>>> From Operating Systems to No-OS
>>> A journey from computers to micro-controllers

Name: João Pedro Dias
jpmdias@fe.up.pt
Operating Systems
Date: May 10, 2021

U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

>>> Operating Systems

"A modern computer consists of one or more processors, some main memory, disks, printers, a keyboard, a mouse, a display, network interfaces, and various other input/output devices. All in all, a *complex* system. For this reason, computers are equipped with a layer of software called the operating system, whose job is to provide user programs with a better, simpler, cleaner, model of the computer and to handle managing all the resources."

Modern Operating Systems, Andrew S. Tanenbaum

>>> Full-fledged OS features

* Process management
* Interrupts
* Memory management
* File system
* Device drivers
* Networking
* Security
* I/O
* …

Do all the computing-enabled objects require an OS?

E.g.: mobile phones, routers, printers, robots, aircraft, coffee machines, light-bulbs...

>>> Not all, but some (most?).

(a) Nokia 1100 (S30+ software)

(b) Samsung Galaxy A21s (Android 10+)

>>> Embedded Development

"Embedded software is computer software, written to control machines or devices that are not typically thought of as computers, commonly known as embedded systems. It is typically *specialized for the particular hardware that it runs on and has time and memory constraints.*"

From Wikipedia

>>> Router

Figure: Linksys WRT54GS. Broadcom BCM4712 @ 200MHz, 1 core, 8MB Flash, 32MB RAM.

Does it run Linux?

Yes! Several versions: stock (device-specific OS), OpenWRT, DD-WRT,…

In the WRT54GS (and other routers) the whole operating system is written in the flash memory (8MB), which is only updated if a new version of the firmware is available (updating the whole "disk"). There's, typically, a small user-writable part where the configurations are saved.

>>> Routers, in general

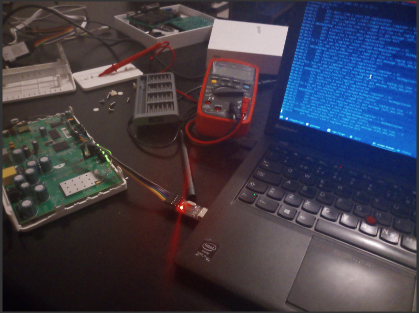


Figure: Getting a shell in a D-Link router.

* Mainly accessed and managed by their Web GUIs;
* Typically, a shell is available using its serial port (not always easy to access);
    * Typically it is limited to a few utilities (e.g. BusyBox)
* There are only the services required for its proper functioning, e.g., networking services.

If we can build really *tiny* Linux images,...
why not use Linux for everything?

* Size/space
* Real-time requirements
* CPU usage

Linux is a general-purpose OS (GPOS), and its use in embedded system is highly motivated by the broad device support, file-systems, network connectivity, and UI support; thus making it easy to configure and deploy at a low-cost (when compared to a hardware-tailored solution).

>>> Linux limitations

* OS size
    * Even a stripped-down kernel has a footprint of a few megabytes.
    * In 2004 Eric Beiderman managed to get a kernel booting with 2.5MB of RAM (https://groups.google.com/g/linux.kernel/c/1oj-12EdzH4)
* Requires significant CPU resources
    * > 200MIPS, 32-bit processor, 4MB ROM, 16MB of RAM just to be able to boot (which may take several seconds)
* No *hard* real-time scheduler
    * Although Linux has several scheduling options, including a real-time scheduler, it is a *soft* real-time --- it is real-time most of the time, but sometimes not. Typical latencies in real-time Linux will be on the order of tens or hundreds of microseconds.

>>> The RTOS alternative

"A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time applications that process data as it comes in, typically without buffer delays.

Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter. A real-time system is a time-bound system which has *well-defined, fixed time constraints*. Processing must be done within the defined constraints or the system will fail.

They either are *event-driven* or *time-sharing*. Event-driven systems switch between tasks based on their priorities, while time-sharing systems switch the task based on clock interrupts. Most RTOSs use a *pre-emptive scheduling* algorithm."

From Wikipedia

>>> RTOS in the wild

* Most programmable logic controllers (PLC), common in manufacturing machines, use some kind of RTOS;
* Air traffic control systems;
* Defense application systems like RADAR;
* Aerospace systems (e.g. Mars 2020 rover uses VxWorks);
* Networking and communication infrastructure;
* Medical systems;
* …

OS alternatives

FreeRTOS, Zephyr, VxWorks, OS-9, LynxOS, OSE, QNX, RIOT, …

>>> RTOS Goals

* Small latency: *real-time*;
* Determinism: deadlines must be met at all times;
* Structured software: easy to add additional components into the applications;
* Scalability: Must be able to run simple to complex applications (stacks, drivers, file systems, …)
* Offload development: Allows developers to focus on the applications. As an example, an RTOS, along with scheduling, generally handles power management, interrupt table management, memory management, exception handling, …

From TI Dev Docs,
https://dev.ti.com/tirex/explore/node?node=ADznPZ0t1iqMbLyS3k3ZFA__fc2e6sr__LATEST

>>> RTOS Components

* Scheduler: Preemptive scheduler that guarantees the highest priority thread is running.
* Communication Mechanism: Semaphores, Message Queues, Queues, etc.
* Critical Region Mechanisms: Mutexes, Gates, Locks, etc.
* Timing Services: Clocks, Timers, etc.
* Power Management: For low power devices, power management is generally part of the RTOS since it knows the state of the device.
* Memory Management: Variable-size heaps, fixed-size heaps, etc.
* Peripheral Drivers: UART (Universal asynchronous receiver-transmitter), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), etc.
* Protocol stacks: BLE, WiFi, etc.
* File System: FatFs, etc.
* Device Management: Exception Handling, Boot, etc.

From TI Dev Docs,
https://dev.ti.com/tirex/explore/node?node=ADznPZ0t1iqMbLyS3k3ZFA__fc2e6sr__LATEST

>>> Types of *execution blocks*

* Interrupt Service Routine (ISR): Thread initiated by a hardware interrupt. An ISR runs to completion. ISRs all share the same stack.
* Tasks: Thread that can block while waiting for an event to occur. Tasks are traditionally long-living threads (as opposed to ISRs which run to completion). Each task has it's own stack which allows it to be long-living.
* Idle: Lowest priority thread that only runs when no other thread is ready to execute. Generally Idle is just a special task with the lowest possible priority.

From TI Dev Docs,
https://dev.ti.com/tirex/explore/node?node=ADznPZ0t1iqMbLyS3k3ZFA__fc2e6sr__LATEST

Do all the computing-enabled objects require an OS?

E.g.: ~~mobile phones, routers, printers, robots, aircraft~~ coffee machines, light-bulbs...

>>> Bare-metal

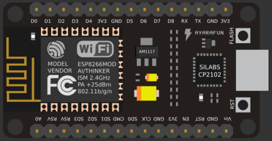
Toggling a LED and other small computational tasks do not require an OS at all. In these cases the hardware can be directly programmed to accomplish the required tasks.

Let's take as an example a simple smart lightbulb, that we can:

* Turn ON/OFF remotely by some protocol (BLE, Zigbee, …);
* Adjust brightness;
* Adjust color.

>>> BrilliantSmart Bulb

(a) BrilliantSmart 4.5W 400lm CCT Bulb


D0 D1 D2 D3 D4 3V3 GND D5 D6 D7 D8 RX TX GND 3V3
MODEL VENDOR
ESP8266MOD AI/THINKER ISM 2.4GHz PA +25dBm 802.11b/g/n
AM1117
SILABS CP2102
FLASH
RST
A0 RSV RSV SD3 SD2 SD1 CMD SD0 CLK GND 3V3 EN RST GND Vin
(b) ESP8266 Microcontroller

(c) Arduino (C)

For scenarios like this one, bare-metal solutions suffice. They are typically small, fast, and relatively easy to understand and build.

>>> Key Components of Bare-Metal

* Initialization: Initializing the hardware and software components in main().
* Super-loop state machine: Code to manage the application. The actions are based on the results of interrupts (e.g. a SPI packet was received or a timer expired) or polling.
* ISRs: Code executed by interrupts for peripherals (e.g. UART), timers or other device-specific items (e.g. exceptions).

From TI Dev Docs,
https://dev.ti.com/tirex/explore/node?node=ADznPZOt1iqMbLyS3k3ZFA__fc2e6sr__LATEST

>>> How to program bare-metal applications? Arduino example

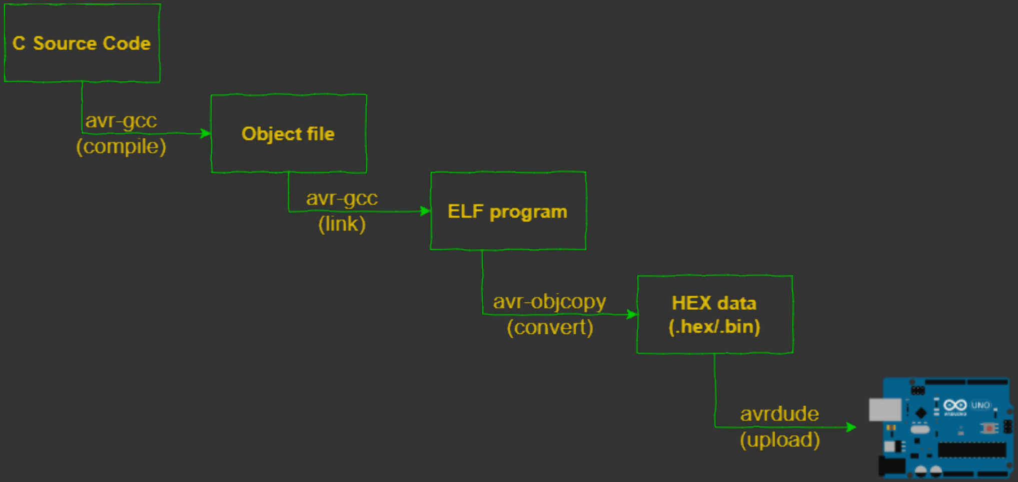
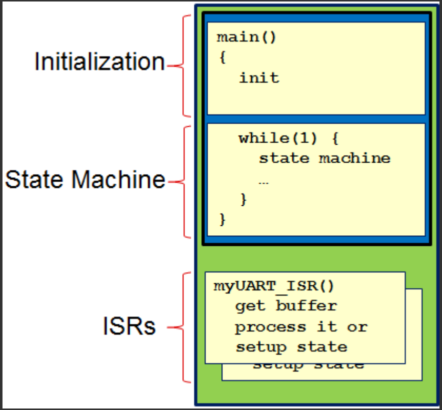


Figure: From C to blink.

>>> Analysis of a bare-metal application



```
main()
{
  init

  while(1) {
    state machine
    ...
  }
}
```

```
myUART_ISR()
  get buffer
  process it or
  setup state
```

Figure: Bare-metal code structure. From TI Docs.

>>> Arduino Blink

```
// Pin 13 has an LED connected on most Arduino boards.
int led = 13;
// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
  // initialize serial output at 9600 baud speed
  Serial.begin(9600);
}
// the loop routine runs over and over again forever
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  Serial.println("LED ON");  // writes LED ON to the serial
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  Serial.println("LED OFF"); // writes LED ON to the serial
  delay(1000);               // wait for a second
}
```

>>> Arduino Blink without Arduino Libraries

```
void setup() {
    // initialize digital pin 8 as an output.
    DDRB = DDRB | B00000001; // The Port B Data Direction Register (R/W)
    /* this is safer as it sets pin 8(PB0) as output
       without changing the value of other */
}

// the loop routine runs over and over again forever
void loop() {
    PORTB = 0B00000001;    // sets digital pin 8(PB0) HIGH
    delay(1000);           // wait for a second
    PORTB = 0B00000000;    // sets digital pin 8(PB0) LOW
    delay(1000);           // wait for a second
}
```

>>> From bare-metal to RTOS

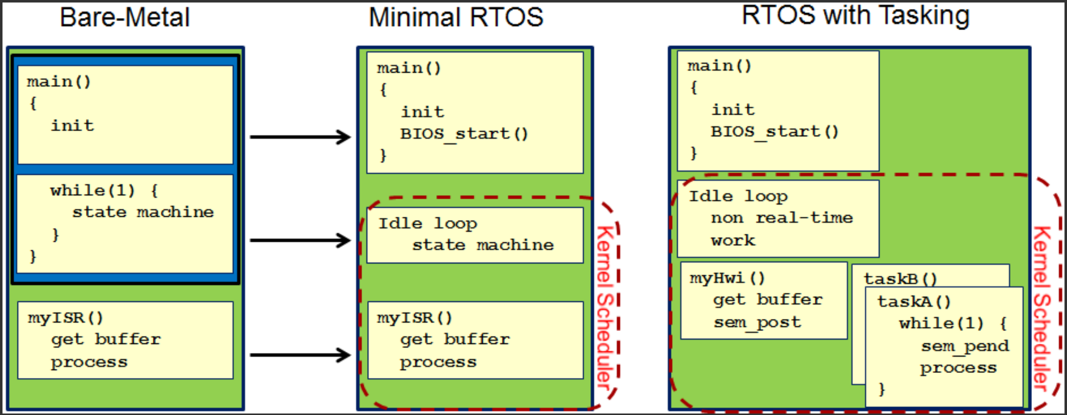


Figure: Bare-metal *vs* RTOS. From TI Docs.

>>> RTOS Blink (1/2)

```c
/* Example for FreeRTOS running on a ESP32 board */

#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "driver/gpio.h"

#define BLINK_GPIO 13

void app_main() {
    nvs_flash_init();
    xTaskCreate(&hello_task, "hello_task", 2048, NULL, 5, NULL);
    xTaskCreate(&blinky, "blinky_task", 512,NULL,5,NULL );
}
```

>>> RTOS Blink (2/2)

```c
void hello_task(void *pvParameter) {
    while(1) {
        printf("Hello world!\n");
        vTaskDelay(100 / portTICK_RATE_MS);
    }
}

void blinky_task(void *pvParameter) {
    gpio_pad_select_gpio(BLINK_GPIO);
    // Set the GPIO as a push/pull output
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
    while(1) {
        gpio_set_level(BLINK_GPIO, 0); // Blink off (output low)
        vTaskDelay(1000 / portTICK_RATE_MS);
        gpio_set_level(BLINK_GPIO, 1); // Blink on (output high)
        vTaskDelay(1000 / portTICK_RATE_MS);
    }
}
```

>>> Some (extra) Alternatives

* MicroPython: a full Python compiler and runtime that runs on the bare-metal. It has a REPL to execute commands immediately, along with the ability to run and import scripts from the built-in filesystem. CircuitPython is an alternative implementation more simple to use.
    * https://micropython.org/
* Forth: An imperative stack-based computer programming language that directly connects the user and hardware. It leverages the use of Stacks and Reverse Polish Notation, and allows direct inspection of the assembly code generated.
    * http://mecrisp.sourceforge.net/
* Espruino: JavaScript interpreter for microcontrollers. It is designed for devices with as little as 128kB Flash and 8kB RAM.
    * https://github.com/espruino/Espruino

>>> Do we really need code at all? Hardware-only Blink

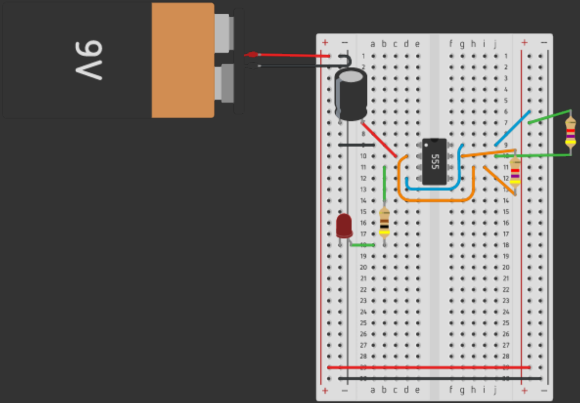
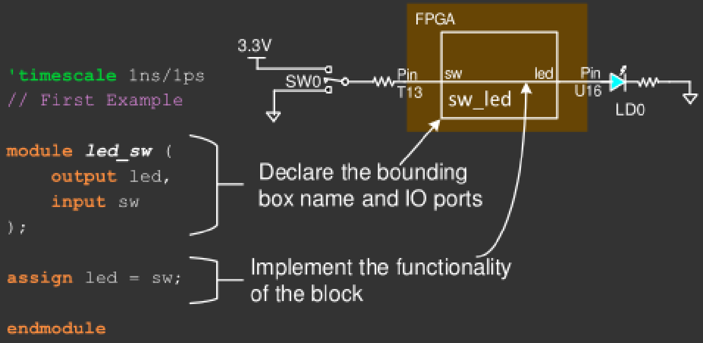


Figure: Driving a LED with a 555 Timer[1].

[1]Read more: https://www.jameco.com/Jameco/workshop/TechTip/555-timer-tutorial.html

>>> Or doing hardware with code? FPGAs



```
'timescale 1ns/1ps
// First Example

module led_sw (
      output led,
      input sw
);

assign led = sw;

endmodule
```

Declare the bounding box name and IO ports
Implement the functionality of the block
FPGA
3.3V
SW0
Pin T13
sw
led
sw_led
Pin U16
LD0

Figure: Switching a LED in Verilog HDL[2].

EEVblog #496 - What Is An FPGA? https://www.youtube.com/watch?v=gUsHwi4M4xE
github/fpga-8bit-console https://github.com/hugoferreira/fpga-8bit-console

[2]From: https://reference.digilentinc.com/learn/

>>> Read (Watch) More

* General sources of knowledge:
    * https://learn.adafruit.com
    * https://www.arduino.cc/en/Tutorial/HomePage
    * Physical Computing https://makeabilitylab.github.io/physcomp/
    * Design and simulate circuits https://www.tinkercad.com
* Finding *cool* projects and ideas:
    * https://hackaday.com/ and https://hackaday.io/
    * https://www.instructables.com/circuits/
    * https://www.hackster.io/
* Computer (and OS) from scratch:
    * "8-bit CPU from scratch" / "Build a 6502 computer" https://eater.net/
    * NandGame https://nandgame.com/
* Youtube:
    * LiveOverflow https://www.youtube.com/channel/UClcE-kVhqyiHCcjYwcpfj9w
    * stacksmashing https://www.youtube.com/channel/UC3S8vxwRfqLBdIhgRlDRVzw
    * EEVblog https://www.youtube.com/channel/UC2DjFE7Xf11URZqWBigcVOQ
    * GreatScott! https://www.youtube.com/c/greatscottlab

>>> The End

Beware of the *magic smoke* when messing around with hardware.

magic smoke: n.

A substance trapped inside Integrated Circuits (IC) that enables them to function. Its existence is demonstrated by what happens when a chip burns up —the magic smoke gets let out, so it doesn't work any more.

magic smoke, http://www.catb.org/jargon/html/M/magic-smoke.html

Some of my magic smoke adventures can be found @ https://jpdias.me/notes/