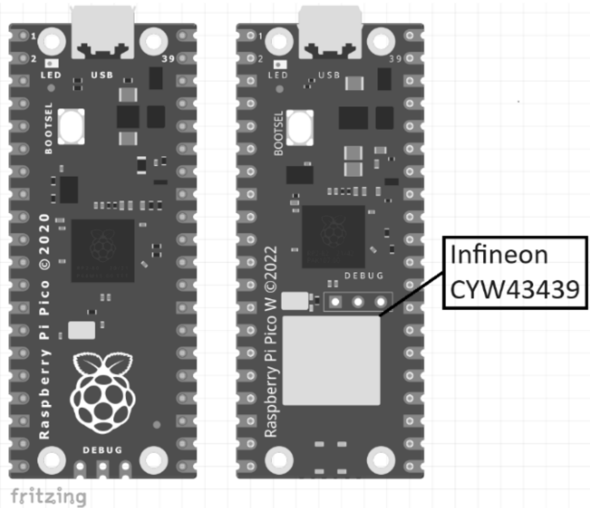


Fig 1-x. The Raspberry Pi Pico (left) and Pico W (right)

At the time of this writing, both the Pico W and Pico 2W sell for about $7 USD to $8 USD apiece, respectively, making it easy and convenient for students, makers and hobbyists to connect electronic components to it using a solderless breadboard.

For the projects in this book, you can use either the Raspberry Pi Pico W or the Pico 2W. They work equally well with the code examples. We'll refer to both variants simply as the "Pico".

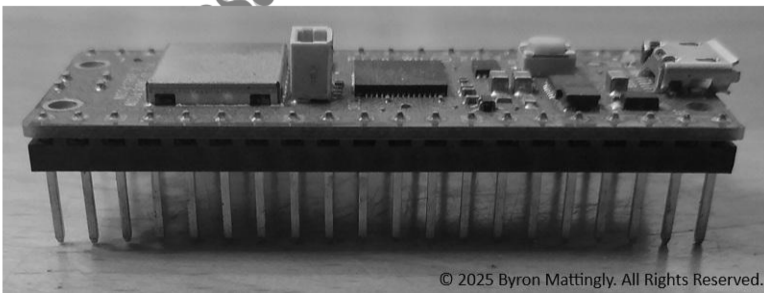


Fig 1-x. Pico WH (Pico W with Headers)

Copyrighted Material

A system of systems

Many different kinds of external electronic components (e.g. LEDs, displays, sensors, and motors) can be connected to a Pico. In general, these physical components are considered separate subsystems (self-contained systems within a larger system). They are components that talk to the Pico microcontroller board (itself a system) through a communications interface which, depending on the component, can be as simple as a holding the voltage on a GPIO pin on the board high or low or as complicated as a wireless exchange using HTTP or BLE protocols.

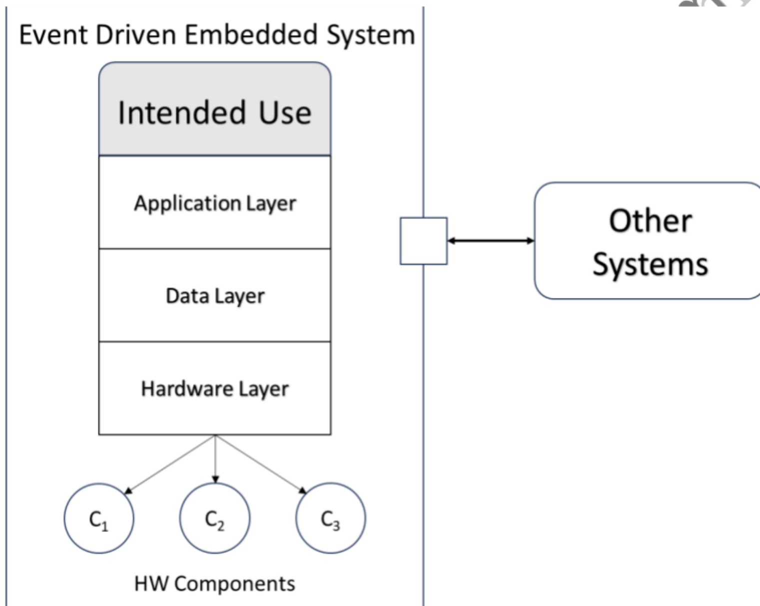


Fig 1-x. Entity Event-Driven Embedded System of (sub)systems

The Pico and its connected subsystems is a "system of systems" with the programs you write in MicroPython communicating with its various subsystems (components) and thereby sensing and controlling the physical world. Although an embedded system like the Pico can operate autonomously, an embedded system often communicates and interacts with other embedded systems and computer systems. It depends entirely on its intended use and its context of use.

Copyrighted Material

Defining Embedded Systems

What is an embedded system?

Unlike a general-purpose computer, an embedded system is a computerized system designed for a specific purpose, to do one thing and to do that thing and well.

Intended Use and User Requirements

www.CassandraSTEAMPress.com

For the examples presented throughout this book on embedded Pico systems, the *purpose* of an embedded system can be defined in terms of its intended use and the user requirements that support the intended use in its context of use. Quite simply, the *purpose* of an embedded system answers the two questions, *What is this application supposed to do* (the intended use)? and *How does this application achieve that* (the user requirements)?

To illustrate the difference between intended use and user requirements: the purpose of a two-car residential garage may be to store vehicles, bicycles, and some outdoor lawn and garden equipment and to protect these from the elements. *How* it achieves that purpose might be with a covered roof, insulated walls, garage doors, a cement floor, etc.

Understanding the specific purpose of an embedded systems application is essential in keeping it simple and robust. Complex designs that are supposed to do many different things with lots of interconnected different parts are inherently costly to design, build and operate. Moreover, they tend to be complicated, break easily, and are expensive to repair or replace.

Working with Constrained Resources

Embedded systems are quite often based on microcontrollers (such as the RP2040 or RP2350) mounted on a PCB board designed by electrical engineers (like the Pico W which is a microcontroller board for the RP2040 or the Pico 2 W which is a microcontroller board for the RP2350). But microcontrollers are by their very nature resource

Copyrighted Material

constrained. Programming them can be a challenge because they are limited by:

- Processor speed
- Code space (ROM or flash)
- Memory (RAM)
- Processor peripherals
- Power consumption

To avoid unnecessary complexity and complication, the best approach to designing microcontroller-based embedded systems is to make them as simple as possible (but, to paraphrase Einstein, no simpler!) for their specific application—make them *do the right thing* and *do that thing right* but minimize the system's dependencies on other systems. Put differently, the simpler you can make such an embedded system that still does what it is supposed to do and do it correctly while keeping it loosely coupled to other systems, the better.

How do you simplify complex designs? Modularize the individual subsystems (self-contained systems that are part of a larger system) that comprise them and, at the same time, make sure that their subsystems integrate well with each other and the other systems to which they might interface.

www.CassandraSTEAMPress.com

Less is More. Simpler is better.

Whether you are a hobbyist / maker or a technical professional, there is an upside to tailoring an embedded system according to its resource constraints. They are more eco-friendly. Such systems are less wasteful of materials, of power consumption, and are easier to design and less costly to build and maintain than systems that try to do lots of different things with lots of different parts. The simpler you can make an embedded system that still does what it is supposed to do and do it correctly, the better.

In fact, one of the greatest challenges in designing and developing large complex systems on any project is that their subsystems don't always work well together. Integration testing done too late in a project's lifecycle frequently reveal incompatibilities that kill a project outright because of budget. Simpler systems are easier to understand, to try out firsthand, and to debug. Their development lifecycles are shorter.

A commercial aircraft, for example, can have millions of parts, all of which that must integrate flawlessly as planned. If they don't, costly rework or schedule overruns might result during development or after

Copyrighted Material

the aircraft goes into production. Worse, as a result of integration errors, an aircraft may experience a catastrophic failure. But even for small learner and small hobbyist / maker projects based on the Pico, it can be frustrating to redo and rebuild parts of an embedded system over and over again. It's a huge waste of time. No one wants that.

Defining Embedded System Entity Events

What are Entity Events?

Entity Events record data acquired from sensing or controlling the embedded system itself and / or the physical world around it. They provide *metadata* along with that data which provide sufficient contextual information required to identify and accurately describe that data. They record what happened to an entity and the actions it might have taken.

Entity Events are most often represented in a key / value format. The key uniquely identifies the event, while the values store the details of the event. We will structure Entity Events that uniquely identify the properties and state of our Pico-based embedded systems at a specific point in time such that the resulting event streams emitted by these systems can be constructed or reconstructed if necessary. Example:

Key	Value
event_id	3e93b24c-6345-4009-af70-a1ac3548e2fb
time_unix	2147483647
machine_unique_id	314159265358979323846zza

Fig 1-x. An embedded systems Entity Event is keyed on the unique event ID

In the examples in this book, your Pico will uniquely identify Events that are associated with entities (e.g. systems, objects, humans) and communicate those events asynchronously in the form of JSON objects (think Python dictionaries and lists) for machine-to-human and machine-to-machine communications. In so doing, your Entity Event-Driven Embedded Pico Systems will be able to participate in multiple domain contexts as a "source of truth" in Event-Driven Architectures.

Since Entity Events are uniquely identified by their event IDs, a microservice that receives Entity Events from one of your embedded Pico systems can both broker a queue of such events and

www.CassandraSTEAMPress.com

Copyrighted Material

simultaneously preserve the *data integrity* of that queue because it has all the information for doing so. If necessary, it can tell if something's changed in the queue.

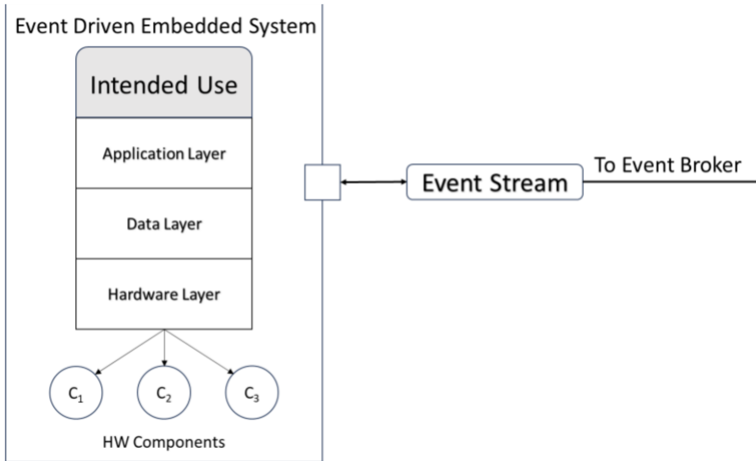


Fig 1-x. Entity Event-Driven Embedded System as an event stream source

www.CassandraSTEAM

The same type of interaction likewise holds true for a data consumer / AI agent consuming Entity Events from an embedded Pico system. These data consumers both can both broker a queue of such events and simultaneously preserve the *data integrity* of that queue because they have all the information for doing so, and, if necessary, determine whether something has changed in the queue.

Benefits of Entity Events

There is a significant benefit to constructing embedded systems with Entity Events as a key element of their designs. The Agile principle of "testing early and testing often" especially applies to communication interfaces. Agile testing can trace back to original sources if events are uniquely identified as they are with Entity Events.

To illustrate: the "babbling sensor" problem in avionics where a sensor on a flap or aileron fails to measure the force a pilot supplies to the flight control surface but nevertheless floods the onboard network with nonsensical readings. Since a flight control computer for a fly-by-wire (FBW) system must determine how to manipulate the control surfaces of a jet aircraft while it is airborne, it becomes critical to identify and isolate the defective sensor from the other sensors on the network.

Entity Events make this diagnostic task much easier because the readings from each sensor are uniquely identified. You know the point of origin for the event. It's easier to test for and to debug.

Fig 1-x. Babbling sensor on a flight control surface

www.CassandraSTEAMPress.com

Because of their uniqueness, Entity Events make it simpler to design and build testable interfaces during a project lifecycle. As a project progresses, integration tasks are made easier when the data associated with events are uniquely and unambiguously identified. Even better: the sooner you can test an interface for your system and the sooner you can progressively and iteratively integrate that system with other systems, the sooner you can see how well it works in its context of use. That's a win for any development task.

When developing safety critical systems for regulated industries such as avionics, identifying and controlling risks of all kinds (security risks and risks to people, to other systems, to the environment, to business needs, etc.) is the first and foremost consideration. The earlier a design focuses on the interface requirements between systems and subsystems, the better. Identifying these requirements and associated messaging along the communication boundaries is essential to analyzing and managing the risks of a particular system. These requirements not only define what a system does but provide a means to test it. Do the system and its subsystems meet their interface requirements? Is the integrated system under test *doing the right thing* (its purpose) and is it *doing it right* (working correctly)?

Copyrighted Material

Setting Context Boundaries

Communication interfaces set context boundaries

The guiding principle of this book is to focus on the communication interfaces between systems and subsystems right away and to stay focused on and keep testing these interfaces until your project is done—maybe even longer(!). While it is impossible to anticipate every possible scenario under which your Entity Event-Driven Embedded Pico System will have to communicate, it is definitely possible to design communication interfaces that are simple and easy to use right from the start.

Importantly, these interfaces can help explicitly define the context boundaries of your embedded system which can then make it easy for you and for others to use and later *re-use* them in different projects and different architectures involving other systems. This beginning book on the Pico attempts to show you how to do so by defining the messages that are sent and received across a communications interface as an Entity Event.

www.CassandraSTEAMPress.com

Bounded context of a domain model

Complex systems are inherently collections of multiple domain models defined by the range of their applicability according to their bounded contexts. Although communication interfaces are by no means the only way to define a bounded context, it is often a very straightforward way to do so—testing of a given context amounts to communicating with the boundary that contains its subdomain.

Typically, a communications interface encloses just one embedded system. Communications between this system and other systems are via network calls which enforces decoupling. Bounded contexts help minimize a system's interdependence on other systems and domains so that changes made to it minimally impact its neighboring contexts.

Copyrighted Material

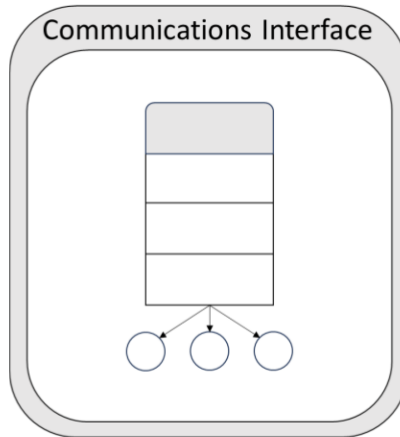


Fig 1-x. Bounded context of a single system defined by its communications interface

Designed with the communications interface in mind, the embedded system is readily available for other systems to interact with as required, provided those systems adhere to its interface requirements. Different microservices and data consumers / AI agents, for instance, might broker different event queues with multiple embedded Pico systems.

Modelling bounded contexts

For large complex projects in regulated industries, a formalized interface requirements document like an ICD (Interface Control Document) may be required to describe the different configurations for exchanging events over a bounded context with an Event Broker. For example, here's how a Systems Engineer might use SysML notation to model this particular use case between a Pico embedded system and a microservice Event Broker:

www.CassandraSTEAMPress.com

Copyrighted Material

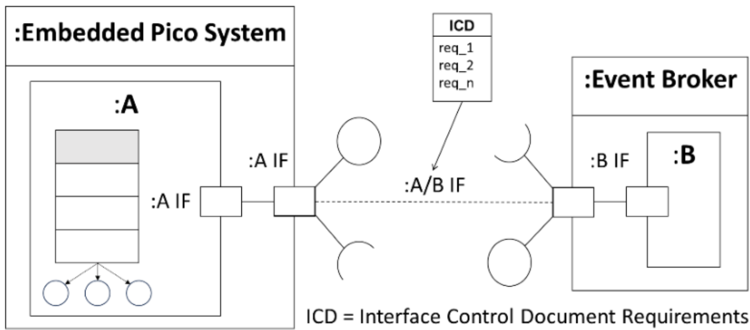


Fig 1-x. SysML Systems Engineering Model of a communications interface defining a bounded context

A systems engineer modelling the bounded context of an embedded system might view the system as having multiple communications interfaces even though the internal workings of that particular system remain the same, just like an application might offer multiple APIs. This book won't begin to cover all these possibilities but only caution against the temptation to create so many interfaces that they can become unwieldy to manage as well as introduce other difficulties (such as documenting them).

www.CassandraSTEAMPress.com

A Simpler Notation: hexagons with interfaces

We do not need to elaborate on the application, data and hardware architectural layers working inside an Entity Event-Driven Pico Embedded System. Our hobbyist / maker embedded systems are simple MicroPython applications running on the Pico often with just a few dozen lines of code. The application implements one or more interfaces that communicate Entity Events to external systems.

For us, the most important information about an Entity Event-Driven Pico Embedded System is the intended use of that system and what is required by the interface that its application provides. Put differently, what's the purpose of the Pico system and how should other systems interact with it?

Copyrighted Material

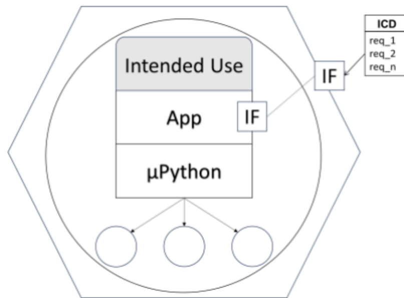


Fig 1-x. Pico embedded system using hexagonal notation

CassandraSTEAMPress.com

To the extent possible, our aim is to reduce embedded systems based on the Pico to the point that they are no longer considered complex. This is the (IEEE / INCOSE / SERC) systems engineering definition of a "component" (https://sebokwiki.org/wiki/Component_(glossary)). Simple embedded systems, when designed well, are both coherent (logically self-consistent) and cohesive. They are easier to understand, design and control throughout their lifecycle than complex systems of systems. Systems of systems made up of redundant components are robust—certainly more robust than systems of systems that are redundant at the system level.

As a component, an Entity Event-Driven Embedded Pico System is a basic modular building block that can be used and re-used in multiple configurations depending on its interfaces. Starting with very simple applications, we will stay focused in this book on each system doing one thing well and on communicating Entity Events with its interface(s).

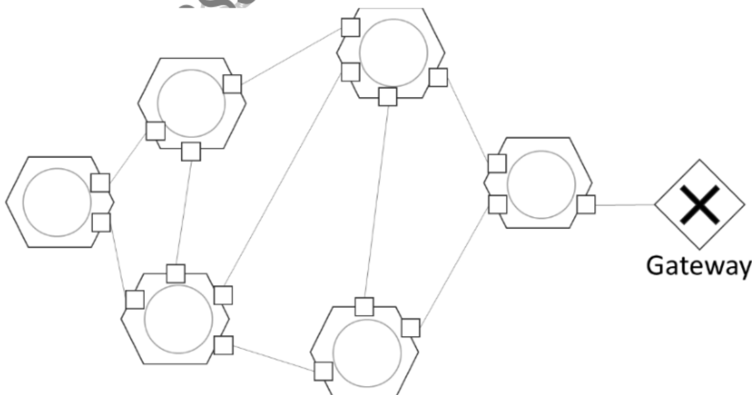


Fig 1-x. Example of Pico embedded systems in a mesh network connected to an exclusive gateway

Copyrighted Material

Some Notes before Proceeding

Avoiding Technical Jargon, Definitions and Generalizations

You'll notice that throughout this chapter I have tried to avoid technical jargon, definitions and generalizations. In a rapidly changing technology like embedded systems, these evolve frequently, whereas the core design principles underlying the technology change more slowly. In the long run, understanding basic concepts is what matters, such as the importance of designing simple systems that do one thing well but that also emit Entity Events. By progressively working through the examples in this book, I'm hoping you'll find that a very good way of keeping embedded systems as simple as possible is through bounded contexts defined by their communication interfaces.

Of course, different disciplines have different point of views on how best to do this: an electrical engineer, for example, may look at an embedded system quite differently than a systems engineer would, or a software engineer might. These subject matter experts are like the proverbial blind men describing an elephant's various parts in terms of their individual perspectives.

For this reason, I have tried to avoid hard and fast rules. Invariably, there are numerous exceptions and corner cases when applying definitions and generalizations to the simplest of embedded systems. It can't be helped. Use cases vary by context. Contexts change over time. Definitions change as generalizations evolve to account for new types of situations. Everything changes as the technology elephant keeps plodding forward, inexorably.

For example, an electronic communications interface defined by a few GPIO pins on the Pico is quite different than a wireless communications interface using the Pico's WiFi or BLE subsystems. When to use which type of interface depends entirely on the use case. What system or subsystem does the Pico need to interface to? What assumptions can be made about the context of use?

Whether internal or external, interfaces have to be understood on their own terms, their pros and cons. For the purposes of learning how to transform your Pico into an Entity Event-Driven Embedded System, I gloss over some of these differences with all due respect to the subject matter experts who would prefer a catalog of distinctions and nuances. —*I understand*. My rationale: I don't want to assume that a beginning learner has the benefit of specialty technical knowledge in any particular field. Learning basic concepts and the tools that go with them requires an incremental bootstrap approach.

www.CassandraSTEAMPress.com

Copyrighted Material

Chapter Summary

Does your embedded Pico system do the right thing and do the thing right?

With respect to learning how the Pico fits into your Entity Event-Driven Embedded Systems projects, I believe that the questions inevitably come down to: does your Pico do what it's supposed to do, and does it do it correctly and simply as possible for the given use case? In other words, does it *do the right thing* and *do the thing right*?

Let's see how to answer these questions with MicroPython and your Pico!

References and Further Exploration

- *SEBoK Component (glossary)* https://sebokwiki.org/wiki/Component_(glossary)

- *Fundamentals of Software Architecture*, 1st edition by Mark Richards and Neal Ford (O'Reilly, 2020, ISBN-13: 978-1492043454,)

- *Software Architecture: The Hard Parts*, 1st Edition by Neal Ford and Mark Richards et al, (O'Reilly, 2021, ISBN-10: 1492086894)

- *Building Event Driven Microservices* by Adam Bellemare (O'Reilly, 2020, ISBN-13: 978-1492057895)

- *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (Addison-Wesley, 2003, ISBN-10: 0321125215)

- *Making Embedded Systems*, 2nd edition by Elecia White (O'Reilly, 2024, ISBN-10: 1098151542)

www.CassandraSTEAMPress.com

Copyrighted Material

www.CassandraSTEAMPress.com

Copyrighted Material

Chapter 2

1. Chapter Introduction

2. Programming Environment and Setup
    a. Introduction
    b. Equipment and Components
    c. Setup and Construction
    d. Code

3. PWM Basics
    a. Introduction
    b. Code
    c. Procedure / Steps
    d. Tests and Results

4. Controlling the brightness of an LED using PWM duty cycles
    a. Introduction
    b. Code
    c. Procedure / Steps
    d. Tests and Results
    e. Questions

5. Using a potentiometer to fade an LED with PWM duty cycles
    a. Introduction
    b. Code
    c. Procedure / Steps
    d. Tests and Results

6. Turn your PWM-based LED system into an Entity Event-Driven Embedded Pico System
    a. Introduction
    b. Procedure / Steps
    c. Steps
    d. Tests and Results

7. References and Further Exploration

www.CassandraSTEAMPress.com

Copyrighted Material

Turn digital to analog with Pulse Width Modulation (PWM)

Use PWM to simulate an analog output signal on your Pico

Your Pico interfaces to external hardware through the 40 pins running along both of its edges. 26 of these are General Purpose Input / Output (GPIO) pins. While some of these pins have a predefined purpose (like VBUS or GND), others have multiple features that allow them to talk to a variety of electronic components, ranging from blinking LEDs or reading analog sensor data (like a potentiometer) to communicating through specialized interface protocols (like I2C or SPI).

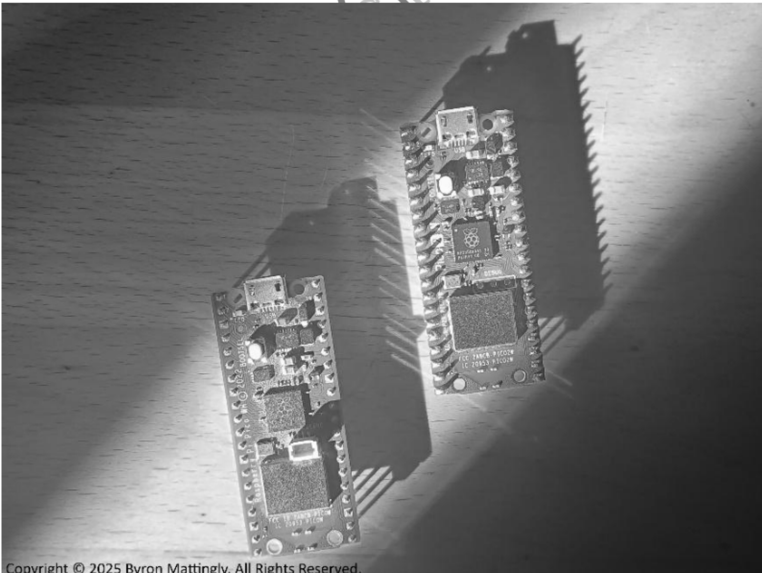


Fig 2-x. The Pico WH (left) and Pico 2WH (right)

(Refer to Appendix A and Appendix C for a pinout guide to the Raspberry Pi Pico W and Pico 2W, respectively.)

Copyrighted Material

GPIO pins are fundamental to many applications, including:

- **Robotics:** Interfacing with sensors, motors, and actuators used in robots.
- **Environmental Control:** Automating lights, fans, and other appliances (in the home, for example).
- **Environmental Monitoring:** Collecting data from temperature, humidity, and pressure sensors for environmental analysis.
- **Data Logging:** Recording and reporting sensor data over time.
- **Internet of Things (IoT):** Remote control and monitoring capabilities of devices connected to the internet.
- **Custom Hardware Interfaces:** Specialized solutions for unique tasks.

GPIO pins provide a versatile interface for a microcontroller board like the Pico to connect to the physical world.

Pulse Width Modulation (PWM) is a technique widely used in electronics for achieving fine-grained control over the average voltage delivered to an electronic device by varying the width of the "on" pulse of a digital signal at regular intervals. MicroPython interfaces to the hardware on your Pico in a straightforward way through its GPIO pins, making it relatively easy to implement PWM in your Entity Event-Driven Embedded Pico Systems.

PWM has a wide range of applications such as:

- Adjusting LED brightness
- Setting the angular position of servo mechanisms
- Controlling the rotational speed of DC motors
- Generating audio tones and effects
- Power regulation

This book takes advantage of electronic components commonly included in hobbyist and educational kits. We'll set them up with various "duty cycles" and different frequencies to demonstrate the basic concepts behind PWM and how you can use PWM in your Entity Event-Driven Embedded Pico Systems. In this chapter, we'll experiment with LEDs and a potentiometer to illustrate and understand the basic concepts behind PWM. In later chapters, we'll use PWM to control servo and DC motors.

The Pico's onboard analog-to-digital converters (ADCs) can convert analog signals to digital data that the Pico can understand and process. The Pico, however, does *not* have an onboard digital-to-analog converter (DAC) that can go in the other direction. Without a DAC, the Pico cannot convert digital data into continuous, variable analog waveforms.

www.CassandraSTEAMPress.com

Copyrighted Material

But the Pico can *approximate* an analog signal through a GPIO pin by rapidly switching a digital signal from ON=1 (its HIGH state) to OFF=0 (its LOW state), thereby "pulsing" that pin. The longer the pin remains ON during that interval, the wider the pulse width. Changing the pulse width *modulates* the average voltage the pin delivers to a load—hence the term "pulse width modulation". Using PWM, a GPIO pin can therefore appear to deliver a voltage between 0V (its LOW state) and 3.3 V (its HIGH state) to an attached electronic device.

The total time that a GPIO pin spends making this transition from ON to OFF is known as the pulse period (or cycle time) which is the inverse of the PWM frequency.

$$Pulse\ Period = t_{on} + t_{off} = \frac{1}{PWM\ frequency}$$

The PWM frequency is the number of complete on-off cycles of the PWM signal per unit time. On the Pico, the PWM frequency can be set from 8Hz to 62.5Mhz (see the PWM section in MicroPython.org's Quick reference for the RP2:
https://docs.micropython.org/en/latest/rp2/quickref.html).

By varying the *duty cycle*—how long a GPIO pin stays ON in a given pulse period—the Pico can deliver an *average* voltage to an electronic device connected to the pin.

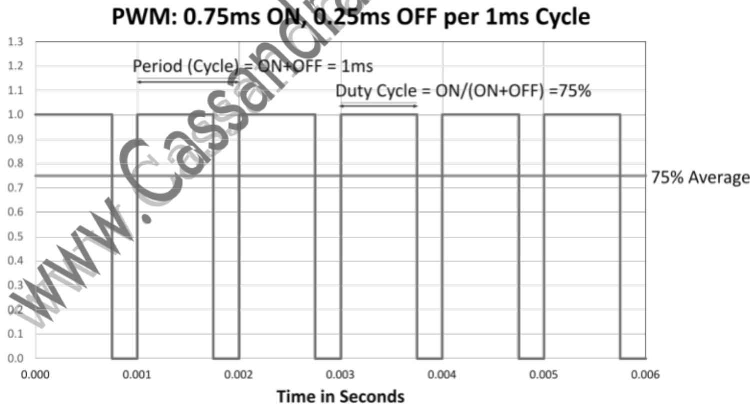


Fig 2-x. The Pico delivers an average voltage of 75% in the illustration above

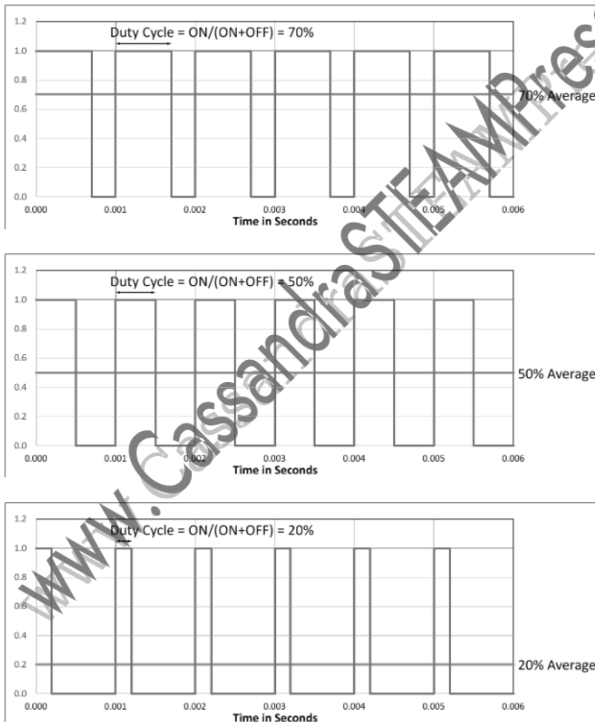
For example, if a GPIO pin which operates on the Pico at a 3.3V logic level is ON for 75% of the time (the duty cycle expressed as a percentage), then the average voltage delivered by that pin will be 2.475 V.

www.CassandraSTEAMPress.com

Copyrighted Material

$$
\begin{aligned}
Average\ Voltage &= 3.3\ V \times Duty\ Cycle \\
&= 3.3\ V \times 75\% \\
&= 2.475\ V
\end{aligned}
$$

To understand how PWM works, think of a light that you rapidly turn on and off. Each time you flip the switch, the more time the light is "on" compared to the time that it is "off", the brighter the light will seem. For example, if the light is on for exactly 3/4 of the time in any given pulse period and the cycles are fast enough, you probably won't notice the flicker. Your eyes will perceive the pulsed light to be the equivalent to that of a light getting 75% of the voltage of a light that is fully turned on. If the light is on for half of the time in each cycle, the light will appear 50% as bright. And so on.



www.CassandraSTEAMPress.com

Fig 2-x. The Pico uses duty cycles to deliver various voltages

In practice, for many embedded applications, the PWM frequency is typically somewhere between 100 Hz and 10,000 Hz (10 kHz). But for optimal results, the frequency to select ultimately depends on the electronic device you're attaching to your Pico.

Volume 2: Pulse Width Modulation and DC Motors

This book is a streamlined introduction to the complex world of DC motors in Entity Event-Driven Embedded Systems.

Using MicroPython's asyncio library and Microdot, makers, hobbyists, students and professionals can provide microservices and other data consumers / AI Agents in Event-Driven Architectures with important "real-time" information about the operating characteristics of DC motors controlled by Entity Event-Driven Embedded Systems using the Raspberry Pi Pico W or Pico 2W.

This practical and straightforward guide takes you from fading an LED with a potentiometer and Pulse Width Modulation (PWM) to full and half step sequencing of a stepper motor. It gets you started with using the Pico's GPIO pins and PWM by way of some of the most common but low-cost DC motors available today.

As you work through the step-by-step examples, you will:

- Use PWM to simulate an analog output signal on your Pico
- Discover how to position an SG90 servo precisely with a PWM signal
- Learn how to drive a TT DC Gearbox Motor with an H-bridge and PWM
- Start full and half stepping a 28BYJ-48 stepper motor with a Darlington Array
- Implement concurrency on your Entity Event-Driven Embedded Pico Systems using MicroPython's asyncio library and Microdot's cooperative multitasking approach, improving responsiveness and efficient resource usage

Who This Book Is For

The primary audience for this book are makers, hobbyists, students and professionals who want to learn more about DC motors used in Entity Event-Driven Embedded Systems.

It is assumed you know the basics of Python and have a host computer (a Raspberry Pi, Windows PC, or Apple mac), a Pico W or Pico 2W (these are available with the headers already soldered on), and some electronic components (such as a breadboard, LEDs, and inexpensive DC motors).

About the Author

Byron Mattingly has been an embedded systems software engineer and hands-on technical manager for over 20 years in regulated development in the medical devices, pharmaceutical, and avionics industries. Currently his work is focused on designing and integrating complex IT systems and training and deploying AI/ML models interacting with embedded systems. An early adopter of the Raspberry Pi platform, he is an open source contributor and avid proponent of STEAM education.

Cassandra
STEAM Press

Enjoying the sample? To purchase on Amazon go to:
https://www.amazon.com/dp/B0FXB3X4BF