Notes for TI

May 12, 2022

Contents

1 RTOS

Real **T**ime **O**perating **S**ystem

Key features:

- **Minimal** interrupt and thread switching **latency**.
- **Deterministic**
- **Scalability**
- **Structured Software** (don't know what this one means :() website says it's easy to add additional components into application
- **Offload development** (takes the load off of developer by doing scheduling, power mangement, memory management, exception handling, interrupt table management)

When is an RTOS needed?

RTOS is used in the context of embedded devices. It is used when "more" complicated functionality is needed, more than just a few simple actions and when scalability is needed. **Embedded device/system** is a small computer that forms parts of a larger system. For example, a computer inside of a microwave to do automatic turning off, a computer inside a washing machine, air conditioner, engine systems inside automatic transmission vehicles, TV, phone, GPS tracker, robotics system (Mars rover), Alexa, etc. On the other end of RTOS is **Bare-Metal** applications.

Thus like any CPU, RTOS has:

- Scheduler

- Communication mechanism (semaphore, message queues, queues)
- Critical region mechanism (mutexes, gates, locks)
- Clocks, timers
- Power management
- Memory management (heaps, fixed or variable sized)
- Peripheral drivers (UART, SPI, I2C)
- Protocol stacks (WiFi, etc)
- File system
- Device management (exception handling, booting, etc)

From the components we can see that an RTOS might be required in Alexa as it needs to connect to WiFi, store commands, understand commands, connect to other devices get data from them using UART etc. But might not be required for AC since it just needs to sense the temperature and turn on or off accordingly, maybe have a timer at max.

2 UART

Universal **A**synchronous **R**eceiver/**T**ransmitter

UART does the job of receiving and transmitting data from CPU to peripheral. It supports the CPU by buffering, by converting serial data coming from peripherals to parallel for CPU use and converting parallel data from CPU to serial for transmission to peripherals. It also stores error information along with data in buffer. The serial-to-parallel and parallel-to-serial communication is what makes it **asynchronous**. It typically functions in a microcomputer system as a serial input/output interface.

Side question what are peripherals? Mouse, keyboard, monitor, GPS, IMU (used in robotics, tells speed, rotational speed and heading direction. Short for Inertial Measurement Unit), seismo sensor, anything that sense and transmits data.

How this is done? Very complicated hardware design. Not for a CS person.

3 SPI

Serial **P**eripheral **I**nterface

Analog devices have digital interface between microcontroller and peripheral, this is where the SPI is needed. The byte of data sent can be sent over 8 parallel lines, or one after the other in a single line. SPI defines the communication protocol for the latter. Communication requires common understanding of what high and low voltages are that represent 0s and 1s. Also require common timing between a master and a slave device; often serial clock from master is used.

SPI has two control lines: slave select and serial clock. SPI bus can control multiple slaves, but there can only be one master. Each slave has a slave select for independent control. Data can be transferred be full duplex.

Key hardware lines:

- SS: slave select
- SCLK: serial clock. syncs data transfer between master and slave.
- MOSI: master in, slave out. sends data from master to slave.

- MISO: the opposite. Can be shared between all slave devices.

Data might be read during the rising or falling edge of S-Clock. Clock polarity is also important. These allow for many modes of SPI.

4 I2C

Inter **I**ntegrated **C**ircuit (I^2C)

Created by Philips Semiconductor in 1982.

Requires only two lines for communication, it has a serial data line and a serial clock line. Connect to many devices with only two lines. Simple and economical. Several speed modes. 100 kbps, 400 kbps, 1 Mbps, 3.4 Mbps, 5 Mbps (5 Mbps mode omits some features and is write-only).

Key hardware lines:

- SCL - serial clock
- SDA - serial data line
- SDA is bi-directional communication, half duplex. Sending configuration data or conversion data etc.
- Unique address allows for multiple controllers and multiple targets on the I2C bus.

Hardware implementation is difficult to understand or explain. Here

5 ARM

Acorn **R**ISC **M**achine

Reduced **I**nstruction **S**et **C**omputing Protoype chip made in 1985. Low power chip, for 1W, but actuall worked at 100mW. Apple Mac with M1 has ARM. By 1986 Apple began using ARM for its R&D.

Advanced **RI**SC **M**achines (the new ARM) spun off from it in 1991 from investment from Apple and VLSI. 6 billion chips per quarter in 2021. Found in smartphones, tablets, laptops, servers, everywhere.

ARMs ISA (Instruction Set Architecture) is now advanced and mature, includes cryptography, MTE (Memory Tagging Extension). Now there are v9 architectures. Called Cortex, A710, X2, Neoverse N2, etc. Scalable Vector Extension 2 (SVE2) at the heart of world's fastest supercomputers, better ML, etc. Arm Confidential Computer Architecture (CCA) (Realms) run mini hypervisor in chip so banking software can't get access to hardware, cloud computing, isolation, etc.

Cortex A processors (found in smartphones): 32 bit is dead, 64 bit is going to come. 4 types of processor. Cortex M (microcontrollers) Cortex R (for real time applications) Neoverse (for servers)

ARM has a simplified, logical and efficient instruction set. Apart from that, it doesn't seem fruitful to look at the exact instruction set right now. Here.

6 Compiler tool chain

Here's what I got on stackoverflow. In order to compile the source code to run on a target machine (not your machine), you need to compile it with a specific compiler, linker, debugger, etc. This is calle cross-compilation and the tools used is called the toolchain. Say for example you have a toolchain for ARM.

7 Makefile

Here. Purpose: make the build process easier by writing the commands to build specific objects in a project as rules.

In simple english, makefile may have rules like

- For target *object.o* run the following command, make sure its dependencies exist.
- For target of type **.exe* run the following command to build them.

It only updates those targets that are needed, not a fresh build every time, this is decided based on dependencies. It can do much more, clean out the installation, delete some things, etc. It's a full language on its own. Makefile can be seen as a script read by the `make` or `cmake` or `gmake` or other make like utilities.