Prototyping Embedded IoT Devices

Introduction to Prototyping in IoT

- **Definition**: Prototyping embedded IoT devices means building early, testable versions of both hardware (sensors/actuators/boards) and software (firmware/UI/cloud hooks).
- **Purpose**
  - Verify feasibility and surface risks early.
  - Compare alternatives quickly (set-based exploration).
  - Communicate concepts to stakeholders and gather feedback.
- **Scope**: Includes **interfaces** (on-device UI, web/mobile dashboards) and **devices** (sensor nodes, bridges, edge gateways).

Decision-Making with the Pugh Matrix

What It Is

- A **criteria-based matrix** for comparing alternatives against a baseline with weighted criteria. Common in **lean**, **Six Sigma**, and **set-based design**.

Why It Matters

- Replaces preferences with **transparent, weighted trade-offs**.
- Ideal for thorny choices (protocols, sensors, boards, OS/RTOS).

How It Works

1. List **alternatives** and choose a **baseline** (if none, select one option to act as baseline).
2. Define **criteria** (power, latency, cost, tooling, security, ecosystem).
3. **Weight** criteria by importance.
4. Score each alternative vs. baseline (**+, 0, –** or numeric such as **1/3/9**).
5. **Sum** scores, review results, iterate if something feels off (tweak weights, revisit criteria)
   .

Example Uses

- **IoT protocol**: MQTT vs. CoAP vs. AMQP (throughput, QoS, tooling, bandwidth).
- **Sensor selection**: accuracy, power, size, range, unit cost.
- **OS/RTOS selection**: determinism, memory footprint, drivers, cloud libraries.

Benefits

- Faster convergence, less bias, clearer stakeholder communication.
- Prunes **dominated alternatives** early.

Set-Based vs. Point-Based Design

Point-Based

- Pick one solution → analyze → iterate. Risk: late-stage surprises force costly restarts.

Set-Based

- Keep **multiple options** open early; run **quick experiments** to eliminate weak candidates.
- Naturally pairs with a **Pugh Matrix** to narrow the set.

Why It Fits IoT

- Requirements are often fuzzy (e.g., protocol not specified).
- Protects schedule/budget by avoiding premature lock-in.

Prototype Platforms for Embedded Devices

Why Platforms Matter

- Board choice affects **speed**, **flexibility**, **I/O**, and **cost**. Use a Pugh Matrix to weigh criteria.

Raspberry Pi Family

- **Strengths**: Low cost, huge ecosystem, great for UI + cloud + protocol tests.
- **Models**: Pi 3 B+ (balanced), Pi 4 (more RAM/CPU), A+ (smaller), **Zero W** (tiny, Wi-Fi), **Compute Module** (for custom carrier boards).
- **HATs**: PoE, ADCs, sensors, motor drivers—snap-on expandability.

BeagleBone Family

- **Black / Green** (with Grove connectors): rich I/O, good for I/O-heavy control.

Python-Forward Boards

- **MicroPython / CircuitPython** devices (e.g., **BBC micro:bit**, **ESP32 boards**) let you write Python "near bare-metal" for fast firmware iteration.

Arduino Microcontrollers

- **Uno/Mega** and many variants; vast shield/sensor ecosystem. Cross-compile from PC → MCU. Great for low-power sensor nodes.

Chip Vendor Dev Kits

- Example: **Silicon Labs Blue Gecko** for Bluetooth—ships with examples, stacks, and tools. Combine with an SBC (Pi) for hybrid prototypes.

Specialized / Maker Boards

- **AWS IoT 1-Click**-style devices, LTE-M modules, tiny **Adafruit GEMMA/Trinket** for wearables. Maker vendors (Adafruit, SparkFun) accelerate PoC.

Custom PCBs

- Move to custom boards when off-the-shelf can't meet constraints. Until then, **don't reinvent**—validate function first.

Quick Selection Heuristics

- **Pi**: UI + Linux + network protos quickly.
- **BeagleBone**: I/O-heavy control.
- **Arduino**: Simple, battery-friendly sensing/actuation.
- **ESP32/MicroPython**: Fast firmware iteration + Wi-Fi/BLE.
- **Vendor kits**: Targeting a specific silicon/stack.

Operating Systems for Embedded Development

Why OS Choice Matters

- Impacts **determinism**, **latency**, **memory/flash footprint**, **driver availability**, **security**, **tooling**, and **time-to-prototype**.

Bare-Metal Options

- **C/C++ loop**: Maximum control; minimal overhead. Great for ultra-simple, ultra-fast tasks. Trade-off: you own drivers/comms/scheduling.
- **MicroPython / CircuitPython**: Python near bare-metal on supported MCUs. Rapid iteration; limited boards/perf trade-offs.

Real-Time Operating Systems (RTOS)

- **Why RTOS**: Deterministic scheduling; bounded worst-case latency; priorities; timers; queues; tiny footprint.
- **When to use**: Hard/firm real-time, lightweight multitasking without Linux, tighter power/boot constraints.
- **Selection**: Use a Pugh Matrix—criteria: determinism, BSPs, TLS/crypto, cloud libs, debugger support, community, license.

OS on Single-Board Computers (Raspberry Pi Focus)

- **Raspberry Pi OS** (Debian-based): fast path to working prototypes, excellent tooling/docs.
- **Other Pi options**: Kali (sec/pen-test), Nard SDK (lean, long-running), LibreELEC/OSMC (media), Ubuntu Core/MATE, **ChibiOS** (RTOS), Windows 10 IoT Core, RISC OS.

Embedded Linux (beyond Pi)

- Mature networking, filesystems, crypto; huge community; generally royalty-free (track component licenses).
- Vendor-supplied distros for their SoCs often minimize bring-up.

Building a Custom Linux (Yocto)

- **Yocto Project**: layered recipes for a minimal, tailored image.
- **Pros**: control footprint/services/licenses; reproducible.
- **Cons**: learning curve—best as fidelity approaches production.

Android for Embedded

- Strong UI toolkit; Linux kernel foundation; good for display-centric devices (kiosks/HMIs/wearables). Consider update/MDM approaches.

OS/RTOS Selection Criteria (for a Pugh Matrix)

- **Functional**: real-time, drivers, networking, security/TLS, OTA, storage/FS.
- **Non-functional**: latency bounds, footprint, boot time, power modes, reliability.
- **Dev Experience**: toolchains/IDEs, tracing, logging, unit tests, CI/CD, docs/community.
- **Ecosystem**: cloud SDKs (AWS/Azure), BSP maturity, maintenance horizon, licensing.

- **Delivery**: time-to-prototype now vs. production later (migrate Pi/Linux → RTOS/Yocto as needed).

FreeRTOS as a Recommended Platform

Why FreeRTOS?

- Many prototypes don't need full Windows/Linux/Android; they need **control over performance and resources** with **preemptive multitasking** and small footprint.
- **FreeRTOS** is a portable, open-source RTOS designed for small embedded systems—ideal for **rapid prototyping** and scalable into production.
- Since **2017**, stewardship moved to **Amazon** (original author: **Richard Barry**, 2003). It remains open source, with optional **AWS-focused extensions**.

Licensing, Quality, and Tooling

- **License**: MIT (open source).
- **Standards**: Conforms to most **MISRA** C guidelines.
- **Compilers**: Works with **GCC** and major commercial compilers.
- **Architectures**: Broad support (ARM variants, PIC, x86, and others).

Variants

- **OpenRTOS**: Commercial build with warranty/support.
- **SAFERTOS**: Safety-certifiable derivative for regulated domains (medical/industrial/automotive).
  Note: These two are **not** open source.

Architecture & Build Model

- **Kernel core** is intentionally small—famously centered around a few core files (`tasks.c`, `queue.c`, `list.c`) plus headers.
- **Port layer** adapts to your MCU/ISA and compiler.
- **Dynamic memory**: multiple heap implementations; you can provide a custom allocator if needed.
- **App-bundled OS**: The kernel is **linked into your application**, producing **one executable image** that contains both your code and FreeRTOS.

Tasks, Scheduling, and Power

- **Single process** model with **multiple tasks/threads** sharing one address space.
- States: **Running**, **Ready**, **Blocked**, **Suspended**.
- **Preemption/time slicing** is configurable (priorities, tick rate).
- **Low power**: Idle task **hook** allows entry into MCU low-power modes; you can add custom idle behavior to hit power targets.

Memory Protection & Responsibility Boundaries

- Tasks **share memory**; simple IPC but **limited isolation**. On MCUs with **MPU**, you can add **some** protection—on smaller MCUs this may not exist.
- Your application **starts the scheduler** (e.g., `vTaskStartScheduler()`), then the kernel orchestrates task execution.

IPC, Sync, Timers, and Debug

- **IPC & Sync**: Central **queue** primitive underlies semaphores, mutexes, event mechanisms—easy to learn and consistent.
- **Timers**: Tick-driven software timers; schedule **relative or absolute** delays for tasks and events.
- **Debugging**: Stack overflow **hooks**, **trace macros** (empty by default) you can instrument, plus ecosystem tooling for task awareness.

FreeRTOS+ Middleware & Extensions

- Optional libraries commonly available in ports:
  - **Filesystem** support
  - **Networking stacks**
  - **Command-line interfaces**
  - **I/O frameworks**
- Choose only what you need to keep the image lean; footprint scales with features selected.

AWS Integrations (Key for IoT Prototyping)

- **AWS IoT Device SDK integrations**: MQTT, HTTP/HTTPS (TLS), secure comms.
- **Device Shadows**: Synchronize desired/reported state even when device is offline.
- **Device Defender**: Security posture and auditing aids.
- **AWS Greengrass**: Run local Lambdas and ML inference on edge devices.
- **Windows-based device simulator**: Prototype logic against AWS without hardware.
- These integrations make **end-to-end cloud demos** possible very quickly.

Practical Pros/Cons for Prototyping

- **Pros**
  - Tiny, portable, deterministic; easy task model and queues.
  - Scales from PoC to production without a platform leap.
  - Strong docs, tutorials, community, and example projects.
- **Cons / Caveats**
  - **Shared address space** means less isolation—coding discipline matters.
  - You own **Board Support Packages** and drivers unless provided by vendor.
  - Needs C/C++ embedded comfort (vs. Python-on-Linux ease).

When to Prefer FreeRTOS (vs. Linux/MicroPython)

- **Choose FreeRTOS** when you need:
  - Deterministic timing/control loops, fast boot, small footprint, tight power budgets.
  - Secure cloud connectivity on constrained MCUs.
- **Choose Linux/MicroPython** when you need:
  - Fast UI work, full OS services, rich networking and storage, rapid experimentation without strict real-time constraints.

Next Steps with FreeRTOS

- Set up: choose board/port, select heap implementation, configure `FreeRTOSConfig.h` (tick rate, priorities, hooks).

- Start with **two or three tasks** (sensor, comms, control), add **queues** for message passing, then layer in **timers**.
- Add FreeRTOS+ / AWS libraries only as needed; measure **stack usage** and **tick load** early.

Key takeaways

- Use **set-based design** + **Pugh Matrix** to make defensible choices on protocols, sensors, boards, and OS/RTOS.
- Start with **platforms/OSes that speed learning** (Pi/Linux, MicroPython, Arduino), then taper toward production (RTOS/Yocto/custom PCB).
- **FreeRTOS**: small, deterministic, battle-tested; AWS integrations make it a powerful path for cloud-connected embedded prototypes.
- **Support**: Ask early—faster iteration beats late, expensive pivots.