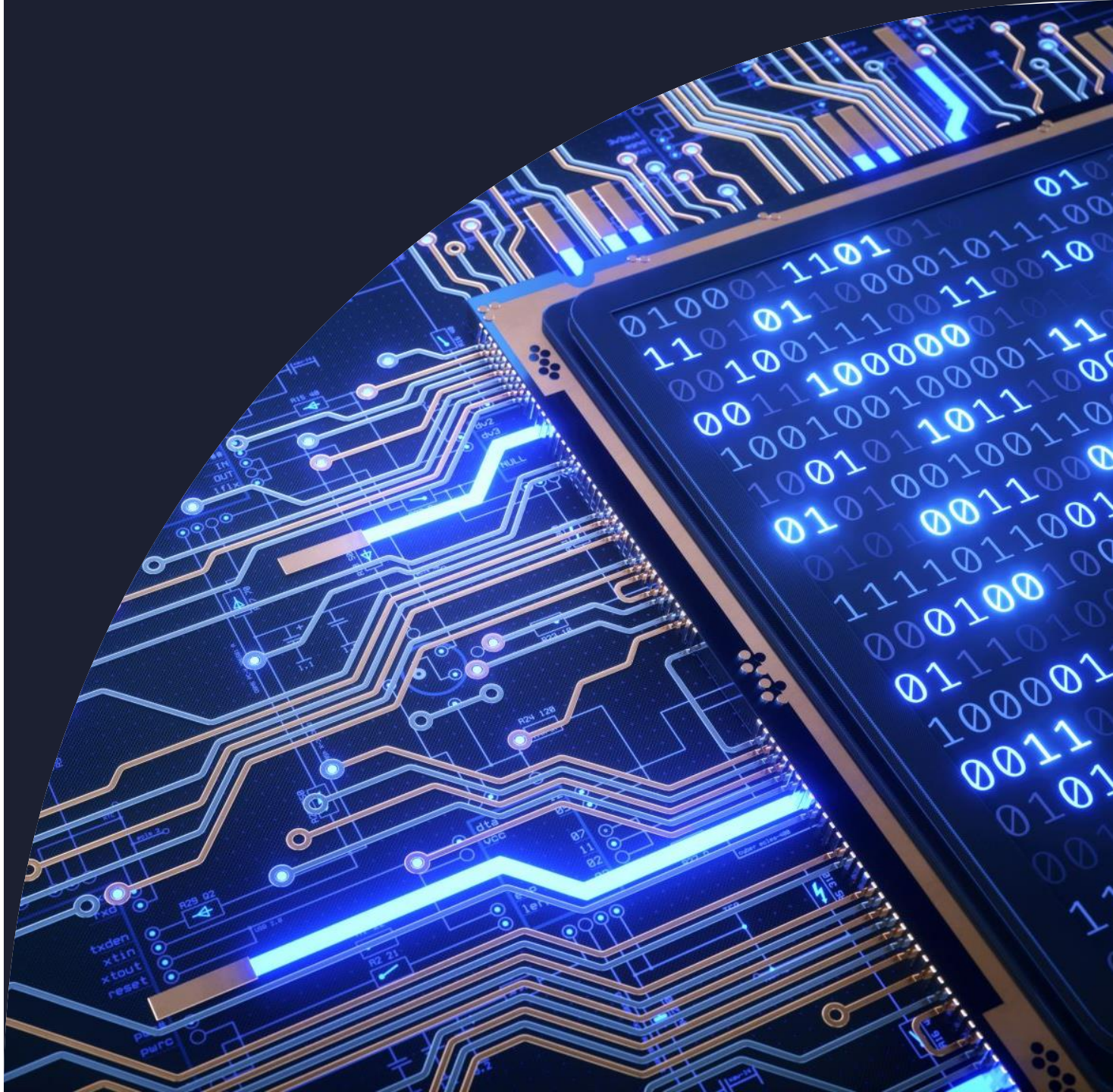
Embedded Software Best Practices

BY: MAHESH MURTY

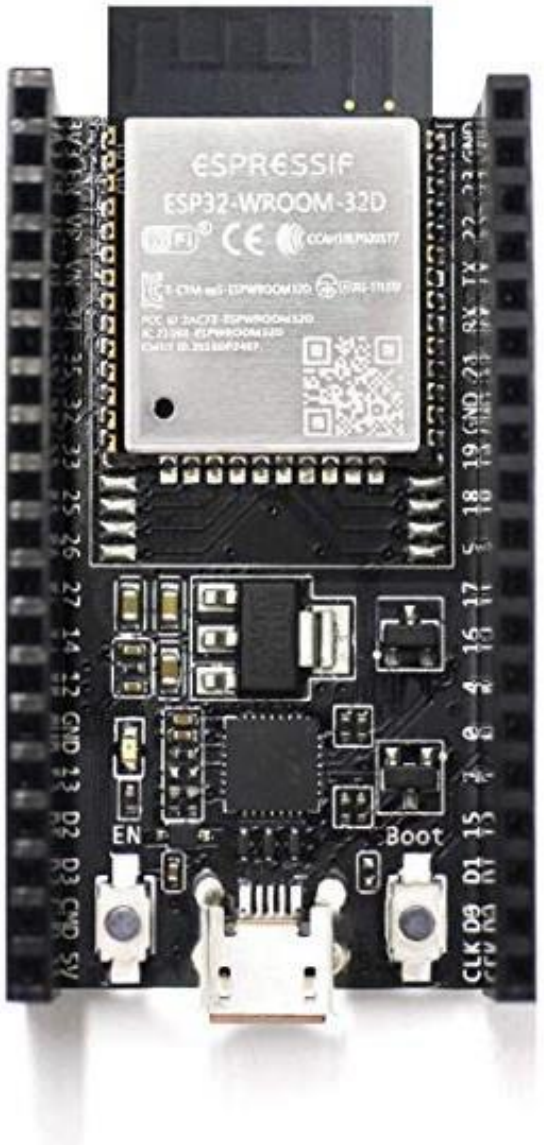


Today’s Agenda

- Introduction to ESP32
  - Features
  - The purpose of two cores.
  - Different Memories
  - Boot process
- Recommended practices to build robust embedded software

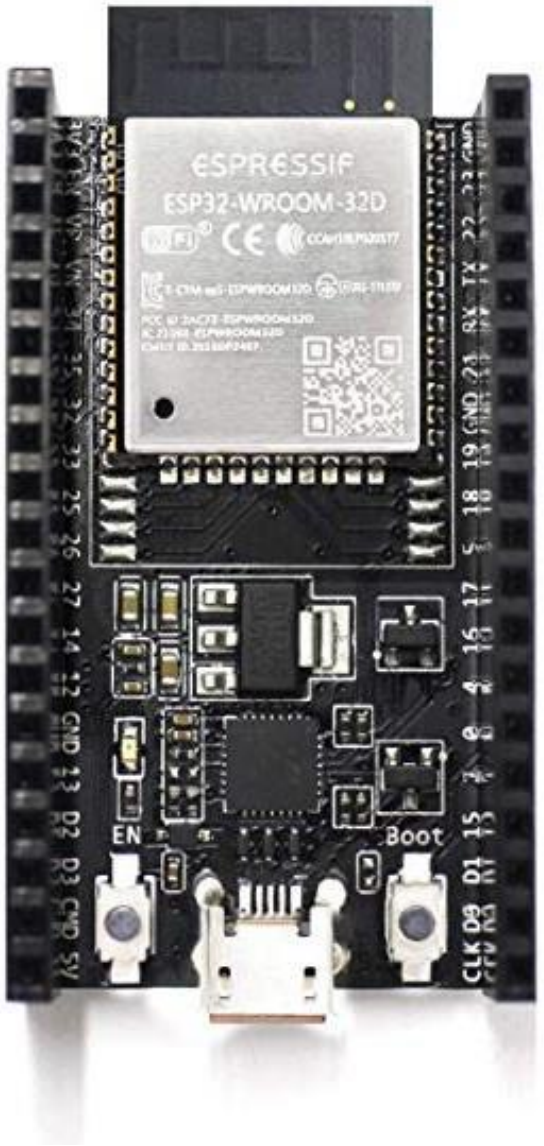
Pre-requisites

- Arduino Framework & IDE / Any other equivalent platform.
- C / C++
- Taken a course on OS is a plus.



Know your hardware well

- Manufacturer: Espressif Systems
- MCU Part No: ESP32-D0WDQ6
- Features:
  - XtensaDual Core Processors.
  - Max Clock Speed: 160 or 240 MHz.
  - Integrated BLE + WiFi+ Ethernet MAC.
  - ULP Co-Processor.
    - Can run in deep sleep mode when main SOCs are sleeping.
    - Can access peripheral devices, internal sensors and RTC registers.
  - Ultra Low Power Solution
    - Modem sleep mode: 3 – 10 mA.
    - Light sleep mode, CPU paused, RTC & ULP-Co Proc Active: 0.8 mA.
    - Deep sleep mode, CPU off, RTC & ULP-Co Proc Active: 0.15 ma if co proc on, 10uA otherwise.
    - Hibernation mode, everything off only RTC timer and RTC GPIOs active: 2.5uA.



Know your hardware well

- Core Peripherals:
  - Pin Muxing.
  - DMA
  - 12 bit –18 channel SAR ADC.
  - 2 x 8 bit DAC.
  - 4 x SPI
  - 2 x I2C
  - 2 x I2S
  - 3 x UART
  - 1 x SDIO
  - 1 x QSPI
  - DSP instructions + DSP library.

What do I mean by a robust Embedded Software?

- Well tested.
- Version Controlled.
- Portable.
- Flexible.
- Well architected.
- Immune to common / known problems.
- Should run 24 x 7 with minimum down time.
- Easy to debug.

Best Practice #1

Select your IDE wisely.

IDE & Framework Selection

ESP IDF

ARDUINO

NODE MCU

MicroPython

IDE & Framework Selection

- Choose an IDE which has at least syntax highlighting and code indexing support.
- Debugging support is a plus.
- Verify whether your framework has the drivers for your peripherals.
- VSCode is a good option.
  - Works also with Arduino framework using PlatformIO.

Best Practice #2

Make use of platform / compiler independent data types.

Compiler independent data types

- Some compilers treat int as a 16-bit datatype and some treat it like a 32-bit datatype.
- Use: uint8_t, uint16_t, uint32_t ….
- Defined in the header stdint.h

Best Practice #3

Be well versed with use of bitwise operations and number formats.

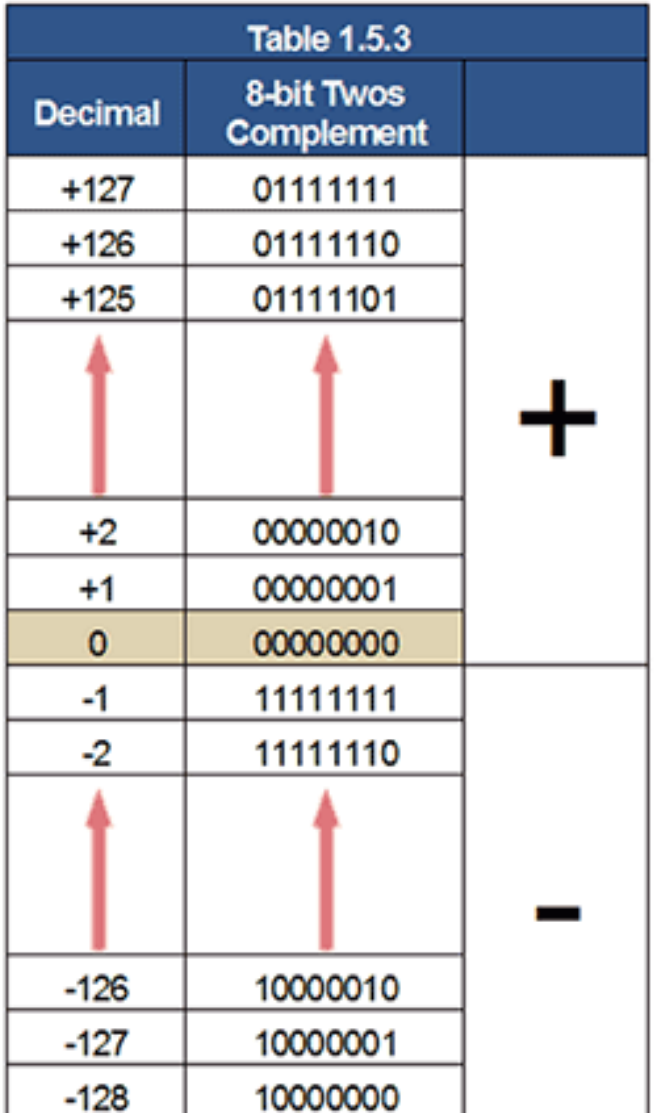


Table 1.5.3		
Decimal	8-bit Twos Complement	
+127	01111111	+
+126	01111110	
+125	01111101	
↑	↑	
+2	00000010	
+1	00000001	
0	00000000	
-1	11111111	-
-2	11111110	
↑	↑	
-126	10000010	
-127	10000001	
-128	10000000	

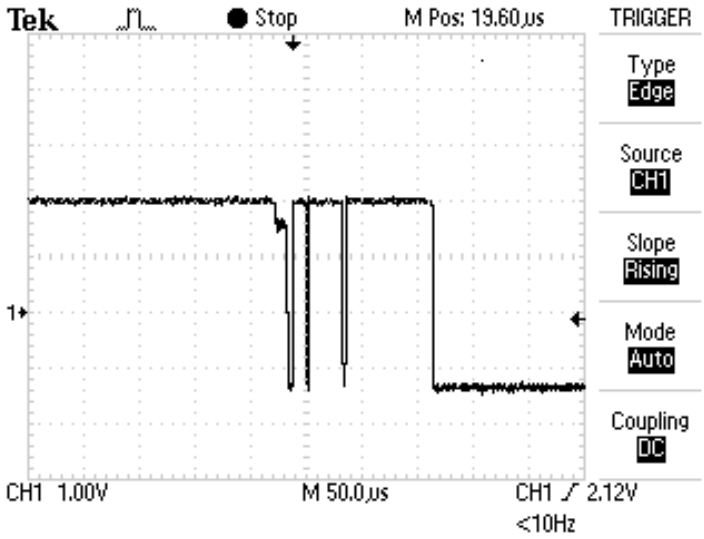
Bitwise operators & Number Formats

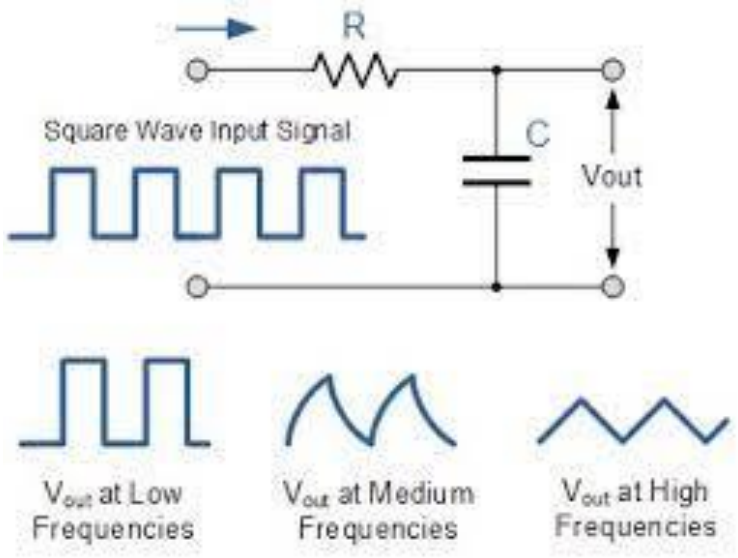
- Most useful one are &, |, ~, << and >>.
- 2's complement based number representation.
- Concept of sign extension. (When used wisely, this is automatically handled by compiler).
- Most used when communicating with an external peripheral like a temperature sensor, ADC or an energy meter.

Best Practice #4

Always debounce your switch inputs.

Switch debouncing





- Always use external pull up resistors.
- Use an RC glitch filter on the switch pin.
  - There is a special hardware glitch filter on Microchip's SAM, NXP's I.MXRT MCUs.
- Implement a software based debounce algorithm.
  - My general practice: Loop until switch is released.

Best Practice #5

Always use edge triggered interrupts.

Edge triggered interrupts

- Problem with level triggered interrupts:
  - Chances of catching multiple triggers because of timing.
  - Continuously triggered if the source device is faulty or not connected.
  - Useless for applications like pulse counting.
- Uses of level triggered interrupts:
  - Safety related critical inputs.

Best Practice #6

Keep your ISRs short

Short ISRs

- Problems with long ISRs:
  - Halts the complete system so long as the ISR is executing.
  - Hard to achieve deterministic / real time behavior.
- Avoid calling functions with large body / which consume huge stack in ISRs.
- Never use delay routines in the ISR.
- TIP: Always declare global variables shared with ISRs as volatile.

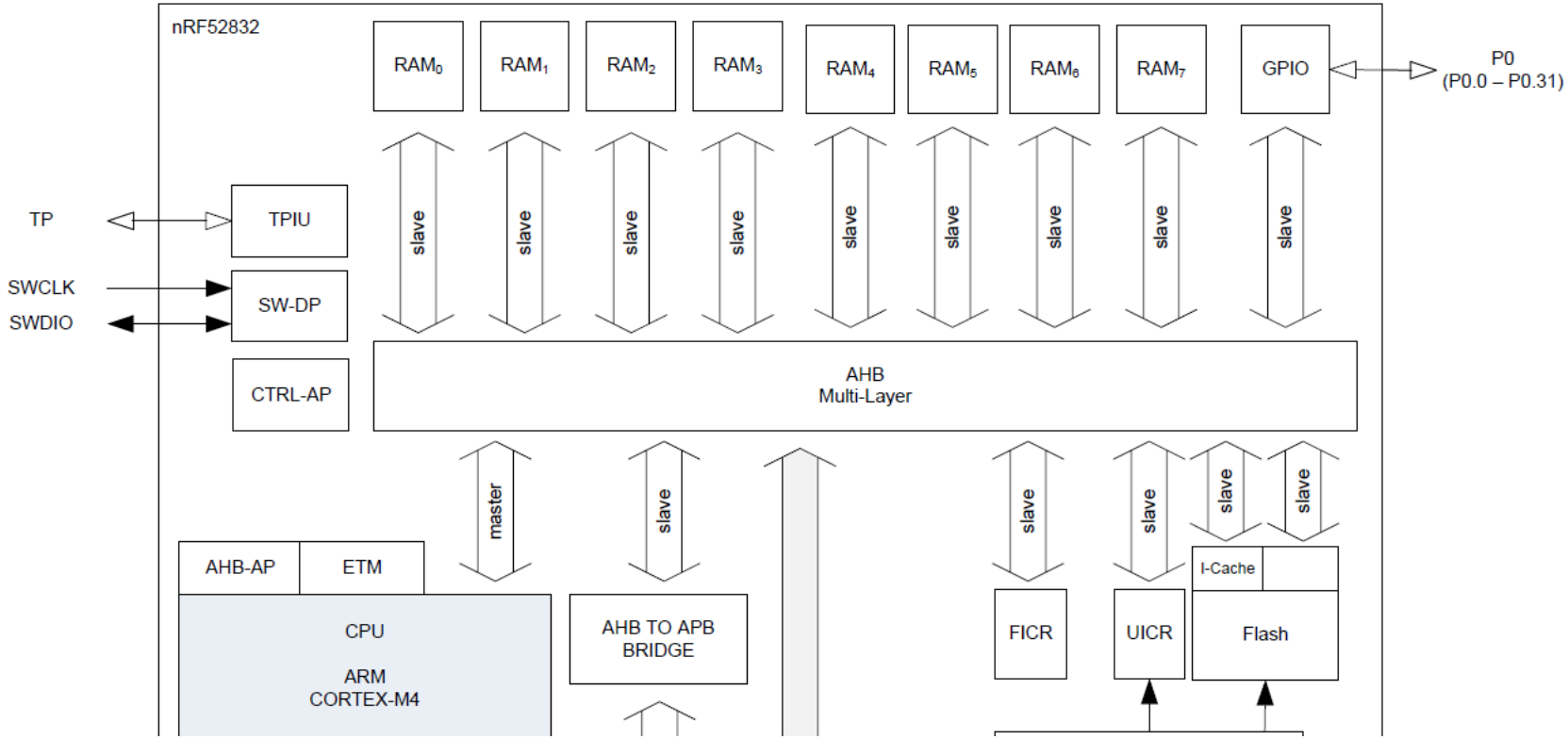
Best Practice #7

Understand your processor’s memory map well

Memories on ESP32

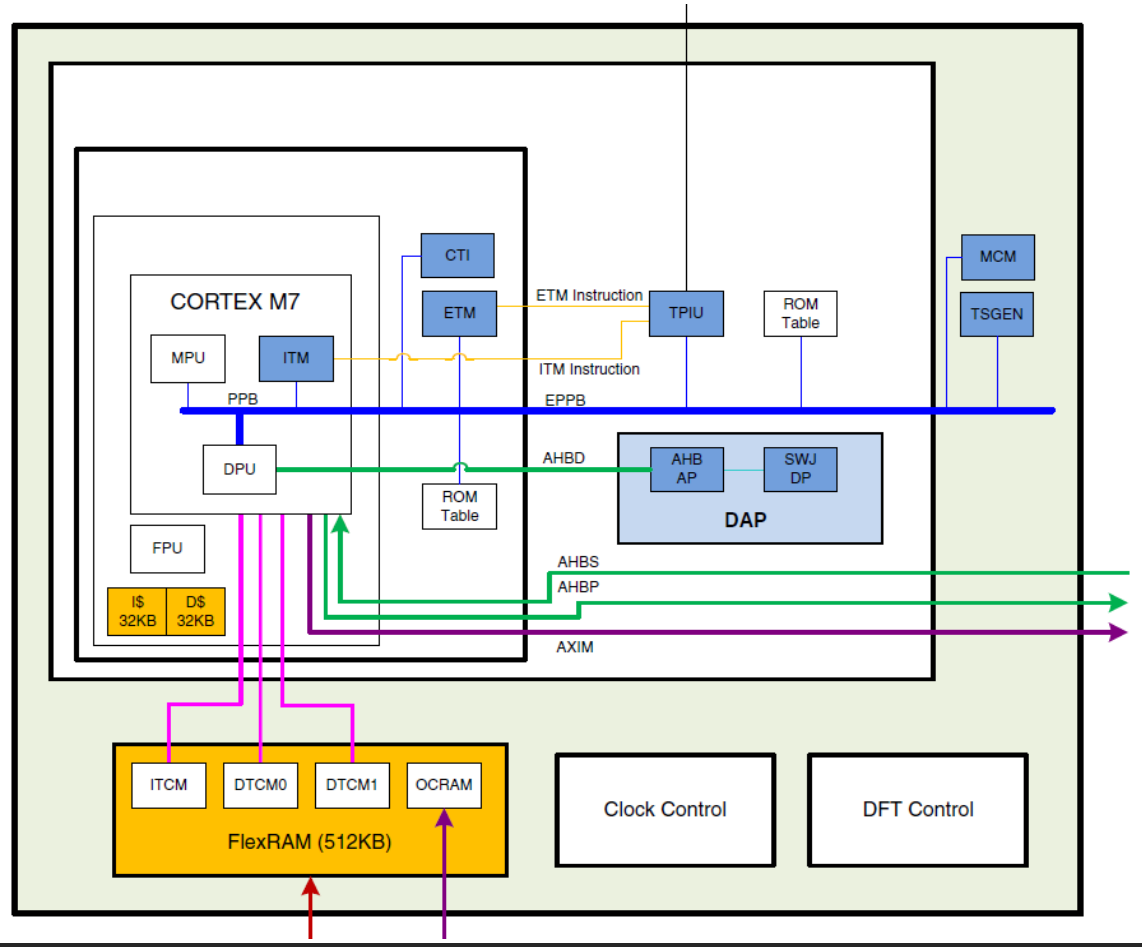
- There is 520KB of SRAM:
  - All variables go to RAM.
  - All ISRs go to RAM.
  - All the functions which are to be called from ISRs go to RAM.
  - All FreeRTOS functions go to RAM.
  - So check how much is left for the application.
- There is 16KB of RTC RAM.
  - Consumes very low power.
  - Can be battery backed.
  - ULP co processor code resides here.
  - Can retain data even between processor resets / when processor enters deep sleep.
- There is no internal flash on ESP32.
  - There is an external QSPI flash.
    - Available flash sizes 2 / 4 / 8 / 16 MB.
  - SPI port 0 & 1 are dedicated for flash access and cannot be used by the application.
  - Write cycles to flash are limited so avoid using it for data logging purposes.
- There is EFUSE Memory:
  - This is OTP memory.
  - Used to store MAC-IDs, Encryption Keys, certificates.

Special Memory Features on nrf52832



- RAM is divided into multiple RAM banks.
- Power is gated for each RAM bank.
- Any RAM bank can be configured as an retention RAM.
- Retention RAM can retain data between resets and wakeup from deep sleep.

Special Memory Features on NXP's I.MXRT cores.



- Flex RAM Memory of 512KB
- Flex RAM can be configured at runtime as:
  - OCRAM – Slowest for data storage
  - ITCM – Instruction RAM.
  - DTCM – Fast data RAM.
- ITCM
  - Has 64 – bit non cached access.
  - Used to store ISRs and instructions where speed is required.
  - Access time: 1 CPU clock cycle.
- DTCM:
  - Has 2 x 32 – bit non cached access.
  - Used to store DMA and high speed access data.
  - Access time: 1 CPU clock cycle.
- OCRAM:
  - Cached access.
  - Used to store data.
  - Access time: 3 CPU clock cycles.
  - Consumes least amount of power and can be used as retention RAM.

Best Practice #8

Architect your software well before you start implementing.

Software Architecture

- Make decisions before hand what logic goes on hardware and what in software
  - Sometimes it makes sense to go with a dedicated hardware solution.
    - Examples: 1-Wire Protocol handling, Encoder Pulse Counting, Energy Metering.
    - Commonly used peripheral: DS18B20.
- List down features: Mandatory, Optional, Future scope.
- Break your logic into several independent tasks.
- Create data flow diagrams, state transition diagrams.
- Estimate system timing.
- Always start writing comments first then start implementing your code.
  - You can follow test driven development approach.
- Make use of TODO:, NOTE:, FIXME: like keywords in your comments.

Best Practice #9

Avoid using dynamic memory allocation

Dynamic memory allocation

- Problems with dynamic memory allocation:
  - Frequent calls to malloc() and free() can cause memory fragmentation.
  - Most implementations of functions malloc() and free() are non-deterministic and can consume huge amount of time to process.
  - Most dynamic memory allocation APIs are not optimized for embedded systems and have huge overhead.
  - Memory leaks can occur if software is not implemented properly.
- NOTE: ESP's Arduino framework is an abstraction over ESP-IDF APIs.
  - ESP-IDF APIs use FreeRTOS internally and makes heavy use of dynamic memory allocation.
- If dynamic memory allocation cannot be avoided try to reduce its usage or use a tracing tool to watch the heap consumption.

Best Practice #10

Implement logs to ease debugging.

Logging & Debugging

- Advantages:
  - Helps you avoid guess work and verify the functionality.
  - Helps you to troubleshoot the system.
- Implement at least UART based logging.
  - ESP32 uses the UART0 port for programming and logging.
  - ESP-IDF has decent logging APIs already implemented.
- You can consider implementing Network based logging to debug the software remotely.
- If the system has a File system make use of file based logs to troubleshoot / confirm the functionality.
- Use JTAG for debugging if there is a provision.
- If there is a JTAG debug port available, you can collect RTOS traces and view the behavior of the system graphically.
  - Segger System View
  - Percepio Tracelyzer

Best Practice #11

Make use of assert() statements to catch bugs in your code.

Assertions

- An assertion is a piece of code which will only get triggered when there is a bug in the program.
- The c header assert.h has the assert() macro defined.
  - ESP-IDF has its own version of assert() defined known as ESP_ERR_CHECK().
    - Will hang up and send a log message on the console UART port.
  - FreeRTOS has its own version of assert() defined known as configASSERT().
    - Will stop the scheduler and hang up at the assert statement.
- You can make use of assert statements:
  - To check pre & post conditions of a function. (Ex: passing NULL pointers).
  - When a call to malloc() fails. (Not always).
  - Whenever you decide that a boolean condition can never occur in your system logically.
- Using this approach will help you catch bugs at a very early stage of your project.

Best Practice #12

Make use of the watchdog timer.

Watchdog Timer

- WDT restarts the MCU if not cleared periodically.
- Helps when the code got hang up may be due to:
  - A hardfault occurred.
  - Wrong logic being executed.
  - Wrong code execution due to field related issues.
  - Interrupt enabled but ISR not implemented.
- Latest MCU cores have windowed WDTs.
- TIP: (If required) Reset your system periodically at some point of time to avoid a possible error condition.

Best Practice #13

Understand communication protocols well before you use them.

Communication protocols

- Make sure you read well about the hardware. (In case of UART, SPI, I2C).
  - Do not use Arduino drivers blindly.
  - Read the external peripheral datasheet.
- Understand what kind of architecture is supported by the protocol.
  - Client / Server
  - Master / Slave
    - Multiple masters allowed?
- Does the protocol guarantee packet delivery?
- Does it support security?
  - Start thinking about security from day 1.
  - Decide what level of security is enough for your system.
- Is the desired level of security implementable?
  - How much time does it take for encryption?
  - Do we need to have special cryptography hardware?
  - What about key generation & storage?

Best Practice #14

Develop a test harness for the software.

Test Harness

- Implement at least manual tests.
  - Tests should be written for every external peripheral.
  - Tests should be written for every actor in your DFD.
  - Make use of conditional compilation macros (#ifdef, #ifndef …) to avoid test's contribution towards memory consumption.
- Implementing automated unit tests is desired.
  - Unity + CMock is a good choice for unit testing.
  - Ceedling helps automate some of the unit testing process.
  - If you follow TDD, the test harness itself will act as a documentation for the software.
- Having a test harness will help you debug problems quickly without the use of a debugger.

Best Practice #15

Use a software version control system

Version control system

- Benefits are same as for any other software project.
- GIT is a good option.
- Host your repositories on Github or Atlassian or any other equivalent cloud platform.
  - Free private repositories.
  - Accessible worldwide.
  - Provides access control and branch level permission control.
  - Provides Issue tracker, wiki, release management
- Use an issue tracker to track features and bugs.

Further Reading

- Unity Test framework.
- Ceedling.
- ESP32 secure boot.
- Hard realtime embedded systems.
- Test-Driven Development for Embedded C – By James W. Grenning
- Mastering the FreeRTOS Real Time Kernel – By Richard Barry.
- The Art of Designing Embedded Systems – By Jack Ganssle.
- Implementing state machines on embedded systems.
- RTOS Tracing.