

Getting Started with Entity Event-Driven Embedded Systems

Volume 1: GPIO and Asynchronous
Programming with Microdot



Raspberry Pi Pico W and Pico 2W

Cassandra
STEAM Press

Byron Mattingly

Copyrighted Material

Getting Started with Entity Event-Driven Embedded Systems

**Vol 1. GPIO and Asynchronous Programming with
Microdot: Raspberry Pi Pico W and Pico 2W**

www.CassandraSTEAMPress.com

Copyrighted Material

Getting Started with Entity Event-Driven Embedded Systems
Vol 1. GPIO and Asynchronous Programming with Microdot: Raspberry Pi Pico W and Pico 2W
by Byron Mattingly
ISBN: 979-8-89860-934-4
Copyright © 2025 Byron Mattingly.
Publisher: Cassandra STEAM Press, www.CassandraSTEAMPress.com
September 2025: First Edition

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-vol1>

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

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

Table of Contents

Preface	vii
Entity Event-Driven Embedded Pico Systems	1
<i>Learn how you can transform your Pico into an Entity Event-Driven Embedded System</i>	
1. Programming the Raspberry Pi Pico in MicroPython	21
<i>Start writing MicroPython for your Pico using Thonny (or your favorite IDE)</i>	
2. Hello, world! (from the astronauts and JSON)	55
<i>Become familiar with your Pico's computational and machine-to-machine (M2M) capabilities using JSON</i>	
3. Hello, blinky! Physical computing with the Pico	105
<i>Learn how to start connecting and controlling the physical world with your Pico's pins</i>	
4. Web API hosting with Sockets	129
<i>Discover how to transform your wireless-enabled Pico into an Entity Event-Driven Embedded System with socket programming</i>	
5. Web API Hosting with Microdot	177
<i>Transform your Pico into an Entity Event-Driven Embedded System using Microdot</i>	
6. Pelican Crossing using asyncio and Microdot	239
<i>Implement concurrency on your Entity Event-Driven Embedded Pico Systems using asyncio's and Microdot's cooperative multitasking approach, improving responsiveness and efficient resource usage</i>	
APPENDICES	
A. Raspberry Pi Pico W pinout guide	297
B. Raspberry Pi Pico W Features	299
C. Raspberry Pi Pico 2W pinout guide	301
D. Raspberry Pi Pico 2W Features	303
E. Raspberry Pi fixes Erratum E9 bug affecting the Pico 2W	305
F. Attributions and License Information	307

www.CassandraSTEAMPress.com

Preface

The Raspberry Pi Pico is fast, versatile, and energy-efficient.

This book is a streamlined but gentle introduction to the complex world of Entity Event-Driven Embedded Systems. Remarkably, the heart of such systems can be a Raspberry Pi Pico, a 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. There's a reason that the educational, maker, and hobbyist communities as well as industry have already purchased millions of these systems.

Importantly, the Pico is communications-savvy. It can communicate through its GPIO pins, built-in hardware buses (like SPI and I2C), and its onboard WiFi and Bluetooth modules, not to mention its Programmable I/O (PIO) hardware—a feature that functions as a set of small, independent state machines that can be programmed for customizing low-level interfaces.

This book builds knowledge from the ground up, offering a step-by-step approach from “Hello, world!” to highly responsive (~a few ms) interrupt driven concurrent systems using MicroPython's `asyncio` library. It gets you started with using the Pico's GPIO pins and WiFi by way of a few LEDs and asynchronous programming with `Microdot`, a remarkably small *ultra-lightweight* web server framework that's even lighter(!) than Python `Flask`.

To begin, you'll need a Raspberry Pi Pico W or Pico 2W, a few inexpensive electronic components (like a breadboard, some LEDs, a button switch, 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/>), to name just one.

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. 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'll have a very good idea of what it means to start creating your own *Entity Event-Driven Embedded Pico Systems*.

Example Code

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

<https://github.com/cassandrasteampress/event-driven-pico-code-vol1>

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

Introduction

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

Entity Event-Driven Embedded Pico Systems

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

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.

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.

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.

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.

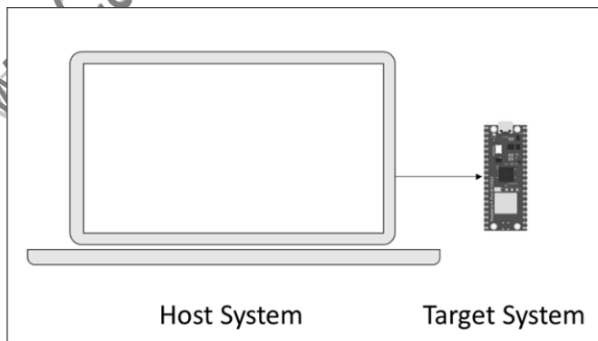


Fig 0-x Host System to Target Pico

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.

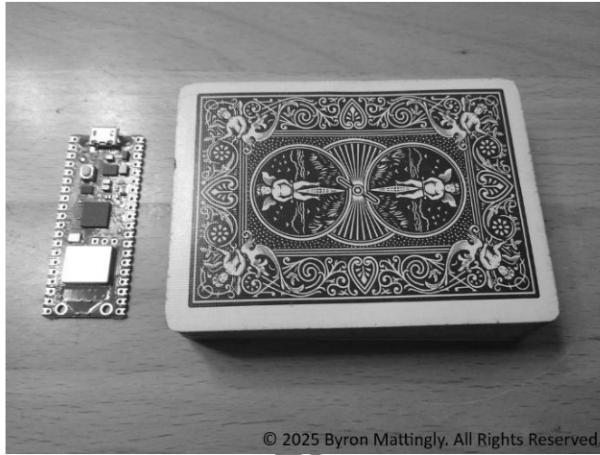


Fig 0-x. Pico W and Playing Cards

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.

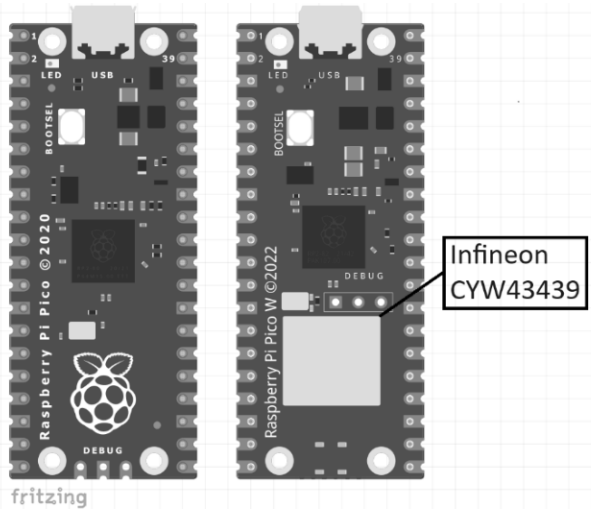
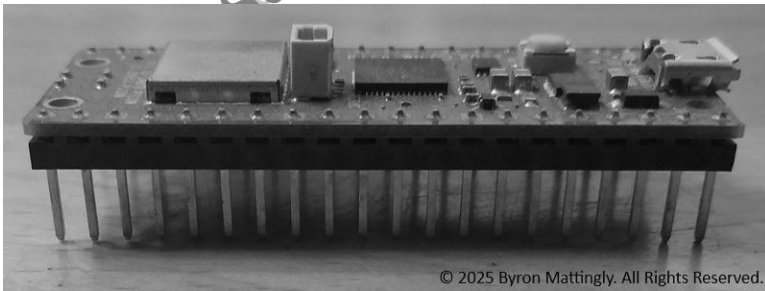


Fig 0-x. The Raspberry Pi Pico (left) and Pico W (right)

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".



© 2025 Byron Mattingly. All Rights Reserved.

Fig 0-x. Pico WH (Pico W with Headers)

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.

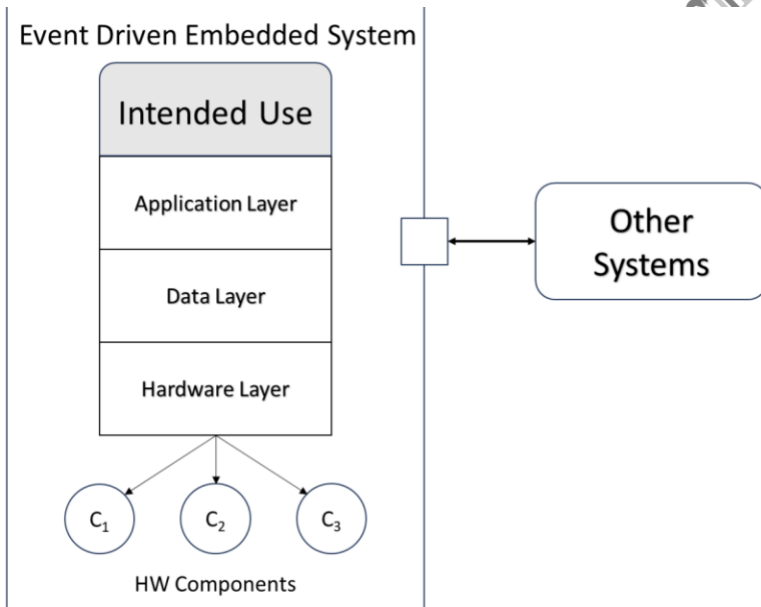


Fig 0-x. Entity Event-Driven Embedded System of (sub)systems

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.

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

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

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.

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

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 0-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

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.

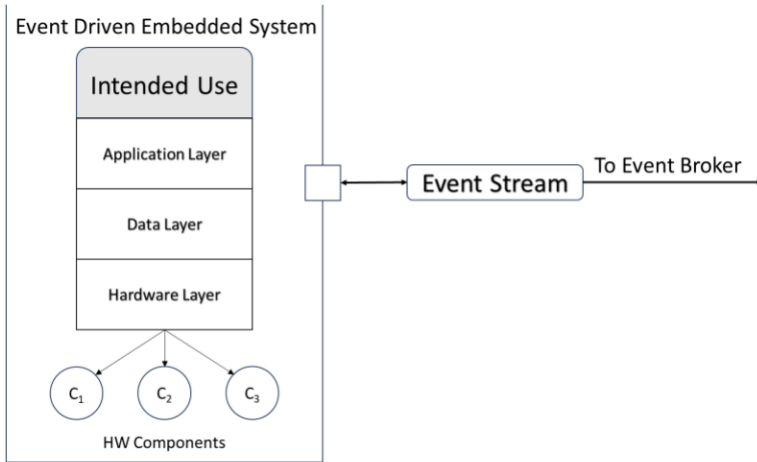


Fig 0-x. Entity Event-Driven Embedded System as an event stream source

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 0-x. Babbling sensor on a flight control surface

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)?

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.

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.

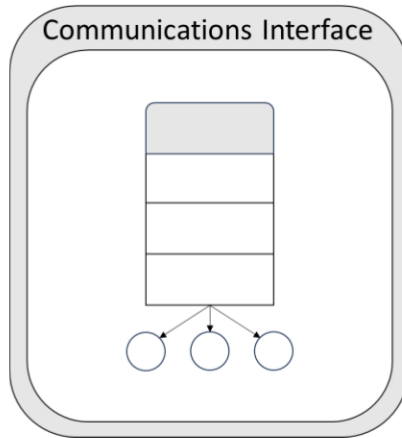


Fig 0-x. Bounded context of a single system defined by its communications interface

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:

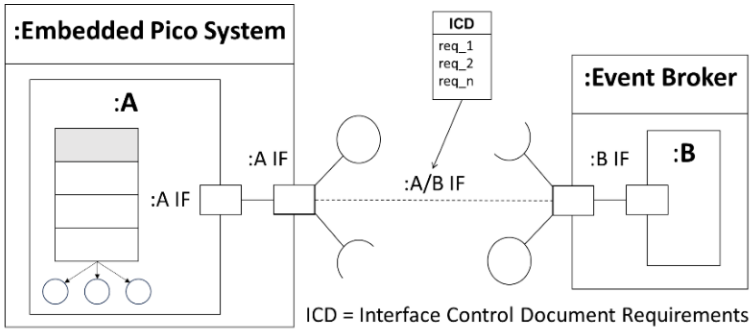


Fig 0-x. SysML Systems Engineering Model of a communications interface defining a bounded context

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).

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?

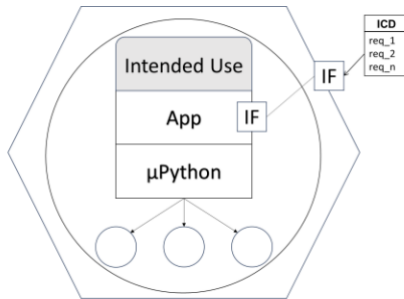


Fig 0-x. Pico embedded system using hexagonal notation

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_\(glossary\)](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).

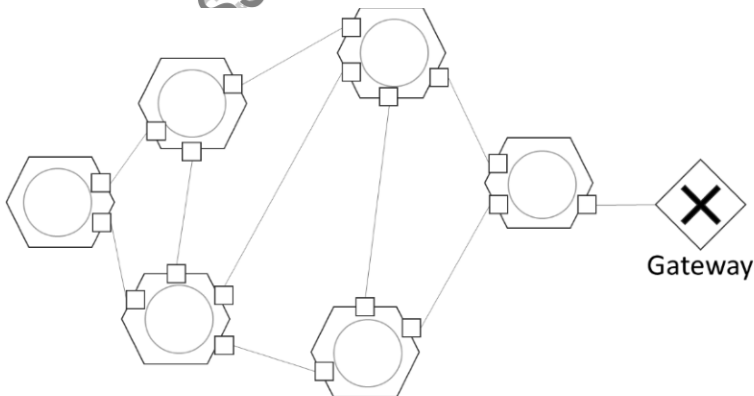


Fig 0-x. Example of Pico embedded systems in a mesh network connected to an exclusive gateway

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.

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_\(glossary\)](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

Chapter 1

1. Background
2. The Thonny IDE
 - a. What is the Thonny IDE?
 - b. Installing and Launching Thonny
 - i. Raspberry Pi
 - ii. Windows
 - iii. MacOS
 - iv. Linux
 - c. Introducing Thonny
3. Introducing MicroPython
 - a. What is MicroPython?
 - b. Installing MicroPython
4. Connecting Thonny and your Pico
 - a. Connecting your Pico to your host computer
 - b. Connecting Thonny to your Pico
 - c. Hello from your Pico!
5. Chapter Summary
6. References and Further Exploration

www.CassandraSTEMPress.com

Programming the Raspberry Pi Pico in MicroPython

Start writing MicroPython for your Pico using Thonny (or your favorite IDE)

Python is a widely used, highly expressive, yet very powerful scripting language that is relatively easy to learn. As a post-doc at Cambridge University, the Australian physicist Damien George (<https://dpgeorge.net/>) created *MicroPython* at the end of 2013 in a Kickstarter campaign (Project: 214379695).

MicroPython is an implementation of Python 3 for microcontrollers and embedded systems that is optimized to run on hardware constrained by limited resources, such as memory and processing power. It combines the simplicity and readability of Python with modules that interact directly with hardware peripherals and components.

In January 2021, MicroPython was ported to the RP2040—the microcontroller on the original Raspberry Pi Pico and Pico W. On November 1, 2024, the v1.24 release of MicroPython officially added support for the RP2350—the microcontroller on the Raspberry Pi Pico 2 and 2W.

Thonny (<https://thonny.org/>) can be installed on a host computer and MicroPython on a Pico W or Pico 2W. All of the examples in this book can run on both Pico variants. We'll simply refer to the Pico W or Pico 2W as just “Pico” and note if there are differences between the two variants to consider along the way.

There are many popular IDEs (Integrated Development Environments) available that can be employed for writing and running MicroPython programs on your Pico. Some of these IDEs are very rich in features and can handle lots of different programming languages at the same time. If you have a favorite IDE, it will likely work for most, if not all, of this book's examples without any issues.

Thonny is a beginner-friendly IDE, however, that focuses on a single language—Python. With its built-in tools support for MicroPython (since v3.2), it is particularly well-suited for MicroPython development on the Pico. Its clean, uncluttered design and integrated features make it straightforward to navigate and understand for those new to developing and debugging MicroPython code. Thonny is freely available for multiple platforms and is officially supported by the Raspberry Pi Foundation. It is included by default in the Raspberry Pi OS. We'll use Thonny for presenting the code examples throughout this book.

The Thonny IDE

Using Thonny, you can run Python code on your host computer and MicroPython code on your target Pico. As previously mentioned, Thonny is an IDE for programming Python and MicroPython that currently comes pre-installed with the Raspberry Pi OS. It also readily installs on host platforms like a Windows PC or an Apple Mac. Thonny has a straightforward and intuitive GUI (graphical user interface) that is relatively easy to use.

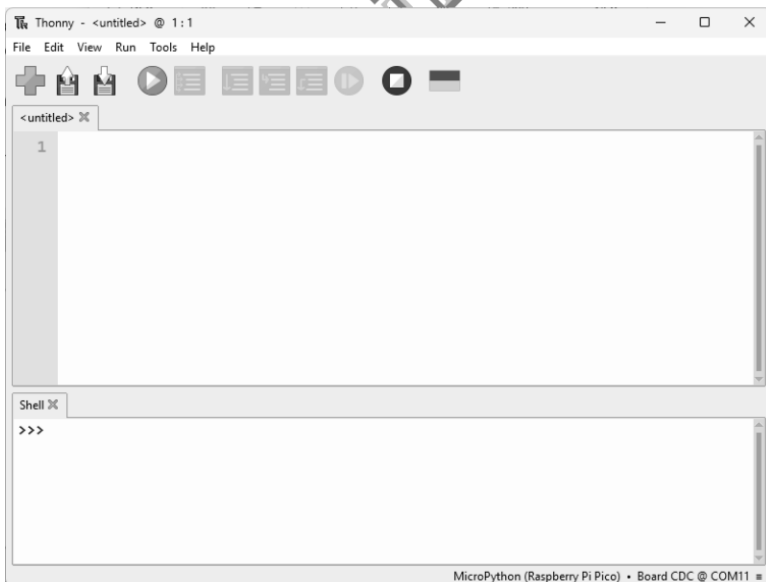


Fig 1-x. The Thonny IDE in regular mode

The Thonny IDE collects all the tools you need to write and develop MicroPython code for the Pico into a single user interface. Thonny even allows you to run code interactively in its Python Shell which makes it easy to debug code line by line.

Thonny's user interface consists of a menu toolbar along the top, an editor window where you write and debug programs and scripts, and a Python Shell where you run individual Python commands and view output in interactive REPL (Read, Evaluate, Print, and Loop) sessions.

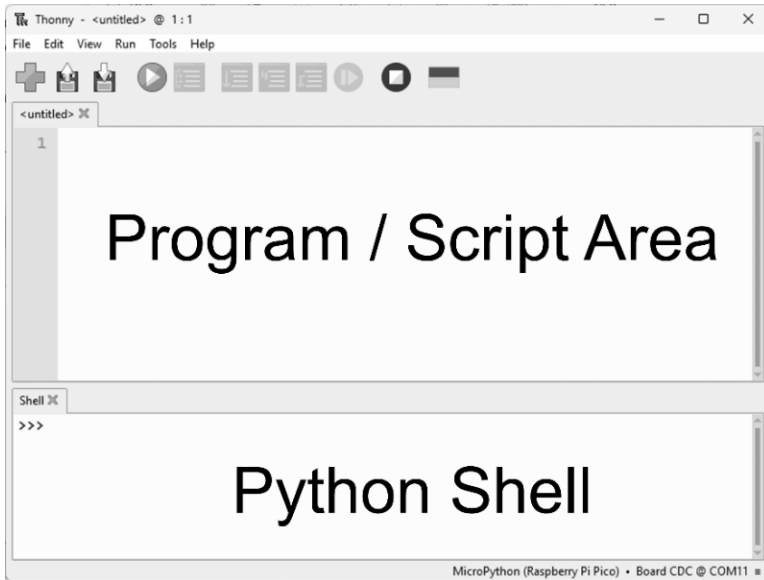


Fig 1-x. The Thonny IDE layout

The drop-down menus in the toolbar running along the top edge of Thonny's IDE allow you to create new programs, save them, load them, and run them. The Program / Script Area directly underneath the toolbar is where you write and edit your programs. The Python Shell below the Program / Script Area allows you to type individual Python commands and run them one at a time by pressing the Enter key after each line. The Python Shell is a great way to try things out without having to run a whole program. It also provides information about a running program.

With Thonny, you are able to select which Python interpreter to run your code—Thonny's Local Python 3 interpreter on your host computer or the MicroPython one installed on your target Pico. Moreover, your programs and scripts can be stored as Python modules (text files) either on your host system, or on your Pico, or both. Thonny executes the lines of code in a Python module from top to bottom in the same folder

that it finds the module, just as if you typed those lines at a prompt in the Python shell.

Thonny has two main modes, a *Simple Mode* and a *Regular Mode*. The Simple Mode interface uses friendly icons that are labelled but the Regular Mode is almost as friendly and a little easier to navigate most of the examples in this book because it also includes a menu bar. Thonny also has an Advanced Mode which can be especially useful for debugging large, complicated programs. But for our purposes, we'll stick with Regular Mode since it has everything we need for this book.

If you find yourself in Simple Mode, you can easily change to Thonny's Regular Mode by clicking on "Switch to regular mode" link in the upper right hand corner of the IDE and relaunch Thonny.

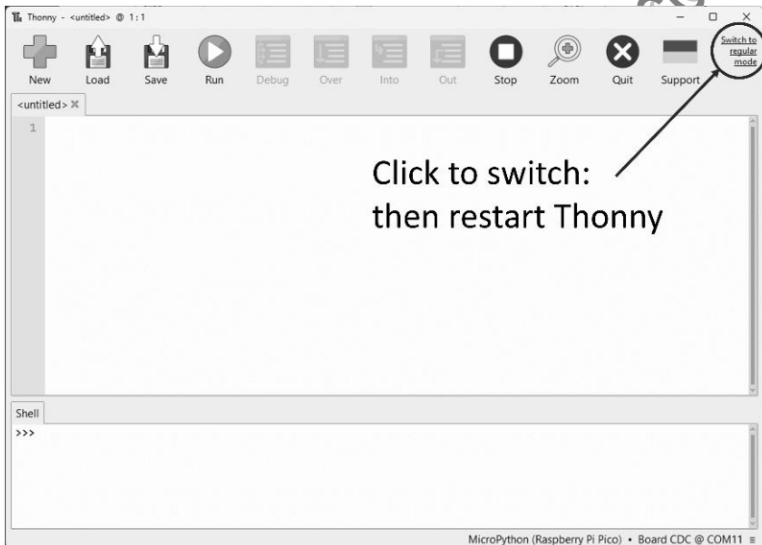


Fig 1-x. The Thonny IDE in simple mode

Installing and Launching Thonny

Thonny can be downloaded and installed on a recent Windows PC, Apple Mac, or Linux computer; Thonny already comes pre-installed on the Raspberry Pi OS!

From the Thonny homepage, download the most recent version of Thonny and run the appropriate installer for your host computer, following the installation wizard.



Copyrighted Material

Volume 1: GPIO and Asynchronous Programming with Microdot

This book is a streamlined but gentle introduction to the world of Entity Event-Driven Embedded Systems.

Using MicroPython, the Raspberry Pi Pico W and Pico 2W, makers, hobbyists, students and professionals can complete projects which are capable of participating in Event-Driven Architectures. This practical step-by-step guide builds knowledge on creating Entity Event-Driven Embedded Pico Systems using MicroPython's `asyncio` library and Microdot taking you from "Hello, world!" to interrupt driven concurrent systems. It gets you started with using the Pico's GPIO pins and WiFi by way of a few LEDs and asynchronous programming with Microdot, a remarkably small *ultra-lightweight* web server framework that's even lighter(!) than Python Flask.

As you work through the examples in this guide, you will:

- Start writing MicroPython for your Pico using Thonny (or your favorite IDE)
- Become familiar with your Pico's computational and machine-to-machine (M2M) capabilities using JSON
- Learn how to start connecting and controlling the physical world with your Pico's pins
- Discover how to transform your wireless-enabled Pico into an Entity Event-Driven Embedded System with socket programming
- Transform your Pico into an Entity Event-Driven Embedded System using Microdot
- Implement concurrency on your Entity Event-Driven Embedded Pico Systems using `asyncio`'s 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 Entity Event-Driven Embedded Systems using MicroPython's `asyncio` library and Microdot.

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, a button switch and Piezo buzzer).

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/B0FVXVPCTV>