# 365-Day Embedded Systems Learning Plan

**Goal:** Prepare for the Garmin **Embedded Systems Software Engineer (Fitness Products)** role, which requires strong C/C++ skills, embedded RTOS experience, ARM architecture knowledge, and familiarity with tools like logic analyzers and JTAG[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge). This plan is divided into phases, ensuring we **separate C and C++ learning** (not mixing them) as requested. Each day provides ~2 hours of focused learning or practice. The plan emphasizes hands-on projects because practical experience is the most effective way to master embedded systems[[2]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=%E2%9A%92%EF%B8%8F%20Projects). (You are fluent in English and Chinese, so resources in both languages can be utilized as needed.)

## Phase 1: C Programming Foundations (Days 1–90)

**Focus:** Build a **strong C language foundation** and basic embedded programming skills. C is the *backbone* of embedded systems[[3]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=C%20Programming), and industry experts recommend mastering C thoroughly *before* moving to C++[[4]](https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/#:~:text=C%2B%2B%20is%20a%20useful%20skill,a%20mandatory%20skill%20in%20embedded). Many embedded SDKs and drivers are written in C[[5]](https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/#:~:text=Image%3A%20Profile%20Badge%20for%20the,Commenter), so these 3 months will ensure you are confident with C syntax, pointers, memory, and using C on microcontrollers (STM32, Arduino, etc.). You will also begin using your hardware (STM32, ESP32, Arduino) for simple projects to solidify concepts.

* **~~Day 1~~:** **Setup Development Environment:** Install or update STM32CubeIDE (for STM32 development) and VS Code (for general editing). Ensure you have the toolchains for C (gcc for ARM, etc.) ready. If using VS Code for embedded debugging, install the Cortex-Debug extension[[6]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=%E2%80%8D%20VS%20Code%20Extensions). Verify your STM32 board connects to the PC (install ST-Link drivers if needed).
* **~~Day 2~~:** **“Blink LED” on STM32:** Create a new STM32CubeIDE C project (use STM32CubeMX if needed to configure pins). Write a simple *blinky* program in C to toggle an on-board LED with a delay. This tests your toolchain and gives an initial hands-on success. (If using an Arduino board, do the equivalent using the Arduino C environment).
* **Day 3:** **C Syntax Review – Basics:** Refresh basic C syntax: data types (int, float, etc.), variables, operators, control structures (if/else, loops). Write a small C program on your PC (or an Arduino) that uses these basics (e.g. a program that prints numbers 1–10 or blinks LED in a pattern). Ensure you understand block structure and main function.
* **Day 4:** **Functions and Modular C:** Learn to write and use functions in C (return values, parameters) and how to split code into multiple files. Refactor your Day 3 code by moving LED toggle or print functionality into a separate function. Practice compiling and linking multiple C files.
* **Day 5:** **Pointers I – Basics:** Study what pointers are and how to use them (declaring pointers, dereferencing, pointer arithmetic). Use a pointer to manipulate a variable in a small C program. For example, create an integer variable and a pointer to it, then modify the value via the pointer. This concept is critical for low-level programming (e.g., accessing memory-mapped registers).
* **Day 6:** **Pointers II – Arrays and Strings:** Understand relationship between arrays and pointers. Practice iterating through an array using pointer arithmetic. Learn about C-strings (null-terminated character arrays) and how to manipulate them. Write a C function that takes a char array (string) and counts length, or concatenates two strings (use strcpy/strcat or manual loop).
* **Day 7:** **Structures and Unions:** Learn how to define struct types in C to group data (for example, define a struct for a GPS coordinate with latitude, longitude, etc.). Write a program that populates a struct and prints its fields. Understand memory layout of structs and what padding is. Also learn about union (e.g., a union to access a 32-bit value and its individual bytes). These are useful for interpreting hardware registers or data packets.
* **Day 8:** **Bit Manipulation:** Practice using bitwise operators (&, |, ^, ~, shifts). These are essential for embedded programming (e.g., setting or clearing specific bits in a register). Create a C program that uses bit masks to set, clear, and toggle bits of a byte. For example, implement functions SET\_BIT(x,n), CLEAR\_BIT(x,n), TOGGLE\_BIT(x,n) using bitwise ops, and test them.
* **Day 9:** **Memory Segments & Storage Classes:** Read about how a C program’s memory is organized (stack vs heap, global/static vs local). Understand storage classes (auto, static, extern) and their meaning. Write a small test: define a static local variable in a function and observe its persistence between calls; define global vs local variables and see their scope. This builds understanding of how embedded memory is managed.
* **Day 10:** **Compile and Flash Process:** Learn the steps from C source to running on hardware: preprocessing, compilation, linking, and loading to flash. Use a simple C program and enable verbose compile logs to see gcc commands. On STM32CubeIDE, observe the generated *.elf*, *.hex* files. No coding today – focus on understanding the toolchain.
* **Day 11:** **GPIO Basics on STM32:** Dive into microcontroller specifics. Read the STM32’s GPIO chapter in its reference manual/datasheet (to understand registers like MODER, ODR, IDR). Then write a C routine to control an LED by writing to the GPIO registers directly (instead of using HAL library). This low-level exercise strengthens understanding of embedded C by manipulating hardware registers (this is *low-level programming* interacting with hardware[[7]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Low)).
* **Day 12:** **Inputs and Outputs:** Using STM32 or Arduino, write a program that reads a digital input (e.g., a push button) and controls an output (LED) based on it. If your board has a user button, use it; otherwise, wire a button to a GPIO. Learn to *debounce* a button in software (simple delay or state check). This teaches handling real-world signals and basic conditionals in embedded context.
* **Day 13:** **Timers and Delays:** Learn how to use a hardware timer on the microcontroller. Configure a timer (using STM32 HAL or registers) to generate periodic interrupts or events. Modify the Day 2 blinky to use a timer interrupt for toggling the LED instead of HAL\_Delay or software loop. This introduces **interrupts**, which are crucial in embedded systems for real-time behavior.
* **Day 14:** **Interrupts in C (Microcontroller):** Study what interrupts are and how ISR (Interrupt Service Routines) are written in C. On STM32, identify vector table and how an ISR is named (e.g., void TIM2\_IRQHandler(void) for Timer2). Extend Day 13: implement the timer’s ISR to toggle LED, and use volatile variables if sharing data between ISR and main code (learn why volatile is needed for shared flags).
* **Day 15:** **UART Communication:** Learn how serial communication works (UART protocol basics). Use STM32 UART (or Arduino Serial) to print debug messages from your microcontroller to your PC. For example, send a "Hello, world" or sensor readings over UART and view it in a serial console (e.g., PuTTY or Arduino Serial Monitor). Being able to use UART for debugging will be very useful in later projects.
* **Day 16:** **C Debugging Basics:** Practice using a debugger with C on the microcontroller. Set breakpoints in your code (e.g., in the button or timer ISR from prior tasks) using STM32CubeIDE’s debugger (which uses JTAG/SWD through the ST-Link). Learn to step through code, inspect variables, and modify them on the fly. This builds familiarity with **JTAG-based debugging**, a critical skill for embedded engineers (JTAG ICE allows you to inspect and control a running system).
* **Day 17:** **C Standard Libraries & printf:** Understand how to use the C standard library in embedded context. Try using printf to send formatted output over UART (you might need to retarget printf to UART on STM32 – many tutorials available). Learn how sprintf can format strings. Also be aware of the cost of printf (large code size due to %f, etc.) – often an interview point.
* **Day 18:** **Makefiles & Build Process (Optional):** (For additional insight) Write a simple Makefile for one of your C projects (perhaps the Day 3 or Day 5 PC program) to compile it via command line. This will demystify build systems, which is useful if you ever need to compile outside an IDE. If unfamiliar, follow a short Makefile tutorial.
* **Day 19:** **C Programming** Quiz**:** Take a short quiz or solve problems to reinforce C basics (data types, pointers, etc.). You can use online resources or textbooks exercises. For example, ensure you can answer: *What is a dangling pointer? How do arrays and pointers relate? What does const mean in different positions?* This helps identify any weak areas to revisit.
* **Day 20:** **Project: LED Pattern Generator:** Build a small project that combines basics. For instance, have 2–3 LEDs (or one LED with different blink patterns) and use a button to cycle through modes (e.g., steady on, blinking, SOS pattern, off). Implement state changes in C using a switch or state variable. This project uses control structures, functions, and I/O together.
* **Day 21:** **Memory Pointers & Data Structures:** Implement a simple data structure in C to deepen pointer skills. For example, create a linked list of numbers: define a struct Node with a pointer to next node, and write append, remove, print functions. Manage memory with malloc and free for nodes. This teaches dynamic memory allocation in C. (Be cautious with malloc on microcontrollers with limited heap; for learning it’s fine, but embedded systems often avoid heap allocation in production.)
* **Day 22:** **Function Pointers and Callbacks:** Learn about function pointers in C – an advanced but important feature (used in driver APIs and callback systems). Write a C program that uses a function pointer to call different functions (e.g., have an array of function pointers for different math operations). In embedded context, this is useful for things like HAL library callback handlers.
* **Day 23:** **Embedded C: Reading Sensor (I²C) – Theory:** Even if you haven’t used I²C hardware yet, read how an I²C protocol works and how to interact with a simple sensor. If you have a sensor module (e.g., a temperature sensor or accelerometer that uses I²C), plan to use it. Read its datasheet to understand what registers to read/write. No coding yet – focus on understanding the protocol (start/stop bits, addresses, etc.) and how to use MCU I²C peripherals.
* **Day 24:** **Embedded C: Reading Sensor (I²C) – Practice:** Implement I²C communication on your microcontroller. Using STM32CubeIDE, enable I²C and attempt to read real data from a sensor (if you have one, like an accelerometer or EEPROM). If you don’t have a physical I²C device, simulate by reading/writing to registers of a device in code (pretend values). The goal is to familiarize with using an MCU peripheral from C code. (This covers **communication protocols** knowledge like I²C, which is essential for embedded engineers[[8]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Communication%20Protocols).)
* **Day 25:** **Project: Temperature Logger (Simple):** If you have a temperature sensor (or use a built-in MCU temperature sensor if available), write a program to read temperature periodically (say every second using a timer) and send the readings over UART or store in an array. This combines timer interrupts, sensor I/O, and data handling in C. If no actual sensor, use a dummy sine-wave function to simulate temperature changes.
* **Day 26:** **Review Computer Architecture Basics:** Before diving deeper, spend time on basic computer architecture concepts: how a CPU executes instructions, what registers and memory are, what a bus is. Specifically, review how an ARM Cortex-M (used in STM32) is structured (core registers, program counter, etc.). This theoretical background will prepare you for later low-level work[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge).
* **Day 27:** **ARM Cortex-M and CMSIS:** Learn about CMSIS (the Cortex Microcontroller Software Interface Standard) provided by ARM – it gives a standard way to access CPU registers and handle interrupts. Write a small piece of code using CMSIS to toggle an LED (e.g., GPIOA->ODR ^= 0x1 style) without using the vendor HAL. This reinforces understanding of the MCU at the register level and how C code maps to hardware.
* **Day 28:** **Embedded C Best Practices:** Read about coding standards and best practices for embedded C (e.g., the Barr Group’s Embedded C Coding Standard[[9]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=,Embedded%20C%20Coding%20Standard)). Focus on topics like: avoiding recursion, using volatile for hardware registers, using enum for states instead of #define constants, etc. Apply one or two practices to your existing code (e.g., replace magic numbers with defined constants, improve code clarity).
* **Day 29:** **Memory Debugging:** Learn to use debugging tools to inspect memory on the microcontroller. In STM32CubeIDE, during a debug session, view memory at a certain address (e.g., examine the memory mapped peripheral registers or an array in SRAM). Practice interpreting the values (for instance, see your linked list nodes in memory). This helps build skill in **debugging and testing**, which is fundamental for embedded engineers[[10]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing).
* **Day 30:** **C Standard Corner Cases:** Review some tricky aspects of C that often trip up developers: integer promotion rules, use of sizeof, pointer vs array differences, shallow vs deep copy, etc. Write quick experiments to verify your understanding (e.g., what is the result of sizeof(pointer) vs sizeof(array) when passed to a function). This solidifies your mastery of C quirks.
* **Day 31:** **Project: UART Command Parser:** Create a project where the STM32 (or any board) receives simple text commands over UART and performs actions. For example, send commands like "LEDON", "LEDOFF" from your PC terminal and have the microcontroller turn an LED on or off accordingly. Parse the input string in C and use strcmp or similar. This enhances string handling and I/O in C, and simulates a basic device command interface.
* **Day 32:** **Microcontroller Data Sheets:** Pick a section of the STM32 microcontroller datasheet or reference manual (e.g., the ADC or SPI section). Spend the day reading and summarizing how it works and what configuration steps are needed in C to use it. Practice reading those dense technical documents – an embedded engineer must be comfortable extracting key info from datasheets.
* **Day 33:** **ADC Basics and DMA (Conceptual):** Read about Analog-to-Digital Converters and how to use them (perhaps in your STM32). Also introduce the concept of DMA (Direct Memory Access) which can transfer data without CPU intervention. You don’t need to implement today, but understand how you would configure an ADC to take a sensor reading and perhaps use DMA to store continuous samples.
* **Day 34:** **ADC Practice:** If your board has an ADC and a variable analog source (potentiometer or a sensor), write a C program to read an analog value via ADC and print it to serial. If no hardware, use STM32 internal temperature sensor or just simulate by generating a pseudo-random number. Get familiar with initializing ADC (you can use CubeMX to generate init code) and triggering conversions.
* **Day 35:** **Polling vs Interrupt vs DMA:** Modify your ADC reading program to use interrupts (i.e., end-of-conversion interrupt) instead of polling for completion. If time permits, try setting up DMA for ADC (so it continuously samples and stores data in a buffer). Compare the approaches (polling vs interrupt vs DMA) in terms of CPU usage. Understanding these mechanisms is key for efficient embedded system design.
* **Day 36:** **Free Day / Catch-up:** Use today to catch up on any previous topic you struggled with or couldn’t complete fully. If all caught up, use the time to organize your code projects, comment your code, or even write a short diary of what you’ve learned so far in C. This helps retention and also gives you material to refer back to.
* **Day 37:** **C++ Preparation – Differences from C:** Before diving into C++, list out what you know will be different or new in C++ (classes, new/delete, references, etc.). Write down any questions you have (e.g., “How does C++ manage memory vs C?”, “What is OOP really used for in embedded?”). This will set a purpose for your C++ learning phase. No coding – just a mindful comparison and perhaps watch a short overview video of C vs C++ features.
* **Day 38:** **Firmware Structure Reading:** Browse through a simple open-source embedded C project’s code to see how things are structured. For example, look at the FreeRTOS *demo* applications or a device driver in an SDK. Don’t worry if you don’t fully grasp it; try to identify use of function pointers, ISR declarations, etc. Seeing real-world C code (beyond our small exercises) gives context on what professional firmware looks like.
* **Day 39:** **Language Practice:** Do some practice problems in C to ensure mastery of logic and problem-solving (since job interviews may still ask you to solve basic coding problems). For instance, write a function to reverse a string in place, or to find the largest element of an array. If you want, attempt them on an online judge or use a book like *C Programming Language* exercises. The aim is to be comfortable coding algorithms in C without resorting to Python (since you mentioned using Python in internship, we want to solidify doing these in C).
* **Day 40:** **Version Control with Git:** Ensure you are using Git for version control on your projects (if not, start today). Initialize a Git repo for one of your projects (e.g., the UART parser or logger) and commit your code. Learn basic Git commands (add, commit, push). This is a vital software skill for any engineer. You might create a private GitHub repository to track your progress.
* **Day 41:** **Debugging Practice – Fault Injection:** Write a buggy C program on purpose and then debug it. For example, induce a common error like an *off-by-one* error in an array loop or a *null pointer dereference*. See how the program behaves on hardware or in a debugger (it might hard fault). Practice using the debugger to find the root cause (inspect registers or use breakpoints to narrow it down). This builds skill in troubleshooting, which employers highly value[[10]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing).
* **Day 42:** **Embedded C Quiz/Review:** Take time to evaluate how far you’ve come with C. Attempt an **Embedded C online test** if available, focusing on embedded-specific C topics: I/O addressing, bit operations, cross-compiling, etc. (For example, an online Embedded C test might cover using pointers with peripheral registers or interpreting compiler warnings[[11]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Our%20Embedded%20C%20Online%20Test,libraries%2C%20crucial%20for%20firmware%20development).) Identify any remaining weak areas in C to address.
* **Day 43:** **MISRA C and Safe C Practices:** Read about MISRA C (Motor Industry Software Reliability Association) guidelines or other safety-oriented C subsets. While you don’t need to memorize rules, be aware of why certain practices (like not using dynamic memory or avoiding arbitrary pointer casting) are recommended in critical systems. This is relevant if your target field (e.g., automotive, or in general Garmin devices) emphasizes reliability.
* **Day 44:** **Optimize C Code:** Learn basics of how to optimize in C: for speed (using fixed-point instead of floating point if no FPU, loop unrolling concept) and for size (avoiding large stdlib calls, using -Os optimization). Experiment by writing a slow function (maybe computing Fibonacci in a naive way) and then improving it (use an iterative approach or memoization). Check the difference in execution time on hardware (you might toggle a GPIO at start/end of function and measure with a logic analyzer or by counting cycles using a debug register).
* **Day 45:** **End of C Phase Summary:** Summarize everything learned about C in a one-page note (in English or Chinese). List key concepts (pointers, interrupts, memory, etc.) and for each, ensure you can explain it briefly. This exercise will reinforce your knowledge and also create handy notes to review later.

*(At this point, you have built a solid C foundation, which is “mandatory” for embedded work*[*[4]*](https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/#:~:text=C%2B%2B%20is%20a%20useful%20skill,a%20mandatory%20skill%20in%20embedded)*. You’ve also gained* *microcontroller knowledge* *by working directly with STM32/Arduino hardware*[*[12]*](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Microcontroller%20Knowledge)*. Next, we move to C++ for the following phase.)*

## Phase 2: Transition to C++ (Days 91–150)

**Focus:** Learn C++ from the ground up and apply it in embedded contexts. C++ offers powerful features (OOP, templates, RAII, etc.) that can improve embedded code when used appropriately[[13]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=). However, embedded developers must use C++ carefully to avoid issues like dynamic memory allocation overhead[[14]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Despite%20the%20features%20that%20C%2B%2B,is%20NO%20dynamic%20memory%20allocation)[[15]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Another%20common%20argument%20against%20C%2B%2B,to%20make%20these%20mechanisms%20possible). In this phase, you will **not mix C and C++** – start fresh with C++ basics, then gradually incorporate embedded use-cases. By the end, you’ll know how to use modern C++ features beneficially (and what to avoid) in resource-constrained systems.

* **Day 91:** **C++ Setup and “Hello World”:** Set up your environment for C++ development. In STM32CubeIDE, you can create a C++ project (or convert an existing one by using .cpp files and ensuring the startup code supports C++ constructors). Also, for PC practice, ensure a g++ compiler is available. Write a simple “Hello World” in C++ on your PC to verify everything works (e.g., std::cout or printf in C++).
* **Day 92:** **C++ Basics – Syntax and I/O:** Learn the basic differences from C: iostream (std::cout/cin) vs C’s stdio, usage of std::string class instead of C strings, and new/delete vs malloc/free. Practice by writing a small C++ program that asks for user input (on PC, using std::cin) and prints a processed result. (Note: std::cout uses dynamic memory internally[[16]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Textbooks%20examples%20of%20C%2B%2B%20code,C%2B%2B%20in%20Niall%20Cooling%E2%80%99s%20article), so we typically won’t use it in embedded devices, but it’s fine on PC for learning.)
* **Day 93:** **Intro to OOP – Classes and Objects:** Learn how to define a class in C++. Understand class members (fields) and methods, public vs private. Write a simple class, e.g., class LED with a member variable for pin number and methods on() and off(). Instantiate objects of this class (perhaps representing different LEDs). This teaches basic encapsulation.
* **Day 94:** **Constructors and Destructors:** Extend your class with constructors (including parameterized ones) to initialize object state. Learn what a destructor is and when it gets called (and how it’s crucial for freeing resources if allocated). Modify your LED class to take a pin number in its constructor and maybe automatically configure that pin. No real hardware action yet, just concept (you can simulate by printing messages in methods).
* **Day 95:** **Inheritance and Polymorphism – Concepts:** Study inheritance in C++ (base and derived classes). For example, imagine a base class Sensor and derive classes TemperatureSensor, PressureSensor with specific methods. Learn what virtual functions are, and how polymorphism allows using a base pointer to call derived behavior. Write a small console program demonstrating this (it could simply print different messages for different sensor types via a virtual function).
* **Day 96:** **Polymorphism – Practice and override:** Implement the example from Day 95 in code. Focus on syntax: virtual keyword, override keyword in derived class, the vtable concept (no need to dive too deep, just understand each object of a polymorphic class carries a vtable pointer). Also experiment with not using virtual and see how behavior differs. This is important for embedded because virtual functions have slight overhead – know when it’s needed.
* **Day 97:** **References and Value vs Reference Semantics:** Understand C++ references (int&) vs pointers. Learn how passing by reference differs from passing by value. Write functions that modify variables via reference and via pointer to see the difference. Also cover const-correctness in C++ (e.g., const int&, int const \* etc.), which helps write safer code.
* **Day 98:** **Operator Overloading:** Learn how C++ allows you to overload operators. Try overloading a simple operator for a class (for example, define operator+ for a vector2D class to add two vectors). Operator overloading isn’t common in embedded development except for maybe custom math types, but it’s good to know for completeness.
* **Day 99:** **Templates – Introduction:** Templates are a key feature in C++. Begin with function templates (write a generic swap function template and test it with different types). Then try a simple class template (like a Buffer<T> that holds an array of T with a size). Understand that templates are compile-time and do not incur runtime overhead – a reason they can be great in embedded for generic code without cost.
* **Day 100:** **STL Containers (Vector, Array):** Learn about a couple of standard containers – std::vector and std::array. Write code using std::vector<int> to accumulate some data, and std::array<int, N> for fixed-size data. Understand the difference: std::vector allocates on the heap (avoid in small microcontrollers unless necessary), while std::array is a wrapper around C array that is stack/compile-time allocated[[17]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=C%2B%2B%20Standard%20Library%20offers%20a,style%20arrays)[[18]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=The%20above%20snippet%20of%20C,into%20the%20following%20C%2B%2B%20code). This distinction is important for embedded use of C++.
* **Day 101:** **Memory Management in C++:** Dive into new and delete, and the issues of dynamic memory in embedded. Understand what happens if you new without delete (memory leak). In embedded systems, dynamic allocation is often avoided due to fragmentation and lack of OS memory management – but you should know how it works. Write a test that allocates objects via new and frees them, perhaps track memory usage if possible.
* **Day 102:** **Smart Pointers (RAII):** Learn about C++11 smart pointers (std::unique\_ptr, std::shared\_ptr). These use RAII to manage memory automatically. Practice using std::unique\_ptr<int> in a small example, and understand how it calls delete when going out of scope. While on microcontrollers one might not use <memory> due to size, the concept of RAII (Resource Acquisition Is Initialization) is very useful (e.g., using destructors to release hardware resources like closing a file or disabling an interrupt).
* **Day 103:** **C++11/14/17 Features Recap:** Ensure you are aware of modern C++ features that improve code safety and clarity: auto type deduction, range-based for loops, nullptr (instead of NULL), enum class (strongly typed enums), constexpr, etc. Make a list and try examples of each. For instance, refactor a previous C++ example to use auto and range-for where appropriate.
* **Day 104:** **Exception Handling (Conceptual):** Understand what C++ exceptions are and how to use try/catch. But note: in embedded (especially on bare-metal or small RTOS systems), exceptions are typically **not** used because they add code bloat and runtime overhead[[19]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Another%20common%20argument%20against%20C%2B%2B,some%20extra%20information%20to%20binary). Many embedded compilers even allow disabling exceptions. You don’t need to practice extensively; just know how it works in case you see code with it, and why you should avoid it in microcontrollers.
* **Day 105:** **C++ in Embedded – Best Practices:** Read an article or documentation on using C++ in embedded systems (for example, MISRA C++ guidelines or an Embedded C++ article). Key points include: avoiding dynamic memory and exceptions, initializing all variables, using constexpr for compile-time computations, and leveraging templates for efficiency. One source notes that many embedded developers avoid certain STL features but do use classes and templates for clarity[[14]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Despite%20the%20features%20that%20C%2B%2B,is%20NO%20dynamic%20memory%20allocation). Summarize a few do’s and don’ts for future reference.
* **Day 106:** **Embedded C++ Project – GPIO Class:** Time to apply C++ on your hardware. Design a C++ class for a GPIO pin on STM32 or Arduino. The class should allow setting the pin as output, writing high/low, etc., encapsulating the register access inside methods. For example, class GpioPin with methods setMode(Output), write(high). Implement this for one pin (you can use STM32 HAL inside or direct register – focus on the C++ usage). Then use it to blink an LED via an object (e.g., GpioPin led(PA5); led.write(true);).
* **Day 107:** **Embedded C++ Project – UART Class:** Similarly, create a class to wrap UART functionality (e.g., Uart class with begin(baud), send(data) methods). Implement it using STM32’s HAL UART functions or register-level if feeling brave. Use the class to send a test message. This further solidifies using classes to organize embedded code.
* **Day 108:** **Mixing C and C++:** Learn how to integrate C and C++ code. Understand extern "C" linkage for calling C library functions or ISRs in C++ code. For example, if your startup code or ISR handler is in C, how to call a C++ function from it. Practice by calling a C library function (like printf or a C-written utility) from your C++ code, and vice versa. This is important because embedded codebases often combine C (for low-level parts or legacy code) with C++ for high-level logic.
* **Day 109:** **Templates Advanced – Policy Classes (Optional):** If you’re comfortable so far, explore a more advanced template usage like policy-based design or template specialization. For instance, design a template for a driver class that takes a policy for communication (one for I2C, one for SPI). This is highly advanced and not mandatory, but a peek at how templates can make embedded code flexible without runtime cost.
* **Day 110:** **STL Algorithms (Awareness):** Look at some common <algorithm> functions like std::sort, std::find. Understand how you could use them on STL containers. **Be cautious**: many algorithms may allocate memory or use recursion, which might be problematic in embedded. But know that std::sort on an array of ints, for example, should work and often is efficient. Try using std::sort on a std::array<int,50> and verify it works in a simple test program.
* **Day 111:** **C++ Code Optimization and Memory:** Investigate the memory footprint of C++ vs C. Write two similar programs (one in C, one in C++) that do the same basic task on STM32. Check the compiled binary sizes. Also, use the map file or nm tool to see symbols and memory usage. See how things like C++ constructors, static object instances, or virtual tables reflect in memory. This will give you insight into what features “cost” in terms of flash and RAM – crucial for fitting code on a microcontroller.
* **Day 112:** **RTTI and typeid (Conceptual):** RTTI (Run-Time Type Information) allows dynamic\_cast and typeid. Learn what it is, but similar to exceptions, know that many embedded systems disable RTTI to save space (because it adds type metadata in the binary)[[15]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Another%20common%20argument%20against%20C%2B%2B,to%20make%20these%20mechanisms%20possible). You likely won’t use this on an MCU, but be aware in case you see dynamic\_cast in code.
* **Day 113:** **Embedded C++ Project – FreeRTOS Preparation:** Anticipating the next phase (RTOS), make sure your project setup can compile C++ with FreeRTOS. FreeRTOS is in C, but you can write tasks in C++. Research how to use FreeRTOS in a C++ project (there are articles on linking C++ with FreeRTOS, e.g., ensuring vTaskCode is a static function or using lambdas with xTaskCreateStatic)[[20]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=). No coding today, just prepare knowledge for integration.
* **Day 114:** **C++ Quiz/Review:** Time to test your C++ knowledge. Use online quizzes or practice problems focusing on C++ (including new features). For example, answer questions like: *What is the Rule of Three/Five? When is a copy constructor called? Explain difference between struct in C vs C++.* Ensure you can confidently reason in C++ now.
* **Day 115:** **Final C++ Project – Multi-Class System:** As a capstone for C++ phase, implement a small system using multiple classes working together. For example, simulate a **simple weather station**: have a Sensor base class (with a virtual read()), two derived classes TempSensor and HumiditySensor (with dummy data), and a Display class that prints values. In main(), instantiate these and periodically read from sensors and display the values (you can just print to UART or console). This ties together inheritance, virtual functions, and composition of objects.
* **Day 116:** **Embedded Patterns – FSM:** Learn about Finite State Machine (FSM) pattern for embedded systems (often used in device firmware). You can use either C or C++ for this. Try implementing a simple state machine for an imagined device mode (e.g., a wearable that has states: Idle, Active, Syncing). Use an enum for states and a switch, or use function pointers for state handlers. This helps structure complex logic cleanly. (Reference: *Making Embedded Systems* by E. White discusses state machines – a recommended read[[21]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=,Refactoring%20Guru).)
* **Day 117:** **Standard Template Library in Embedded (Awareness):** Review which parts of the STL are safe or commonly used in embedded. For instance, std::array and maybe std::vector (in larger MCUs) are used, but std::string might be avoided. Some companies use a subset of STL. Knowing this will help you adapt to whatever environment Garmin has. Summarize which STL components you plan to use and which to be careful with (e.g., avoid heavy use of <algorithm> that might use heap).
* **Day 118:** **Build an Embedded Application in C++ (Project start):** Plan a small real embedded application combining C and C++: for example, *a data logger with OOP structure*. Outline it today – e.g., have a Sensor class (maybe interface to a real sensor or dummy), a Logger class (that stores data in memory or sends over UART), and main that ties them. Designing with classes should make it modular.
* **Day 119:** **Implement and Test Embedded C++ Application:** Code the project you planned on Day 118 on your STM32 or ESP32. For instance, if it’s a logger: one task reads sensor and populates a buffer (could just be an array), another prints or transmits the data. If not using RTOS yet, you can alternate in main loop or use timer interrupts. Test it thoroughly. This project consolidates your ability to structure embedded code in C++.
* **Day 120:** **C++ Phase Reflection:** Write a brief summary or journal of what you learned in C++ and how it contrasts with C. Note down key takeaways like “use classes for modularity, but be wary of dynamic memory”, etc. Also list any topics you want to explore more later. By now, you should feel comfortable in both C and C++, and knowing when to use each. 🎉

*(You have now completed the C++ phase. C++ is a valuable skill in embedded (many modern projects use C++ for organization), but C remains fundamental*[*[4]*](https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/#:~:text=C%2B%2B%20is%20a%20useful%20skill,a%20mandatory%20skill%20in%20embedded)*. Next, we will focus on* *embedded systems topics beyond language: real-time OS, hardware architecture, and advanced debugging.)*

## Phase 3: ARM Architecture & Low-Level Programming (Days 151–180)

**Focus:** Deepen your understanding of **computer architecture, especially ARM Cortex-M**, and low-level programming aspects like compilers and assembly. The Garmin job listing values “Computer architecture and ARM Compiler” knowledge[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge). This phase will be more theory-intensive but with practical labs to tie it to real code. You’ll learn how the hardware and the compiled code interact, preparing you to write efficient and hardware-aware firmware.

* **Day 151:** **CPU Architecture 101:** Review general CPU architecture concepts: registers, ALU, program counter, memory hierarchy (cache, if applicable; note that Cortex-M0/M3 have no cache, M7 does). Understand terms like ISR, pipeline, clock cycle. This high-level refresh sets the stage for ARM specifics.
* **Day 152:** **ARM Cortex-M Architecture Overview:** Study the ARM Cortex-M series architecture. Focus on Cortex-M4 (if your STM32 is M4) or M3. Key points: 32-bit registers (R0–R15), special registers (MSP/PSP for stack pointer, LR, PC, xPSR), the NVIC (interrupt controller), and how exceptions (interrupts) are prioritized and handled. ARM’s documentation or a summary online will be useful.
* **Day 153:** **ARM Assembly Basics:** Learn the basics of ARM assembly language. Understand the syntax (MOV, LDR, STR, ADD, SUB, etc.), and how C code maps to assembly. Do a simple experiment: write a very short C function (like adding two numbers) and use the compiler to generate an assembly listing (e.g., arm-none-eabi-gcc -S). Inspect the assembly to see the function prologue/epilogue and instructions used[[22]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Historically%20speaking%2C%20C%2B%2B%20started%20as,and%20they%20are%20well%20documented). This will demystify what high-level code translates to.
* **Day 154:** **ARM Thumb Instruction Set:** Cortex-M uses the Thumb2 instruction set (mixed 16/32-bit instructions). Read about how Thumb2 differs (mostly for curiosity/understanding, you don’t need to memorize opcodes). Knowing that the MCU is running Thumb code helps if you ever debug at the assembly level.
* **Day 155:** **System Startup (Reset Vector):** Investigate what happens when the microcontroller resets. Open the startup assembly file in your STM32CubeIDE project (usually startup\_stm32.s) and identify the reset handler, initial stack pointer, and how it calls SystemInit and then main. Understand how the .data and .bss sections are initialized (copy from flash to RAM, zero out BSS). This is often done in assembly before main – *know this flow*.
* **Day 156:** **Linker Script and Memory Map:** Open the linker script (usually \*.ld file) for your project. Study how memory regions are defined (FLASH, RAM addresses and sizes) and how sections (.text, .data, .bss, .heap, .stack) are placed. Modify a small thing in the linker script (for learning) – e.g., create a new section for constants – and place a variable there to see effect. Understanding the linker script is part of “knowing the ARM compiler” and how it organizes code.
* **Day 157:** **Computer Architecture – Advanced:** Learn about advanced concepts that affect embedded software: endianness (ARM is little-endian on MCUs), unaligned memory access (Cortex-M allows unaligned access for some types, but it’s best to naturally align data for performance). Also, understand how memory accesses might be buffered (write buffer) or how peripheral registers might have delays – this is typically found in reference manuals as “bit write timings”.
* **Day 158:** **ARM Cortex-M Peripherals & Bus Matrix:** Read about how the Cortex-M core interacts with peripherals (AHB/APB buses on STM32). Know that different peripherals might be on different buses with different speeds. For example, memory or DMA can access certain regions. While detailed chip specifics aren’t needed by heart, being aware of bus architecture helps optimize (e.g., don’t do heavy computation when bus is busy with DMA).
* **Day 159:** **Interrupt Mechanism Deep Dive:** Take a deeper look at how an interrupt is handled: from the hardware perspective. Learn about the vector table (located at address 0x0 in flash by default on ARM Cortex-M), how the NVIC latches an interrupt, pushes registers onto the stack, and jumps to the ISR. Understand interrupt latency and tail-chaining (Cortex-M feature to handle back-to-back interrupts efficiently). This knowledge is useful for writing latency-sensitive code.
* **Day 160:** **Context Switching and System Mode:** If you plan to use an RTOS, understand how the CPU allows switching contexts. Learn about the PendSV interrupt on Cortex-M (used by many RTOS for context switch) and how MSP and PSP (two stack pointers) are used – one for OS (handler mode) and one for threads (thread mode). Even if you don’t implement it, conceptually knowing this will make RTOS usage clearer.
* **Day 161:** **Profiling and Cycle Counting:** On ARM Cortex-M, there is a DWT (Data Watchpoint and Trace) unit with a cycle counter register (DWT\_CYCCNT). Learn how to enable and read this to measure cycles taken by code. Write a test in C to measure how many cycles a function takes (enable DWT\_CYCCNT, read it before and after calling the function). This is a powerful trick for optimizing code by actual measurement.
* **Day 162:** **Compiler Optimizations:** Explore how compiler optimization levels affect your program. Compile the same code with -O0, -O2, -Os and compare the size and speed. Look at the assembly output for a small function at different optimization levels to see differences (e.g., -O0 might use a lot of stack, -O2 might keep variables in registers and unroll loops). This teaches what the ARM compiler is doing under the hood, and why sometimes code behaves differently with optimizations (e.g., needing volatile).
* **Day 163:** **Inline Assembly and Intrinsics:** Learn how to include a bit of assembly in your C/C++ code using inline asm or compiler intrinsics. For example, use the \_\_NOP() intrinsic (from CMSIS) to insert a no-operation, or write a few lines of inline assembly to do something trivial like read control register. Be careful and follow examples (since incorrect inline asm can crash the program). This skill isn’t used daily, but it’s part of low-level programming prowess and sometimes needed for things like issuing a special CPU instruction.
* **Day 164:** **Memory Protection and Faults:** Understand how the ARM Cortex-M Fault handlers work (HardFault, MemManage, BusFault, UsageFault). Trigger a HardFault intentionally (e.g., by dereferencing an invalid pointer or executing an undefined instruction via asm) and see how to catch it. Learn to read the fault registers (like MMFSR, BFAR) to diagnose crashes. This deepens your debugging ability – you can interpret *why* a crash happened (crucial in firmware debugging).
* **Day 165:** **Other Architectures (Contrast):** Briefly look at how other microcontroller architectures differ (e.g., 8-bit AVR used in Arduino vs 32-bit ARM, or MSP430). This will give perspective on ARM strengths. For instance, AVR has limited stack and different instruction set. Knowing this broadens your understanding but also highlights common principles (like all have program counters, etc.).
* **Day 166:** **Embedded Linux (High-Level OS intro):** Although the job is focused on bare-metal/RTOS, having “other OS knowledge”[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge) is a plus. Spend today learning at a high level how embedded Linux works (just conceptually): that it runs on more powerful processors with an MMU, how device drivers on Linux differ from firmware on microcontroller. If you have a Raspberry Pi or similar, you could blink an LED using Linux sysfs GPIO as a quick exercise to see the contrast (user-space vs kernel-space).
* **Day 167:** **Real-Time Systems Theory:** Learn what makes an operating system *real-time*. Understand concepts like deterministic scheduling, interrupt latency, priority inversion, and deadlines. This theory will be important when working with FreeRTOS or any RTOS. Write notes on the difference between a general-purpose OS and an RTOS (e.g., Linux vs FreeRTOS) – for instance, Linux can’t guarantee timing, whereas an RTOS can if used properly.
* **Day 168:** **ARM Compiler Specifics:** Explore any ARM compiler specific features or extensions. For instance, **ARM GCC intrinsics** (like \_\_enable\_irq(), \_\_disable\_irq() from CMSIS) and how to use them instead of writing assembly. Also, read about ABI (Application Binary Interface) for ARM – not in detail, just know that function call conventions (how parameters are passed in registers) follow the ARM EABI. This can help when debugging assembly or linking with libraries.
* **Day 169:** **Toolchain Exploration:** Try using a different compiler or IDE to broaden exposure. For example, install **Keil MDK** or **IAR Embedded Workbench** (they often have code size limited free versions) and compile one of your earlier projects. See if any compiler warnings or issues arise. Each compiler has slight differences (Keil uses Arm Compiler, IAR has its own). While you may stick to GCC, exposure to others is useful (Garmin or others might use different toolchains).
* **Day 170:** **Project: ARM Performance Tuning:** Take an earlier project (e.g., the sensor logger) and attempt to make it more efficient using knowledge from this phase. For example, if you used a floating-point calculation, try using integer math to compare speed. If you have an LED or scope, measure a loop timing before and after optimization. Use the cycle counter from Day 161 to quantify improvement. Document the changes you made (this demonstrates an ability to optimize low-level code).
* **Day 171:** **Interview Prep – C/Architecture Questions:** Compile a list of common interview questions related to C and low-level knowledge. For example: *Explain what happens when you type printf("Hello"); in a bare-metal program (covering linking, how it might not work without a stdout device).* Or *How does an interrupt differ from a function call in terms of execution?* Practice answering these out loud. The goal is to articulate these technical concepts clearly – a likely scenario in interviews.
* **Day 172:** **Catch-up / Extra Reading:** Use this day to catch up on any topic from days 151–171 that you feel you didn’t fully absorb. Alternatively, do some extra reading on an area of interest – e.g., read a chapter from **Joseph Yiu’s “Definitive Guide to ARM Cortex-M3/M4 Processors”** (a great resource if available to you). Solidify any remaining doubts.
* **Day 173:** **Build an ARM Cheat-Sheet:** Create a one-page cheat-sheet or mind-map of ARM Cortex-M key points: registers, interrupts, memory layout, etc., and C/C++ low-level tips (like keywords volatile, inline, alignment). This will be handy for quick review later and ensures you have the big picture in one place.
* **Day 174:** **Masters Thesis / Academic Linking (Optional):** Since the job expects a Master’s degree knowledge, try to connect what you learned to any academic theory from your coursework. For instance, if you studied operating systems, relate it to FreeRTOS; if you studied computer architecture, relate to pipeline and caches. This can help you in interviews to show you can bridge theory and practice.
* **Day 175:** **Phase 3 Wrap-up Discussion:** Explain (to yourself or a peer) the journey of a program from C code to execution on ARM. Start from writing code, compiling (mention the compiler and linker roles), to how the binary is laid out in flash, to reset vector, initialization, and execution. Include how an interrupt might occur and be handled. Being able to narrate this coherently means you’ve integrated the knowledge well.
* **Day 176:** **Real-World Architecture Application:** As a fun application, research what architecture Garmin devices use (if information is public – many Garmin wearables use ARM Cortex-M or A for higher-end). See if you can find any teardown or specs. Understanding the target platform gives you context. If they use Cortex-M, you’re spot on; if some use Cortex-A (for more advanced devices), note the differences (Cortex-A runs Linux typically).
* **Day 177:** **Revisit C and C++ Code with Fresh Eyes:** Go back to some of your older code (from Phase 1 or 2) and review it using your improved understanding of low-level. Can you spot any potential inefficiencies or pitfalls? (e.g., a variable that should be volatile, or an array that could be static const to place in flash). Make improvements if needed. This is good practice for code review skills and shows continuous improvement.
* **Day 178:** **Begin RTOS Theory – Preparation:** The next phase is RTOS, so spend today gathering what you know and don’t know about RTOS. List concepts: tasks/threads, scheduling, context switch, synchronization (semaphores, mutexes), etc. Read an introductory article on FreeRTOS or a chapter from a textbook on RTOS basics. This will prep your mind for hands-on RTOS work.
* **Day 179:** **Set Up FreeRTOS Environment:** Make sure FreeRTOS is available for your STM32. You can use STM32CubeMX to add FreeRTOS to a project (CubeMX can generate the OS initialization code and tasks) or download FreeRTOS source yourself. Create a new STM32CubeIDE project with FreeRTOS enabled (or use an example project provided by ST for FreeRTOS). Verify it compiles and understand the project structure (you’ll see tasks.c, queue.c, etc.).
* **Day 180:** **FreeRTOS Config and Tuning:** Open the FreeRTOSConfig.h file in your project and browse the settings. Understand important ones like configCPU\_CLOCK\_HZ, configTICK\_RATE\_HZ, configMINIMAL\_STACK\_SIZE, configTOTAL\_HEAP\_SIZE, etc. These configure how the RTOS runs. You don’t need to change them now, just be aware. Also note how FreeRTOS uses heap (you can choose heap\_1.c, heap\_4.c etc. for different allocation schemes). This knowledge will help when creating tasks and avoiding memory issues.

*(By now, you have a strong grasp of the hardware and low-level aspects (computer architecture, compiler, assembly). These are the underpinnings that make you a* *firmware engineer* *rather than just a coder. Next, we leverage this knowledge as we dive into* *Real-Time Operating Systems* *and advanced embedded development.)*

## Phase 4: Real-Time Operating Systems (FreeRTOS) & Concurrency (Days 181–240)

**Focus:** Gain hands-on experience with RTOS, particularly FreeRTOS (an industry-standard real-time OS for microcontrollers). The job expects experience with embedded systems or RTOS[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge), and RTOS knowledge is crucial for managing time-sensitive tasks in complex devices[[23]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Real). In this phase, you will learn how to create and synchronize tasks, use RTOS features (semaphores, queues), and build a multi-threaded firmware similar to what runs in real products (like Garmin’s devices). We’ll integrate knowledge of interrupts and hardware with the RTOS environment.

* **Day 181:** **FreeRTOS Basics – Task Creation:** In your FreeRTOS-enabled STM32 project, create two simple tasks (threads). For example, TaskA blinks an LED, TaskB sends “Hello” over UART in a loop. Use xTaskCreate (or the static create variant) to create these tasks with appropriate stack sizes and priorities. Start the scheduler (vTaskStartScheduler) and verify both tasks run (the LED blinks and "Hello" prints periodically). Congrats – you have a multitasking system running!
* **Day 182:** **FreeRTOS Delay and Scheduling:** Learn how the RTOS scheduler works with ticks. Use vTaskDelay() or vTaskDelayUntil() in your tasks to manage their execution period. For instance, make TaskA blink LED every 500ms using vTaskDelay, and TaskB print every 1000ms. Observe that the RTOS handles the timing. Understand the concept of *time slicing* (if two tasks have the same priority) and how tick interrupts drive scheduling.
* **Day 183:** **Task Priorities:** Experiment with task priorities. Assign TaskA a higher priority than TaskB and see the effect (TaskA should preempt TaskB if it becomes ready). Then try the reverse. This will teach you how priority inversion can be an issue if a lower-priority task holds a resource needed by a higher one (we’ll address that with mutexes later). FreeRTOS lets you assign configMAX\_PRIORITIES – ensure you understand that higher number means higher priority by default.
* **Day 184:** **Inter-Task Communication – Queues:** Learn about FreeRTOS queues for exchanging data between tasks. Modify TaskA and TaskB such that TaskA produces data (e.g., reads a pretend sensor value or counter) and sends it to TaskB via a queue. TaskB will receive from the queue and, say, print the value. Use xQueueCreate, xQueueSend, xQueueReceive. This is a powerful mechanism to safely communicate in RTOS without global variables.
* **Day 185:** **Binary Semaphores for Sync:** Learn about binary semaphores (and why they’re often used to signal events between ISRs and tasks). For practice, set up an external interrupt or timer interrupt (from earlier phases) and in the ISR give a semaphore (xSemaphoreGiveFromISR). Have a high-priority task pending on xSemaphoreTake. When the interrupt occurs, the task unblocks and handles the “event” (e.g., button press). This mimics real-world interrupt-to-thread synchronization.
* **Day 186:** **Mutexes for Shared Resources:** Understand the need for mutual exclusion. Create a scenario with a shared resource (for example, two tasks trying to send over UART). Protect the UART transmit function with a mutex (xSemaphoreCreateMutex). Make both tasks attempt to print in a loop without and then with the mutex to observe the difference (without mutex, messages may intermix; with mutex, they serialize properly). Learn about priority inheritance – FreeRTOS mutexes implement this to avoid priority inversion.
* **Day 187:** **FreeRTOS Memory Management:** Deep dive into how FreeRTOS allocates memory for tasks and objects. If using dynamic allocation (xTaskCreate), it uses the FreeRTOS heap. Ensure you understand the heap types (heap\_1 is simplest (no free), heap\_4 is a malloc/free implementation). Try using pvPortMalloc and vPortFree (FreeRTOS’s versions of malloc/free) in a task to see how you might allocate memory safely in a threaded environment. If possible, switch to using static allocation (xTaskCreateStatic) for tasks to get a feel for both methods.
* **Day 188:** **Timing and Tick Hook:** Implement a tick hook or use the idle hook in FreeRTOS (configUSE\_TICK\_HOOK / configUSE\_IDLE\_HOOK) to do something (maybe increment a counter or toggle an LED very fast). This helps understand how the OS can run your code in the idle time or every tick – but be very mindful of not doing heavy work in these hooks. It’s a glimpse into more advanced usage of RTOS for background tasks.
* **Day 189:** **Advanced RTOS – Event Groups:** Learn about Event Groups (a FreeRTOS feature to signal multiple bits/events in one object). For example, you could have multiple flags (bit0 = sensorA ready, bit1 = sensorB ready, etc.) and tasks waiting for combinations. You may not need this often, but it’s good to know. Create a simple test where two tasks each set different event bits (using xEventGroupSetBits), and a third task waits for both bits to be set before proceeding.
* **Day 190:** **RTOS Trace Tools (Optional):** If available, try a trace tool for FreeRTOS (like FreeRTOS+Trace or Percepio Tracealyzer – they might have trial versions). These tools record task execution and can visualize timing. Alternatively, use uxTaskGetSystemState or the built-in tracing macros to gather basic runtime stats (like task CPU usage). This teaches you how to analyze a running RTOS system, which is useful for performance tuning.
* **Day 191:** **Compare RTOS vs Bare Metal:** Reflect on how writing the LED blink and UART code under FreeRTOS differed from the bare-metal (no RTOS) approach. List pros and cons: e.g., RTOS made timing easier (using delays) and organization cleaner (separate tasks), but adds overhead and complexity (context switches). Understanding this trade-off will help you choose when to use an RTOS in projects.
* **Day 192:** **Introduction to ROS2 (Optional/Extra):** You mentioned using ROS2 in your internship (which is a robotics middleware, usually on Linux). While not directly related to our microcontroller focus, it deals with distributed real-time messaging. Spend a short time thinking how the concepts overlap: e.g., ROS nodes kind of like tasks, topics like queues. This isn’t directly needed for the Garmin job, but since you have exposure, connecting the dots might deepen understanding of real-time communication patterns.
* **Day 193:** **ESP32 and FreeRTOS:** The ESP32’s native SDK (ESP-IDF) runs FreeRTOS under the hood. If interested, try writing a simple FreeRTOS task on the ESP32 (e.g., using Arduino core’s xTaskCreate or via ESP-IDF if you set it up). For example, blink the onboard LED as one task and print WiFi status as another. This will show you how RTOS is everywhere, even in WiFi/IoT chips, and give you practice on a different platform.
* **Day 194:** **RTOS Project Planning:** Plan a larger project that utilizes RTOS to simulate a **fitness device** behavior. For example, consider a “Fitness Tracker” system with multiple tasks: one task reads sensor data (accelerometer for steps, or GPS coordinates), another task logs data or updates a display, another handles communication (BLE or saving to flash). Today, design the tasks, their responsibilities, and how they interact (draw a block diagram or write it down).
* **Day 195:** **RTOS Project – Sensor Simulation Task:** Implement Task 1 of the project: a *Sensor Task*. If you have an accelerometer or GPS module, use it; if not, simulate sensor input. For instance, simulate step counts or GPS coordinates changes. This task will periodically produce data (e.g., every 1 second update steps). Make it post the data to a queue or global structure protected by a mutex.
* **Day 196:** **RTOS Project – Data Processing Task:** Implement Task 2: a *Processing Task* that receives data from Task 1 (via queue or direct if global) and processes it. Processing could be calculating total distance from GPS coords, or summing step counts, or filtering sensor noise. Even if simulated, apply a meaningful computation (e.g., a moving average filter on a noisy sensor value). This task might run at a lower priority than sensor task if real-time data acquisition is more critical.
* **Day 197:** **RTOS Project – Communication/Output Task:** Implement Task 3: an *Output Task*. This could be responsible for communicating the processed data off-device. Options: send via UART (to simulate sending to a PC or logging to an SD card) or if you have an ESP32 and want a challenge, have it send data over WiFi or BLE (ESP32 could act as a BLE server broadcasting data). Simpler: just have it print the values (steps, distance, etc.) to UART or toggle LEDs to indicate status.
* **Day 198:** **RTOS Project – Synchronization & Refinement:** Now ensure these tasks work together properly. Use semaphores or direct-to-task notifications if needed to sync (e.g., maybe the output task waits for new data available signal). Ensure no data races: if tasks share data, protect with mutexes or only use queues. Debug any issues (like one task starving others – adjust priorities or add delays). This is a mini simulation of a wearable device firmware.
* **Day 199:** **RTOS Project – Testing:** Test your multi-task system thoroughly. Simulate various scenarios: sensor task overload (maybe increase its frequency to see if others keep up), queue overflow (what if data produced faster than consumed?), etc. This will teach you to observe and fix real-time issues. Use debugger or toggle an LED in each task to visually see their execution (fast blink vs slow blink patterns).
* **Day 200:** **RTOS Project – Add an Interrupt (Button):** Integrate an interrupt into the system. For example, a button press could trigger an ISR that unblocks a high-priority task (maybe a *Mode Switch Task*). Implement that: when button is pressed, an ISR gives a semaphore and a task then toggles between modes (maybe switches the units from metric to imperial, just for simulation). This adds more realism – devices often have interrupts (button, sensor interrupts) waking tasks.
* **Day 201:** **Code Review and Cleanup:** Perform a self code review of your RTOS project. Ensure you have no obvious issues (like forgetting to delete a mutex, or potential priority inversion unhandled – though FreeRTOS handles mutex inversion). Clean up the code, add comments, and possibly write a short README as if you were going to show this project to someone (this can later be part of your portfolio for job applications).
* **Day 202:** **RTOS Advanced Topics:** Read about any advanced FreeRTOS features you haven’t tried: for example, **Timers** (FreeRTOS software timers) which call a callback in timer context, **Queuesets**, or **Stream Buffers**/MessageBuffer. If any seem relevant, experiment briefly. For instance, a software timer could be used to trigger a periodic sensor read instead of a dedicated task delay. Add one software timer to your project for practice (e.g., a timer that toggles an LED every 5 seconds by calling back in timer context – note you must be careful what you do in that context).
* **Day 203:** **Power Management & RTOS:** Research how an RTOS system can enter low-power modes. FreeRTOS on ARM can be configured to tickless idle (where it stops the tick timer when idle to save power). Read about “tickless idle” mode. If your application has idle time (and it should), consider enabling tickless idle in FreeRTOSConfig and see if you can measure difference (this might require an oscilloscope or current measurement, which you might not have – so just theoretical understanding is fine). Given wearables run on battery, power saving is crucial, so know the concepts.
* **Day 204:** **Memory Management in RTOS Apps:** If your project uses dynamic memory (via malloc or new), consider what happens if memory runs out or fragments. Learn strategies used in firmware: e.g., using memory pools or all static allocation. As a thought exercise, design how you might allocate memory for, say, 100 data samples every minute – would you malloc each time or use a ring buffer? Perhaps adjust your project to use a static buffer instead of continuously malloc’ing to simulate best practice.
* **Day 205:** **Embedded Linux vs RTOS (Revisit):** Take a moment to contrast an RTOS-based system with an embedded Linux system (like a smartphone or Raspberry Pi). Know that Garmin’s devices likely run on microcontrollers with RTOS rather than Linux (for power and simplicity), but some advanced models might use an embedded Linux. If possible, find an example of a Linux-based embedded product vs an RTOS one and list differences (boot time, complexity, capabilities). This broad knowledge fits “other OS related knowledge”[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge) and shows you can contextualize why one might use an RTOS.
* **Day 206:** **Fault Tolerance in RTOS:** Consider what happens if a task crashes (e.g., divides by zero or stack overflow). FreeRTOS has hooks for stack overflow (if enabled). Enable configCHECK\_FOR\_STACK\_OVERFLOW and deliberately create a small stack task that overflows to see the hook fire (make sure to implement vApplicationStackOverflowHook). Also consider using assert in code to catch issues. This is about building robust firmware that can handle or at least detect faults.
* **Day 207:** **RTOS Quiz/Review:** Use today to solidify RTOS knowledge. Quiz yourself: *What is the difference between a binary semaphore and a mutex? How does priority inheritance work? What is the worst-case interrupt latency if you have interrupts off for a critical section?* Ensure you can answer such questions clearly. Maybe write them and answers down. These are the kinds of questions an interviewer might pose to gauge your practical RTOS understanding.
* **Day 208:** **Soft Skills – Documentation:** Practice writing a small design document for the RTOS project you just built. Outline the purpose, how tasks interact, and why you chose certain priorities. Since the job also values communication (English/Chinese fluency), being able to document and explain your design is important. Aim to make it clear enough that another engineer (or interviewer) reading it can follow your logic.
* **Day 209:** **Team Skills – Code Review Simulation:** If you can, have a friend or colleague review your code, or at least imagine a critical eye going over it. Think of questions they might ask (“Why did you use a queue here instead of a direct global variable?”, “How do you handle if the sensor task overruns its period?”). Formulate answers. This prepares you for technical discussions about your work and shows the reflective, team-oriented mindset (soft skill).
* **Day 210:** **Wrap-up RTOS Phase:** Summarize the key learnings from RTOS work. You now have a multi-tasking firmware running, which is a big milestone. Write a list of FreeRTOS features you used and those not yet explored. Note the areas you found challenging (e.g., thread-safe design, configuring stack sizes) and how you overcame them. This not only reinforces your knowledge but gives you talking points for interviews (“One challenge I solved was tuning task stack sizes to avoid overflow...”).

*(At this stage, you have developed a functioning RTOS-based system, essentially simulating a simplified wearable device firmware. You've covered* *RTOS (FreeRTOS) skills* *that are crucial for embedded engineers*[*[23]*](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Real)*. The next phase will focus on polishing up* *tools and debugging skills* *like logic analyzers and JTAG, as well as final projects and preparation for the job application.)*

## Phase 5: Tools & Advanced Debugging (Days 241–270)

**Focus:** Gain experience with **embedded testing and debugging tools** – specifically logic analyzers and JTAG ICE – and other practical skills (soldering, basic circuit understanding) to round out your profile. The job mentions knowledge of logic analyzers and JTAG[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge), which are used for debugging hardware/software interactions[[10]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing). This phase will be a mix of learning to use these tools and tying up any remaining loose ends in your embedded skillset.

* **Day 241:** **Logic Analyzer Basics:** Learn what a **Logic Analyzer** is and how it works. Read a tutorial or guide (there are many online, even in Chinese if you prefer, e.g., 儀器使用指南). Understand that an LA samples digital signals at high speed and lets you inspect waveforms and timing. Key specs: number of channels, sample rate. This is for debugging communication buses (I2C, SPI, UART) or timing issues.
* **Day 242:** **Set Up a Logic Analyzer:** If you have access to a logic analyzer (even a cheap USB one or a borrowed unit), install the necessary software (e.g., Saleae Logic software or Sigrok’s PulseView for clones). If not, consider using a simulator like Wokwi (which has a virtual logic analyzer)[[24]](https://docs.wokwi.com/zh-CN/guides/logic-analyzer#:~:text=%E9%80%BB%E8%BE%91%E5%88%86%E6%9E%90%E4%BB%AA%E4%BD%BF%E7%94%A8%E6%8C%87%E5%8D%97%20,%E2%80%9D%E3%80%82%20%E6%8E%A5%E4%B8%8B%E6%9D%A5%EF%BC%8C%E5%B0%86%E8%A6%81%E8%AE%B0%E5%BD%95%E7%9A%84%E4%BF%A1%E5%8F%B7%E8%BF%9E%E6%8E%A5%E5%88%B0Logic%20Analyzer%E5%BC%95%E8%84%9A%E3%80%82%E4%BE%8B%E5%A6%82). Connect the logic analyzer to your microcontroller: for instance, attach it to the I2C SCL and SDA lines from your Day 24 sensor project or the UART TX line from Day 15.
* **Day 243:** **Capture and Analyze I2C:** Perform a logic analyzer capture of an I2C transaction (or UART if I2C not available). Trigger your STM32 to communicate over I2C (maybe re-run the sensor read code). Use the LA software to decode the I2C frames (most LA software can decode protocols). Examine the timing: do you see the start bit, address, ACK/NACK bits, data bytes? This visual confirmation helps solidify your understanding of the protocol and shows how LA is used to **troubleshoot hardware comms**[[10]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing).
* **Day 244:** **Capture and Analyze UART:** Similarly, capture a UART transmission from your microcontroller (e.g., the "Hello" messages). Use LA to decode UART bytes or simply observe the waveform (start bit, data bits, etc.). Try to capture at different baud rates. This exercise demonstrates using LA for asynchronous serial debugging (useful if you need to reverse-engineer or verify baud timings).
* **Day 245:** **Logic Analyzer for Timing:** Use the logic analyzer to measure timing precisely. For example, toggle a GPIO at the start and end of a code section (or use the built-in square wave output if LA has it) to measure execution time, and capture that on LA. Alternatively, if you have two tasks blinking LEDs, capture both LED signals to see how they interleave. The logic analyzer can function as a poor-man’s oscilloscope for digital signals, helping verify timing constraints (e.g., is your 1kHz PWM really 1kHz?).
* **Day 246:** **JTAG and SWD Theory:** Review what **JTAG** is (Joint Test Action Group interface) and how it allows control of the processor for debugging. Understand that STM32 uses a simplified 2-wire SWD (Serial Wire Debug) which is essentially JTAG-like. JTAG also can chain multiple devices. Know the terms: boundary scan (JTAG originally for testing hardware), and how debuggers use JTAG to halt/run the CPU, set breakpoints, read memory. This underpins the debugging you’ve been doing in your IDE.
* **Day 247:** **Using JTAG/SWD Debuggers:** You’ve used ST-Link (SWD) via CubeIDE; now try using a different approach to deepen knowledge. For instance, install OpenOCD (open-source JTAG server) and GDB. Connect to your STM32 using OpenOCD + GDB manually (this is a bit advanced: it involves config files for your board). If successful, practice basic GDB commands (step, continue, print var, etc.) to debug. This lower-level view helps you appreciate what the IDE does and prepares you to debug in environments without a fancy IDE (or to troubleshoot JTAG connections).
* **Day 248:** **Debugging with JTAG – Practice:** Intentionally break your firmware (e.g., introduce a bug that causes a HardFault). Use the debugger (CubeIDE or GDB) to catch it. Inspect the call stack, registers, and figure out the cause. Also, practice using watchpoints (data breakpoints) to stop when a variable changes – a powerful debugging feature enabled by the debug hardware. This showcases advanced usage of the JTAG debugger beyond basic breakpoints.
* **Day 249:** **JTAG/SWD Tools:** Familiarize with other debug tools: for instance, SEGGER J-Link (popular JTAG probe) and its software (Ozone debugger, etc.), or Black Magic Probe. You might not have these, but knowing of them is good. Read a bit on how they differ (performance, features like RTT for real-time logging). If aiming for Garmin, see if they mention any specific tools; Garmin might have their own hardware debuggers or use J-Link.
* **Day 250:** **Soldering and Hardware Skills (Optional Hands-on):** As an embedded engineer, hardware understanding is valuable. If you have access to a soldering iron and some components, practice soldering by making a simple circuit (like solder header pins to a breakout board or assembling a small kit). If not, watch a tutorial on PCB assembly. Also, ensure you can read simple schematics (like your STM32 board’s schematic: identify power supply, oscillator, debug connector). This isn’t strictly software, but well-rounded knowledge helps in debugging issues (like recognizing if a wire is connected wrong or a component could be faulty).
* **Day 251:** **Oscilloscope (the basics):** While the logic analyzer covers digital, a word on oscilloscopes: read about how an oscilloscope is used for analog signals or quickly changing digital signals where you need to see shape (e.g., signal integrity issues). If possible, get a brief hands-on or simulation with an oscilloscope (maybe your workplace has one). This knowledge complements logic analyzer usage (LA for protocol/decode, scope for analog and analog-esque digital issues).
* **Day 252:** **Test Equipment Integration:** Think of a scenario: you have a device not working, how do you approach it with tools? For instance, a sensor isn’t responding – you’d use a logic analyzer to see I2C lines, a multimeter to check if power is correct, maybe an oscilloscope for noise. Write down a hypothetical troubleshooting procedure for a bug that could occur in a wearable (e.g., heart-rate sensor not giving data). This prepares you to answer troubleshooting questions and demonstrates a methodical approach using tools.
* **Day 253:** **Continuous Integration Basics:** Many firms (possibly Garmin) use CI/CD to build and test firmware. Get familiar with the concept: maybe set up a simple continuous integration for your project using GitHub Actions or another service, to automatically compile your code on commit. This is optional, but shows awareness of professional development practices. Alternatively, at least document how you would test your code (unit tests on PC for algorithm parts, integration test on hardware manually, etc.).
* **Day 254:** **Protocol Deep Dive (BLE or ANT):** Garmin devices often use ANT+ and BLE for wireless connectivity. Take a day to learn the basics of BLE (Bluetooth Low Energy) protocol, since you have an ESP32 (which can do BLE). Understand concepts like advertising, GATT services. If time permits, run a simple BLE example on the ESP32 (like a beacon or thermometer profile) just to expose yourself to it. Or read about ANT+ (Garmin’s proprietary protocol for sensors) – at least know that such protocols exist and key characteristics.
* **Day 255:** **GPS Technology Basics:** Since the target products involve **Fitness GPS**, learn how GPS works. Understand terms: satellite signals, NMEA sentences, the need for line-of-sight, how latitude/longitude are obtained. If you have a GPS module (or smartphone GPS accessible via an app), get some sample NMEA data. Write a small parser in Python or C for one NMEA sentence (like $GPRMC). This isn’t directly coding on the device, but domain knowledge so you know what the device’s “GPS product design” entails.
* **Day 256:** **Sensor Fusion & Algorithms (Conceptual):** Fitness devices often integrate data from multiple sensors (GPS, accelerometer, maybe altimeter, heart rate). Read about basic sensor fusion or filtering techniques (e.g., Kalman filter basics, or simply how to combine accelerometer data to detect steps). You don’t need to implement a full algorithm, but be aware of how raw data becomes meaningful info (e.g., step counting algorithm from accelerometer peaks). This could be a nice discussion point in interviews to show you understand the product domain.
* **Day 257:** **Error Handling Strategies:** In embedded systems, robust error handling is key. Think about how your projects handle errors: e.g., what if a sensor read fails? Practice implementing a simple error handling scheme: perhaps a function returns an error code and the caller task logs it. Or use an LED blink pattern to indicate an error (common in devices: e.g., 3 blinks = error code 3). Being ready to discuss how you ensure reliability is important.
* **Day 258:** **Bootloaders and Firmware Updates (Overview):** Most products need firmware update capability. Read about how a bootloader works – a small program that can receive a new firmware image (via USB, BLE, etc.) and program it. If interested, look at STM32’s built-in bootloader options (many have a UART or USB DFU bootloader). You don’t have to build one now, but understand the principle (this is bonus knowledge that could impress in relevant discussion).
* **Day 259:** **Embedded Security Basics:** As devices are connected (GPS devices might connect to phone or PC), basic security is a topic. Learn a bit about embedded cryptography: e.g., what is AES and how might a small device use it (maybe to store data securely or authenticate firmware updates). Also concepts like secure boot (ensuring firmware isn’t tampered). This is another “extra” that shows you think broadly about the system.
* **Day 260:** **Open-Ended Exploration:** Use this day to explore any remaining curiosity or weakness. Perhaps revisit Chinese resources to see if any local community projects similar to what you’re doing exist. Or join an online embedded forum (like Reddit r/embedded or a local group) and read some Q&A to see common problems others face. This can expose you to things you might not have covered and is a form of continuous learning.
* **Day 261:** **Project: Custom Peripheral Driver:** Design a mini-project where you write a driver for a peripheral from scratch in C (no HAL). For instance, choose SPI this time. If you have an SPI sensor or device (maybe an SPI Flash chip or a screen), write code to send and receive data over SPI by directly manipulating registers (or using minimal HAL calls if needed). The driver should initialize SPI, send some commands, and read responses. This demonstrates your ability to work at the register level to create drivers – a valuable skill for embedded engineers.
* **Day 262:** **Project: Use Logic Analyzer on New Driver:** Use the logic analyzer to verify the SPI driver from Day 261 is functioning. Capture the SPI clock, MOSI, MISO lines. Check that the clock polarity and phase match expectations (you might see the bits shifting). If reading a known device ID, verify the bytes on LA. This tight integration of writing code and using LA to debug it is exactly what embedded devs do when bringing up new hardware.
* **Day 263:** **JTAG Boundary Scan (Optional Advanced):** If curious, read about using JTAG for boundary scan (testing pins connectivity). This is more in the realm of hardware test engineering, but interesting. You likely won’t need to do this for the job, but understanding it can’t hurt.
* **Day 264:** **Revisit Internship Skills (Python, Qt, ROS2):** Consider how your current internship skills might integrate. For instance, Python could be used to write a test script for your device (e.g., a Python script on PC sending commands to your UART command parser to automate testing). Qt could be used to make a simple GUI to display data from your device. If time, make a very simple PySide6 GUI that reads serial data from your device and plots a value (e.g., the step count). This is extra, but leverages what you know and creates a cool demo.
* **Day 265:** **Soft Skill – English Technical Writing:** Since good English communication is required[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge), practice by writing an *English* blog post or article-style summary of one of your projects (perhaps the RTOS fitness tracker project). Explain the goal, how you designed it, and what you learned. Aim for clarity and conciseness. You could even publish it on a personal blog or LinkedIn. This not only practices communication but also is something you could show interviewers to demonstrate your enthusiasm and ability to communicate complex ideas.
* **Day 266:** **Soft Skill – Chinese Technical Discussion:** Equally, ensure you can discuss technical topics in Chinese (since the workplace is in Taiwan and Chinese fluency is expected). Try explaining one of your projects or a concept (like how FreeRTOS works) in Chinese, either in writing or to a friend. Use correct technical terms (e.g., RTOS = 即時作業系統, etc.). This will prepare you for any Chinese-language interviews or day-to-day explanations.
* **Day 267:** **Review Resume and Projects:** Start compiling your achievements and projects from this learning journey into your resume or a portfolio document. List the key skills you’ve acquired (C, C++, RTOS, ARM, etc.) and projects with one-liner descriptions (e.g., “Implemented a multi-tasking data logger on STM32 using FreeRTOS”). Make sure to emphasize the skills that match the job description (C/C++, embedded/RTOS, etc.)[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge).
* **Day 268:** **Mock Interview Q&A:** Prepare answers to common interview questions. For example: *“What was a challenging bug you encountered and how did you solve it?”* – you can talk about the hardfault debugging or race condition you fixed. *“Explain the difference between malloc and new”*, *“How do you ensure real-time tasks meet deadlines?”*, *“Describe a project where you used an RTOS.”* Practice answering these aloud in a structured way.
* **Day 269:** **Interview Algorithm Practice:** Some embedded interviews include simple algorithm or coding tests. Practice a few in C or C++: e.g., reverse a linked list, find if an array has duplicates, implement a basic state machine logic from spec, etc. Use pen and paper or a whiteboard if possible, since you might need to write code by hand. This keeps your general problem-solving sharp.
* **Day 270:** **Phase 5 Wrap-up:** Reflect on the tools and advanced skills you’ve learned. Ensure you can give examples of using them: *“I used a logic analyzer to debug I2C communication on a sensor project”*[*[10]*](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing)*, “I regularly used a JTAG debugger (ST-Link) to step through code and diagnose issues,”* etc. By articulating these, you’ll be ready to demonstrate that you meet the “knowledge of logic analyzer and JTAG” requirement in the job posting.

*(Now you have not only coding skills but also practical debugging experience – a hallmark of a well-rounded embedded engineer*[*[10]*](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing)*. The final phase will consolidate everything with a capstone project and final preparations for your job application.)*

## Phase 6: Capstone Project & Final Integration (Days 271–330)

**Focus:** Complete a **capstone project** that brings together all your skills (C, C++, RTOS, hardware interfacing, etc.) into one polished application. This project will serve as a showcase of your abilities and a confidence booster. We’ll allocate ample time to plan, implement, and refine the project. Afterwards, we’ll focus on final review and preparation to apply for the job (or any similar embedded role).

* **Day 271:** **Capstone Project Brainstorm:** Decide on a final project that is impressive yet achievable. Ideally, something related to a **Fitness GPS wearable or device**, since that’s the job domain. Ideas: a simplified *GPS fitness tracker* that logs “distance” (you could simulate GPS input), or a *cycling computer* that measures wheel rotations (simulate sensor input) and calculates speed, or a *smart pedometer* with step count and maybe calorie estimate. Pick one that excites you and seems feasible with your hardware.
* **Day 272:** **Capstone Project Requirements:** Write down the project’s requirements and features. For example: “Device will track steps and distance; user can start/stop logging with a button; data will be stored or displayed; must be low-power.” Defining requirements upfront is a good practice (and something you can discuss in interviews to show a systematic approach).
* **Day 273:** **Capstone System Design:** Design the system architecture. Decide which microcontroller to use (STM32 is a good choice for familiarity; ESP32 if you need wireless; Arduino if simplicity is needed, but STM32 likely). Determine if you will use FreeRTOS or bare-metal for this (depending on complexity – using FreeRTOS if multiple concurrent functions). Define the major modules or tasks: e.g., Sensor interface task, Logging task, UI task (if any display or LED feedback), Communication task (maybe to send data out). Draw a block diagram or task diagram.
* **Day 274:** **Hardware Setup:** Gather any hardware needed. If using GPS, do you have a GPS module? If not, perhaps simulate GPS by feeding NMEA sentences via UART from a PC. If counting steps, maybe use the accelerometer if you have one (or simulate by toggling a GPIO to mimic sensor). Ensure you have necessary wires, sensors, etc., or plan the simulation approach. Set up your dev board with any peripherals (e.g., connect an LED or LCD if you plan to use one, connect buttons).
* **Day 275:** **Project Initialization:** Start a fresh repository for the project. Set up the basic project structure in STM32CubeIDE (or PlatformIO, etc.). Get a “skeleton” running: e.g., FreeRTOS up (if using), or at least a blinking LED main loop to ensure the project is configured correctly. Commit this initial state to version control.
* **Day 276:** **Implement Sensor Module:** If your project involves a sensor (accelerometer for steps, or reading a simulated GPS input), work on that first. For a pedometer: use a timer to periodically read accelerometer data (or simulate a step count increment). For GPS: perhaps use a second USART to receive NMEA strings (you can send from PC or use an existing log of GPS data). Focus on getting raw data input working and verified (e.g., print raw data to console to confirm).
* **Day 277:** **Implement Data Processing Module:** Write the code that processes the raw data into useful info. For steps: implement a simple step detection (maybe count whenever acceleration on Z-axis crosses a threshold – or simply increment if simulating). For GPS: parse NMEA (latitude/longitude) and compute distance traveled (haversine formula or a simpler approximation). Ensure this runs in a reasonable time (use integer math if microcontroller lacks FPU and you want efficiency). Test the logic with sample data offline if possible (maybe in a Python script or small PC program) then integrate.
* **Day 278:** **Implement Storage/Logging Module:** Decide how to record the data. Options: store in flash memory (internal flash or an SD card if available), or transmit to PC for logging. If you have an SD card breakout and FATFS ready in STM32Cube, you could log to a file. Alternatively, simpler: store data in an array in RAM (limited duration) or send live data to PC via UART/USB (and capture with a terminal). Implement whichever is feasible. Ensure you handle memory carefully if storing in RAM (not to overflow).
* **Day 279:** **Implement User Interface Module:** If your project has any user interaction (buttons, LEDs, display), code that now. For a wearable, maybe a simple UI: a button to start/stop logging, an LED that blinks faster when logging, etc. If you happen to have a small LCD/OLED, try showing one piece of info (like step count). Otherwise, the UI might just be the PC printout or an LED pattern. Make it as interactive as possible – this will make your project feel complete and is good experience in handling user input debouncing, etc.
* **Day 280:** **Integrate Modules (FreeRTOS if used):** If using RTOS, set up separate tasks for these modules now and make sure they work together (you may have already done some of this). For example, one task reads sensor and updates global data (protected by mutex), another task checks for user input (button ISR giving semaphore), another logs data to storage. If not using RTOS, integrate in the main super-loop with state checks (that’s fine too, just ensure timing is managed, e.g., using timer interrupts for periodic actions). This integration is often the most challenging part – take it step by step and test after adding each piece.
* **Day 281:** **Testing – Functional:** Perform a functional test of the whole system in a basic scenario. E.g., simulate a short walk or some movement and see if steps accumulate, or feed a small GPS route and see if distance is calculated. Check that the start/stop button works if applicable, and data logging triggers as expected. Write down any bugs or issues noticed.
* **Day 282:** **Debugging and Issue Fixing:** Resolve the issues found in testing. For instance, if the step count seems off, tweak the detection threshold or algorithm. If data logging missed some entries, check if queue overflows or increase task priority. Use your debugger or logic analyzer to troubleshoot timing issues (e.g., is the sensor read happening as scheduled?). This might take a couple of iterations – that’s normal.
* **Day 283:** **Performance Optimization:** Check if the system performance is adequate. For example, if using RTOS, check CPU utilization (maybe by measuring idle task time or using a debug tool). Ensure no task is starving others. If the loop or tasks need to run faster, consider optimizations (like using DMA for sensor reading if applicable, or increasing tick rate if more responsiveness needed). For a simulated project this might not be critical, but it’s good to think in terms of efficiency especially for battery-operated devices.
* **Day 284:** **Power Consumption Consideration:** If you have the means, measure the device’s current draw (with a multimeter on a low-range if possible, or a USB power monitor if running off USB). If not, conceptually analyze where power could be saved: e.g., could you put the MCU to sleep when idle? If using FreeRTOS, is tickless idle on? Could you lower clock frequency when high performance isn’t needed? Document a couple of these thoughts. This aligns with building products that can run on battery for long, which is key in wearables.
* **Day 285:** **Final Feature Tweaks:** Add any “nice-to-have” features if time allows. For instance, maybe add a second button to reset data, or an LED indication for GPS signal lock (if simulating, perhaps after a delay turn an LED on to indicate “GPS lock”). These small touches can make your project feel complete and also show your attention to user experience. Just be mindful not to destabilize your working system; keep them simple.
* **Day 286:** **Extended Testing – Edge Cases:** Test edge cases: what if sensor data is noisy (maybe simulate rapid random changes), does your algorithm break? What if the user presses start/stop rapidly? Does the system handle it or does it glitch? Try to break your system now and see how it copes. It’s better you find any weakness than an interviewer! Fix any discovered issues or at least note them with a plan on how you’d address them in a real product (shows awareness).
* **Day 287:** **Documentation – User Manual & Dev Notes:** Write two short documents: (1) A “user manual” (even if just a page) for the device – describing how to use it, what the indicators mean, etc. And (2) Developer notes – describing module design and maybe a brief how the code is structured. This practice helps in organizing your thoughts and could be useful to show you can create clear documentation (Garmin being a large company likely has documentation standards).
* **Day 288:** **Prepare Demo (if possible):** If you have the opportunity, prepare to *demo* this project as if to an interviewer. For example, you might record a short video of it working (explaining in English what’s happening), or have your board and PC ready to show output. Even if you don’t actually present it, thinking through a demo scenario helps you ensure the project is presentable and you can succinctly explain it.
* **Day 289:** **Reflect on Project Learnings:** Write a post-mortem: what went well, what was hard, what you’d do differently if you had more time. For instance, “Using FreeRTOS made dividing tasks easier, but I had to be careful with shared data (used mutexes to avoid race conditions). I learned a lot about integrating sensor data and timing.” This reflection can be gold in interviews – it shows maturity and continuous learning.
* **Day 290:** **Review Against Job Requirements:** Look again at the Garmin job description[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge). Make a checklist of each requirement and bonus skill, and ensure you have something to say or show for each:
* C/C++ programming: (you have multiple projects and this capstone demonstrating that)
* Fitness GPS product design: (your capstone is basically that in miniature, discuss how you simulated or would handle actual GPS/sensors)
* English communication: (you’ve practiced documentation and can converse fluently)
* Computer architecture & ARM: (you can mention your Phase 3 deep dive, knowledge of Cortex-M, etc.)
* Embedded system/RTOS: (you built an RTOS project, clearly you have experience)
* Logic analyzer, JTAG: (you used them in Phase 5, have examples of bugs you solved with them)
* Other OS: (you have familiarity with Linux/ROS2 from internship, plus you learned some context about where RTOS vs Linux is used)  
  This preparation ensures you can confidently address each point in an interview or cover letter.
* **Day 291:** **Compile Portfolio and Code Repos:** Organize all your code and projects from this year on a platform like GitHub or GitLab (if you haven’t been doing so continuously). Ensure they have README files explaining what they are. This shows professionalism. Even if some code is rough, a well-written README can make a positive impression. Specifically, have the capstone project repo in good shape (comments, instructions to run).
* **Day 292:** **Polish Resume (English):** Update your resume in English to highlight relevant skills and projects. Add your Delta internship (mention Python/ROS2 work briefly) but focus on embedded skills you gained: e.g., “Developed embedded firmware for ARM Cortex-M microcontrollers using C and FreeRTOS,” “Experience with debugging using JTAG and logic analyzers,” etc. Quantify or give context where possible. Keep it concise, one page if you can, and clear.
* **Day 293:** **Polish Resume (Chinese):** It might be useful to have a Chinese version of your resume since the job is in Taiwan. Translate or rewrite key points in Chinese, ensuring proper technical terms (like “即時作業系統” for RTOS, “邏輯分析儀” for logic analyzer). Have a mentor or friend review if possible for language accuracy.
* **Day 294:** **Cover Letter (if needed):** Draft a cover letter or email body for your job application. Explain why you’re interested in the position and how your year of intensive self-study makes you a great fit. Mention the projects you completed as evidence of your skills and passion. Keep a professional but enthusiastic tone. (English is usually fine for cover letters at multinationals like Garmin, but if you write in Chinese, ensure it’s well-written.)
* **Day 295:** **Mock Interview (Technical):** Arrange a mock interview if possible. Have someone ask you technical questions (in English or Chinese) about your projects and general knowledge. If no one available, do it yourself: simulate answering a question as if in an interview. Pay attention to structure: e.g., for “Tell me about a project you’re proud of,” have a clear storyline (problem, solution, your role, result). For technical stumpers (if any), practice how to think aloud logically.
* **Day 296:** **Mock Interview (HR/Behavioral):** Don’t forget general questions: *“What are your strengths and weaknesses?”*, *“Why Garmin/why this role?”*, *“Describe a time you solved a difficult problem,”* etc. Prepare answers for these too, highlighting soft skills like teamwork, communication, and perseverance learned during this journey. For example, discuss how you kept a disciplined schedule for 365 days – it shows dedication and self-motivation.
* **Day 297:** **Interview Day Prep Checklist:** Create a checklist for the day of interview (if/when it gets scheduled): items like “Board and laptop to demonstrate project (if in-person and appropriate)”, “copies of resume”, “notebook with key points”. Also, prepare a few thoughtful questions to ask the interviewer about the role (e.g., “What development process does the team use? Agile? How are new features tested on devices?”). Good questions show your genuine interest.
* **Day 298:** **Knowledge Refresh Blitz:** Go through all your summary notes and cheat-sheets from previous phases (C cheat-sheet, ARM notes, RTOS notes, etc.). Spend some time on each to refresh in your mind. This is like final exam prep, consolidating the vast amount of information you’ve gathered. By now, much is second nature, but a quick refresh ensures terms and details are on the tip of your tongue if asked.
* **Day 299:** **Rest and Recover:** This journey has been long and intense. Take a day mostly off to relax your mind. Maybe just light review of something you enjoy (watch an engineering video or read news about tech), but nothing too strenuous. A fresh mind will help you perform better in actual job application steps.
* **Day 300:** **Apply for the Job:** If all has gone well, today is a good day to submit your application to the Garmin position (or multiple positions including this one). Tailor each application if needed. Then celebrate the fact that you prepared thoroughly! Remember to follow any application instructions (the listing said to apply on Garmin’s website[[25]](https://www.104.com.tw/job/7so7n#:~:text=%E5%85%B6%E4%BB%96%E6%A2%9D%E4%BB%B6) – do that).

*(The remaining 65 days can be used for continued practice, learning, or handling interview processes. Below is a plan for those days focusing on continuous improvement and fallback plans.)*

## Phase 7: Continuous Learning & Interview Season (Days 331–365)

**Focus:** Use the final days to continue learning in areas of interest or weakness, work on any feedback from applications/interviews, and ensure you stay sharp and motivated. Even after applying, the learning shouldn’t stop – the field evolves, and so will you.

* **Day 331:** **Feedback Analysis:** If you’ve started getting responses or interviews, analyze any feedback. Did an interviewer stump you on a question? Research and fill that gap today. If you haven’t had interviews yet, review your application materials or ask a mentor for feedback on resume/cover letter. Continuous improvement.
* **Day 332:** **Deep Dive: Specific Technology:** Pick one technology relevant to Garmin products to dive deeper. For example, **ANT+ protocol** (used in Garmin for connecting to sensors). Learn how it differs from BLE. This is niche, but showing knowledge of Garmin-specific tech could impress. If not ANT, maybe learn about **Garmin’s Connect IQ** (their app platform) just for context.
* **Day 333:** **Project Euler / Coding Challenges:** Keep doing some coding challenges (in C/C++) to maintain problem-solving agility. Sites like LeetCode, HackerRank, etc., can provide small problems that ensure you don’t get rusty on coding under pressure. Do one or two today.
* **Day 334:** **Explore Embedded Community Projects:** Find an open-source project or library in embedded systems to study or contribute to. For instance, the FreeRTOS kernel is open-source – browse its code to see if you understand it better now. Or a smaller project on GitHub like an Arduino library for a sensor – read the code and see if you could write something similar. This practice of reading others’ code improves your own coding and gives perspective.
* **Day 335:** **Soft Skills – Presentation:** Practice explaining a complex concept to a non-expert (maybe a friend or family member). For example, try to explain how GPS determines location in simple terms, or how a microcontroller differs from a PC. This can improve your ability to communicate clearly, which is valuable in any job (and might be tested in interviews too).
* **Day 336:** **Research Latest Trends:** Spend time reading recent articles or news in the embedded field. E.g., new microcontroller releases, new versions of C/C++ standards (C++20/23 features?), advancements in RTOS or IoT. Having knowledge of current trends can be good conversationally and shows you are passionate beyond just this training.
* **Day 337:** **Language Practice – Technical English:** Read some technical documentation or a whitepaper in English that’s challenging. Maybe an ARM technical manual or a research paper on wearable technology. This keeps your technical reading comprehension strong (helpful if you need to quickly learn something on the job from docs).
* **Day 338:** **Language Practice – Technical Chinese:** Similarly, find a technical article or forum in Chinese about embedded systems. Practice reading it and even writing a response or summary. This ensures your technical Chinese vocabulary stays sharp – useful when working with local colleagues or documentation.
* **Day 339:** **Plan B Opportunities:** Identify other companies or positions similar to Garmin’s that you could apply to if needed. For example, other wearable tech companies or any embedded systems roles in Taiwan that excite you. Make note of their requirements and see if there’s any skill you wish to quickly brush up (though by now, you’ve covered a lot!). Diversifying applications can increase your chances.
* **Day 340:** **Hackaday or Contest Project:** If you want an extra challenge, see if there are any ongoing contests or community challenges (like Hackaday Prize or a local maker competition). Joining one with a small project can be motivating and also looks good (shows passion). Even if you don’t fully join, browsing others’ project ideas might inspire you further or give talking points.
* **Day 341:** **Teach/Share:** Try to teach a mini-lesson on something you learned to someone else (or simulate it). For example, give a short presentation on “Introduction to FreeRTOS” or “Tips for C++ in Embedded” to an imaginary junior engineer or a friend. Teaching is a great way to reinforce your knowledge and highlight any areas you need to clarify for yourself.
* **Day 342:** **Interview Follow-ups:** By this time, if you had interviews, send any follow-up or thank-you notes as appropriate (professionally and briefly). Reflect on how you did, and list any questions you found difficult. If you’re still waiting for interviews, use this day to go over your stories and strengths one more time.
* **Day 343:** **Build Something Fun:** After so much structured learning, do a one-day fun build. Maybe program your Arduino or ESP32 to play a tune on a buzzer, or use the ESP32 to blink lights via a web server (combining IoT and embedded). Something outside the strict curriculum – to remind you that engineering is fun and creative, not just for passing interviews.
* **Day 344:** **Evaluate and Iterate:** Think about how you learn best and what this 344-day journey has taught you about that. If there are any topics you still feel weak in, plan how to address them even beyond day 365. Maybe you’ll continue with a weekly schedule after landing a job to learn new things (like learning Rust for embedded or diving deeper into machine learning on microcontrollers if interested).
* **Day 345:** **Networking:** Reach out to peers or professionals. Maybe connect with someone on LinkedIn who works at Garmin or similar companies (you can introduce yourself politely, expressing interest in the field – not directly asking for a job, but for any advice). Networking can sometimes open doors or at least give insight. Use today to draft a couple of messages or posts (in English or Chinese professional circles).
* **Day 346:** **Revisit University Knowledge:** Look back at your Master's coursework or projects. Are there any theoretical pieces you can now see in a new light thanks to practical experience? For example, if you took a course on operating systems or digital signal processing, you might now appreciate those concepts more. It can be satisfying to bridge theory and practice, and you could mention these insights in conversation, showing you have both academic and practical understanding.
* **Day 347:** **Time Management and Learning Skills:** Consider the process you followed for a year – what worked, what didn’t. Write down strategies that helped you (like “consistency is key, even 2 hours daily can achieve a lot” or “hands-on projects cemented my understanding”). This meta-knowledge is useful for your future self as you continue learning on the job.
* **Day 348:** **Celebrate Progress:** Look back to Day 1’s knowledge vs now. Celebrate how far you’ve come! Maybe treat yourself to a new dev board or gadget as a reward (if budget allows) – something you can tinker with in the future (like a new STM32H7 board or a smartwatch dev kit). This can rekindle excitement for the next challenges.
* **Day 349:** **Latest Garmin Products Research:** Check Garmin’s latest product lineup in fitness/wellness. Read about their features, any press releases on technology used. This not only prepares you for potential interview talk (showing you know the company’s products) but also might reveal what skills are most relevant (e.g., if a new watch has music playback, maybe they run an RTOS + a small file system, etc.). It might spark last-minute focused study on something.
* **Day 350:** **Interview Simulation in Chinese:** Just in case you face a part of interview in Chinese (sometimes companies do a casual chat in the local language even if technical in English), practice describing your projects and answering basic questions in Chinese once more. You want to be as fluent in expressing your technical journey in Chinese as you are in English.
* **Day 351:** **Check Application Status:** If by now you haven’t heard back, consider following up on your job application politely (if it’s been a while since submission). Reaffirm your interest briefly and maybe mention any new project or learning you’ve done since applying (shows you keep growing). If you have other opportunities in progress, manage those accordingly.
* **Day 352:** **Plan Transition to Job Life:** Think ahead – once you get the job, how will you continue learning while working? Perhaps note a few things you’d like to get better at which you can pursue on the job (like learning the specific tools they use, or improving at writing production-quality code with peer reviews). This shows forward-thinking and will help you hit the ground running.
* **Day 353:** **Wrap-up any Unfinished Learning:** If there’s any topic you wanted to explore but skipped due to time, use the last stretch to do it. Maybe you wanted to try using a **logic analyzer on a communication bus** you didn’t get to, or try **writing a simple Linux device driver** for practice, or learn another RTOS like Zephyr. Even a cursory exposure now can make it easier to dive deeper when needed.
* **Day 354:** **Community Involvement:** Consider writing a summary of your learning journey on a forum or personal blog. It could inspire others. Plus, teaching others (even by sharing your story and tips) reinforces what you know. This could also incidentally get you noticed in the community (though that’s a bonus, not the goal).
* **Day 355:** **Thank Your Supporters:** If anyone (professors, friends, online communities) helped you along the way, take time to thank them. Send a short message appreciating their role. Not only is this good practice, but it keeps your network warm – those people will be rooting for you.
* **Day 356:** **Final Review of Everything:** Do one last comprehensive review of key points: C/C++ concepts, RTOS API, ARM details, project summaries. This is essentially cramming the night before an exam but in a structured way – maybe use your notes and make sure nothing vital has slipped your mind. At this stage, you likely won’t forget major things, but a quick pass might catch a detail or two you want to remember.
* **Day 357:** **Relaxation and Mental Prep:** On the day before an interview or as you near the end, ensure you get good rest. You’ve done the work; confidence is now your friend. Perhaps do a light coding puzzle to keep your mind active but nothing too draining. Trust your preparation.
* **Day 358–364:** **(Buffer Days)** – Use these as needed depending on your interview/job timeline. They can serve as contingency (for example, if an interview gets scheduled later, or if you decide to learn one more mini-skill). Otherwise, enjoy a well-deserved break or start exploring new horizons at a comfortable pace.
* **Day 365:** **Reflection and Next Steps:** It’s been one year! Reflect on this structured learning journey – how it has equipped you not just for this job, but for a career in embedded systems. Identify the next big goal (maybe excelling in the new job, or aiming for a specialization). Make a broad plan for continuing growth (perhaps a list of books to read or projects to do over the next year, but on a more relaxed schedule). The end of this 365-day plan is really the beginning of your professional journey. 🎓👷‍♂️

**Conclusion:** By following this day-by-day plan, you’ve methodically built up the competencies required for an Embedded Systems Software Engineer (Fitness products) – mastering C and C++ separately[[4]](https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/#:~:text=C%2B%2B%20is%20a%20useful%20skill,a%20mandatory%20skill%20in%20embedded), gaining microcontroller and RTOS experience[[23]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Real), and practicing with industry-standard tools like logic analyzers and JTAG debuggers[[10]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing). You’ve also completed practical projects (blinking LEDs to a full RTOS-based tracker) that mirror real product development, aligning with the job’s focus on fitness GPS devices[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge). Throughout, you strengthened your problem-solving and communication skills in both English and Chinese. With this intensive preparation, you should feel confident in interviewing for the role and starting your career as an embedded software engineer. Good luck – and remember that learning and improving is a lifelong journey, but you have proven you have the dedication to succeed! 🚀

**Sources:** The structure and topics of this plan were informed by industry guidelines and roadmaps for embedded engineering[[26]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=7%20fundamental%20Embedded%20Software%20Engineer,skills%20and%20traits)[[1]](https://www.104.com.tw/job/7so7n#:~:text=1,Other%20OS%20related%20knowledge), expert advice on prioritizing C before C++[[4]](https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/#:~:text=C%2B%2B%20is%20a%20useful%20skill,a%20mandatory%20skill%20in%20embedded), and known key skills for embedded developers (C/C++, microcontroller/ARM knowledge, RTOS, debugging tools, etc.)[[27]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=The%20best%20skills%20for%20Embedded,Level%20Programming)[[28]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing). These ensure the plan covers all critical areas expected by employers in this field.

[[1]](https://www.104.com.tw/job/7so7n" \l ":~:text=1,Other%20OS%20related%20knowledge) [[25]](https://www.104.com.tw/job/7so7n#:~:text=%E5%85%B6%E4%BB%96%E6%A2%9D%E4%BB%B6) 軟體工程師 (嵌入式系統\_ 健身產品)\_ 林口｜GARMIN\_台灣國際航電｜桃園市－104 人力銀行

<https://www.104.com.tw/job/7so7n>

[[2]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=%E2%9A%92%EF%B8%8F%20Projects) [[6]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=%E2%80%8D%20VS%20Code%20Extensions) [[9]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=,Embedded%20C%20Coding%20Standard) [[20]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=) [[21]](https://github.com/m3y54m/Embedded-Engineering-Roadmap#:~:text=,Refactoring%20Guru) GitHub - m3y54m/Embedded-Engineering-Roadmap: Comprehensive roadmap for aspiring Embedded Systems Engineers, featuring a curated list of learning resources

<https://github.com/m3y54m/Embedded-Engineering-Roadmap>

[[3]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=C%20Programming) [[7]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Low) [[8]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Communication%20Protocols) [[10]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing) [[11]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Our%20Embedded%20C%20Online%20Test,libraries%2C%20crucial%20for%20firmware%20development) [[12]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Microcontroller%20Knowledge) [[23]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Real) [[26]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=7%20fundamental%20Embedded%20Software%20Engineer,skills%20and%20traits) [[27]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=The%20best%20skills%20for%20Embedded,Level%20Programming) [[28]](https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/#:~:text=Debugging%20and%20Testing) Skills required for Embedded Software Engineer and how to assess them

<https://www.adaface.com/blog/skills-required-for-embedded-software-engineer/>

[[4]](https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/#:~:text=C%2B%2B%20is%20a%20useful%20skill,a%20mandatory%20skill%20in%20embedded) [[5]](https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/#:~:text=Image%3A%20Profile%20Badge%20for%20the,Commenter) Which language is better for learning embedded systems, C or C++ : r/embedded

<https://www.reddit.com/r/embedded/comments/18r00kc/which_language_is_better_for_learning_embedded/>

[[13]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=) [[14]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Despite%20the%20features%20that%20C%2B%2B,is%20NO%20dynamic%20memory%20allocation) [[15]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Another%20common%20argument%20against%20C%2B%2B,to%20make%20these%20mechanisms%20possible) [[16]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Textbooks%20examples%20of%20C%2B%2B%20code,C%2B%2B%20in%20Niall%20Cooling%E2%80%99s%20article) [[17]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=C%2B%2B%20Standard%20Library%20offers%20a,style%20arrays) [[18]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=The%20above%20snippet%20of%20C,into%20the%20following%20C%2B%2B%20code) [[19]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Another%20common%20argument%20against%20C%2B%2B,some%20extra%20information%20to%20binary) [[22]](https://www.embeddedrelated.com/showarticle/1532.php#:~:text=Historically%20speaking%2C%20C%2B%2B%20started%20as,and%20they%20are%20well%20documented) Modern C++ in Embedded Development: (Don't Fear) The ++ - Amar Mahmutbegovic

<https://www.embeddedrelated.com/showarticle/1532.php>

[[24]](https://docs.wokwi.com/zh-CN/guides/logic-analyzer#:~:text=%E9%80%BB%E8%BE%91%E5%88%86%E6%9E%90%E4%BB%AA%E4%BD%BF%E7%94%A8%E6%8C%87%E5%8D%97%20,%E2%80%9D%E3%80%82%20%E6%8E%A5%E4%B8%8B%E6%9D%A5%EF%BC%8C%E5%B0%86%E8%A6%81%E8%AE%B0%E5%BD%95%E7%9A%84%E4%BF%A1%E5%8F%B7%E8%BF%9E%E6%8E%A5%E5%88%B0Logic%20Analyzer%E5%BC%95%E8%84%9A%E3%80%82%E4%BE%8B%E5%A6%82) 逻辑分析仪使用指南 - Wokwi Docs

<https://docs.wokwi.com/zh-CN/guides/logic-analyzer>