4. Embedded System Software

Introduction

- Embedded system = Hardware + Software

- *"The hardware is the blank canvas and the software is the paint that we add in order to make the picture come to life".*

Source: "Programming Embedded Systems", Anthony Massa and Michael Barr

4.1. Embedded Software Features

Embedded Software Developer

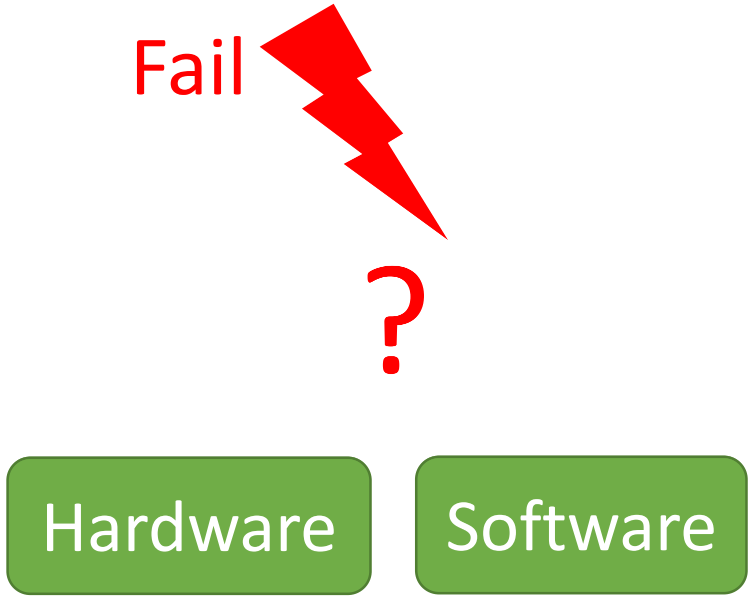
Differs from other types:

- Hardware knowledge
- Efficient code
- Peripherals interfaces: communicate with devices to control or reacting
- Robust code
- Minimal resources: run out the memory if you do not plan ahead
- Reusable software: code portability
- Development tools

Embedded Software Challenges

An embedded system performs a specific task, cutting out the **resources** it doesn't need:

- Memory (RAM)
- Code space (ROM)
- Processor cycles or speed
- Battery life (or power savings)
- Processor peripherals



You have to know about what the hardware is capable of !!!

Embedded Software Challenges

- Customers expect "perfect" embedded SW
  - Bugs can lead to lawsuits
  - Upgrades can be painful to deploy (Flexible → Modularity)
  - Limited hardware resources

- Real-time operation
  - Interaction with system-specific sensors and actuators

- Most embedded software is Mission Critical
  - Safety: someone gets killed or injured
  - Mission Critical: failure results in loss (money, business,...)

Source: 2020 Philip Koopman

Principles of High Quality Embedded Software

- Maintainable
- Testable
- Portable
- Robust
- Efficient
- Consistent

Embedded Systems decisions

- The choice of microcontroller. √

  Software requirements → choosing tecnology

- The choice of programming language.
- The choice of operating system.

4.2. Embedded Software Languages

Embedded Software Languages

- **Assembly**: In the early days exclusively in the assembly language

  Disadvantages:

  - Higher development cost
  - Lacks of code portability.
  - Now used as adjunct to the high-level language

- **C language**:
  - Small and simple to learn
  - Compilers available for almost every processor
  - Large experience of programmers.
  - Processor-independence
  - Gives direct hardware control and have benefits of high level

Embedded Software Languages

- **Embedded C** differs from C in:
  - Efficient Memory Management
  - Direction HW/IO control
  - Code size constraints
  - Optimized execution
- C++:
  - Functionality for better data abastraction
  - More object-oriented style of programming.
  - Reduces the efficiency of the executable program.
- Java:
  - Not as widely **used** as C or C++ owing to its relatively large memory requirements.
  - Non hard real time tasks.
- Rust

4.3. Bare metal versus Operating System

Bare Metal versus S.O.

Two differences between the embedded systems and desktop computer systems:

- Embedded systems are required to run only **one program**.
- Facilities provided by the S.O. are of little value in embedded systems.

Bare-Metal (Super-loop)

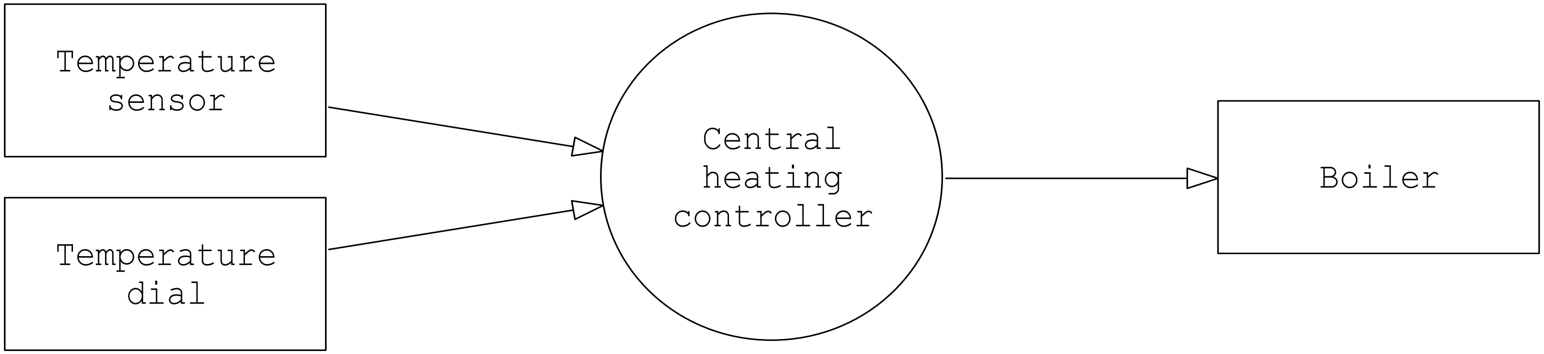
The smallest embedded C program:

```
void main(void)
   {
   /* Prepare for task X */
   X_Init();

   while(1) /* 'for ever' (Super Loop) */
      {
      X(); /* Perform the task */
      }
   }
```

Source: "Embedded C", Michael J. Pont. Addison-Wesley

Example: Central-heating controller




```
void main(void)
   {
   /* Init the system */
   C_HEAT_Init();

   while(1) /* 'for ever' (Super Loop) */
      {
      /* Find out what temperature the user requires
         (via the user interface) */
      C_HEAT_Get_Required_Temperature();

      /* Find out what the current room temperature is
         (via temperature sensor) */
      C_HEAT_Get_Actual_Temperature();

      /* Adjust the gas burner, as required */
      C_HEAT_Control_Boiler();
      }
   }
```

Source: "Embedded C", Michael J. Pont. Addison-Wesley

Bare-Metal (Super-loop)

- To perform all the tasks in a reasonable amount of time, but also in a good order

Setup

Loop

```
Function Main_Function()
{
  Initialization();
  Do_Forever
  {
    Check_Status();
    Do_Calculations();
    Output_Response();
  }
}
```

Bare-Metal (Power-save Super-loop)

- Example: An embedded system which has an average loop time of 1ms, and needs only to check a certain input <u>once per second</u>.

```
Function Main_Function()
{
  Initialization();
  Do_Forever
  {
    Check_Status();
    Do_Calculations();
    Output_Response();
   Delay_For_Next_Loop();
  }
}
```

Bare-Metal (Power-save Super-loop)

- Microcontrollers have **power-save modes**

A microcontroller uses 20mA of current in "normal mode", but only needs 5mA of power in "Low-Power Mode". Let's say that we are using the previous example super-loop, which is in "Low-Power Mode" 99.9% of the time (1ms of calculations every second), and is only in normal mode 0.1% of the time:

$$\text{Power} = \frac{(99.9\% \times 5\,\text{mA}) + (0.1\% \times 20\,\text{mA})}{100\%} = 5.015\,\text{mA} \quad \text{Average}$$

Bare-Metal (Super-loop)

Strengths and weaknesseses of “super loops”

- ☺ The main strength of Super Loop systems is their simplicity. This makes them (comparatively) easy to build, debug, test and maintain.
- ☺ Super Loops are highly efficient: they have minimal hardware resource implications.
- ☺ Super Loops are highly portable.

BUT:

- ☹ If your application requires accurate timing (for example, you need to acquire data precisely every 2 ms), then this framework will not provide the accuracy or flexibility you require.
- ☹ The basic Super Loop operates at ‘full power’ (normal operating mode) at all times. This may not be necessary in all applications, and can have a dramatic impact on system power consumption.

Source: “Embedded C”, Michael J. Pont. Addison-Wesley

RTOS vs Bare-Metal : Early decision

- Bare-metal →
  - No scheduler. (Only one task)
  - All activity is either polled or interrupt-driven.
  - More deterministic, easier to test and debug
- RTOS →
  - Includes a scheduler
  - Harder to debug and trace, but easier to manage the tasks, high-priority or not
  - TCP/IP stacks, USB, HD video, and other subsystems → Not 'bare-metal'

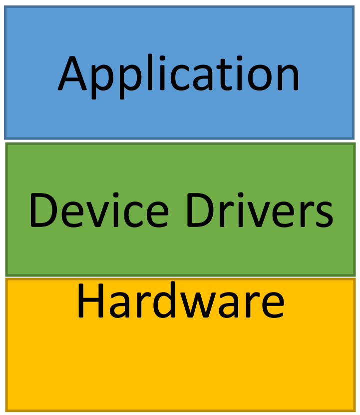
Bare-metal → project is simple or penalty for failure is high.

RTOS → lots of tasks, lots of desktop-style I/O, or a sophisticated user interface.

4.4. Software organization

Embedded software organization

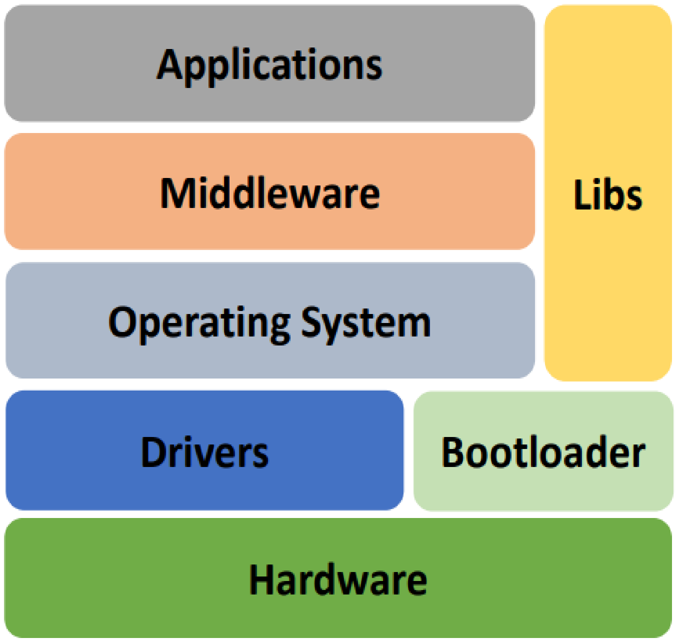
Basic diagram (Bare metal)



- Hardware
- The Device Drivers: functionality to operate with the hardware avoiding application to know the hardware.
- The application layer processes the different inputs and controls the outputs based on what the user commands it to do.

Embedded software organization

More complex diagram (RTOS)



Source: University of Colorado

- Hardware
- The Device Drivers
- Code Booting: to start the system
- Operation System (OS)
  - Abstracts High from Low levels
  - Scheduling process
  - Resource management

Real-time operating system (RTOS)

- Code Libraries for shared code
- Application