A low-cost temperature control system for learning real-time programming

Sounkalo Dembele, Patricia Hirtz, Tristan Muller

A low-cost temperature control system for learning real-time programming

Sounkalo Dembélé*, Patricia Hirtz, Tristan Muller
FEMTO-ST Institute, AS2M Department, Univ. Bourgogne Franche-Comté,
Univ. de Franche- Comté/CNRS/ENSMM, 24 rue Savary, 25000 Besançon, France
* Corresponding author: sounkalo.dembele@univ-fcomte.fr

Abstract

Real-time multitasking, by opposition to shared-time multitasking, is required for applications where the response-time to events are important, even more critical like in control and automation or drivers programming. Find a low-cost model to learn real-time systems, that is accessible to a high number of people is an issue. The paper deals with this issue by presenting a temperature control system that consists of salvage components, and open-source hardware and software. It fits the circular and star-up industries philosophy: it is cheap and can be manufacture on demand.

Keywords: Temperature control, Educational model, Real-time, Linux-RT, Raspberry Pi, Arduino, OpenCV, C/C++, Circular industry, Start-up industry.

1 Introduction

Educational systems are known to be very expensive because of their requirement of robustness during labs, but more particularly because of the small number of products to manufacture. Indeed, the higher the number to produce, the lower the production cost and then the price. In turn, high price makes the acquisition of systems very hard, particularly for African universities.

Let us focus on a system allowing to learn real-time application design and development through temperature control and monitoring. A quick search on Internet gives few conclusive results: "real-time" leads to global information about real-time, "temperature control model" gives only general results on temperature control, "temperature control lab" gives a system based on Arduino with Python or Matlab codings (Oliveira and Hedengren (2019)), the educational system vendors Didalab proposes a vehicle to control and monitor with an adhoc real-time kernel (Didalab (2022)).

Finally, there is a requirement of a low-cost system, for learning real-time systems and applications, that is accessible to many people, more particularly African universities and students. Since the number of systems to manufacture will be very few, the only solution remains, a start-up industry solution: a solution based on the use of open-source hardware and software, and can be manufactured on the demand. The paper presents this kind of solution. It extends the system described in (Grolleau et al. (2018)): a temperature control and monitoring system that uses Raspberry Pi running Linux-RT (i.e. Linux with the patch PREEMPT_RT), Arduino, camera for image acquisition with OpenCV and C/C++, low-cost temperature sensor, fan taken from a old computer.

2 Real-time basics

A real-time application (RTA) is a time-critical application. Its implementation requires a real-time system, i.e. a computer system where the response-time to an event (timer, interrupt, value of an input, etc.) or **latency** is predictable. Typically, latency is in the order of milliseconds or nanoseconds. When it is strictly defined (i.e. constant) whatever computational load, the system is said hard real-time, otherwise it is said soft real-time.

RTAs are found in control and automation (of airplane flight, car drive, etc.), video and audio plays, electronics and embedded electronics (programming of drivers), supervisory control and data acquisition (SCADA), computer vision, or robotics, where sampling time and response-time accuracy are fundamental. The other important feature of RTAs is the coexistence of multiple tasks with different dynamics: from low frequency tasks (hundreds milliseconds periods) like displaying data, to high frequency tasks (few milliseconds periods) like signal acquisition, as a consequence the best solution for RTAs is multitasking.

RTAs are implemented in computer systems, that are of two types depending on whether they run or not an OS (Operating System):

- symmetrical multiprocessor (SMP) systems that may be single or multiple board computers (multiprocessor computers, multicore computers, hyperthreading computers), boot into a firmware that loads a multiple purpose application, the

OS,

- bare-metal systems, that include PLCs (programmable logic computers), MCU (MicroController Unit)-based boards and DSP (Digital Signal Processor)-based boards, boot into a firmware that initializes hardware and loads nothing or a monolithic and single purpose application, user application.

Firmware can contain a boot loader or a code that loads the boot loader.

OS performs two *a priori* unrelated tasks. It provides programmers and application software with a clear set of usable resources, e.g. processors, memories, timers, disks, mouse, screens, network interfaces, etc. It manages theses resources by allocating them to the various competing programs that request them. The corresponding application that is the core of OS is the **scheduler**. When the latter is capable of high performance latency management, it is said real-time scheduler and the corresponding OS, a RTOS (Real-time OS). Only RTAs require high-performance scheduling.

FPGA, DSP, SMP running Xenomai, RTAI, VxWorks, NI Linux Real-time or TwinCAT in association with Windows are hard real-time systems whereas SMP running Linux with a real-time scheduler, Linux-RT, i.e. Linux with the patch PREEMPT_RT are a soft real-time systems. However, bare-metal systems are not flexible enough for applications development and do not fit high level communications (USB, TCP, etc.), whereas RTOS running systems do. This limitation is such that it may be useful to install lightweight RTOS on bare-metal, i.e. `freeRTOS` on Arduino.

3 Real-time with Linux

The paper focuses on the development of RTAs using Linux and C programming (Blaess (2019), Blaess (2005)).

Because of multiple processor, SMP computers are capable of running genuinely multiple task in parallel. A task may be implemented as a **process** or a **thread**.

A process is a stand-alone program: it has everything private, particularly physical memory, to run autonomously. In a multiple process application, processes run independently by, eventually, exchanging messages through high level communication protocols, e.g. inter-process communication (IPC), channels (signals, sockets, files, etc.). That communication is slow and requires a large amount of internal resources.

A thread is a piece of code, i.e. typically it is a function. A process may consist of a single thread or a multiple thread. In the latter case each thread runs independently of the others, but all threads share the same address space and most of the same data. These sharing and lack of data protection make it easier to implement threads than processes.

Regarding C language, every program has at least one thread, which is started by the C runtime: the thread running `main()` function or main thread. This thread can then launch additional threads.

Linux deals with threads through the POSIX (Portable Operating System Interface in uniX) standard: it is said Posix threads or simply `Pthreads`. It has functions for creating and managing threads:
`pthread_create()`, `pthread_exit()`,
`pthread_join()`,
`pthread_mutex_lock()`, etc.

All Linux applications, except scheduler, run in user mode/space (or restricted mode/space), i.e. they can run only a part of computer instructions set, can achieve only some input/output operations towards external peripherals and can access only to some range of virtual memory. Corresponding processors are said to be in user/restricted mode/space. By contrast, the scheduler application can do everything: it is said to be in kernel mode/space (or privileged mode/space). Corresponding processors are said to be in kernel/privileged mode/space.

Usually, there is more tasks (processes and threads) than available resources (processors, console, memory, etc.). Then the scheduler, with it kernel mode, has to manage the affectation of tasks to resources: a task maybe running on a processor, runnable by waiting for a processor to be free, sleeping because of a system call or waiting an event, etc. Because of the ability to remove a processor from a task (running state) that switches into runnable state, and to elect another task, the scheduler is said pre-emptive. Transitions between the states of a task are due to the arrival of three types of event: hardware interrupt, system calls or exception (figure 1).

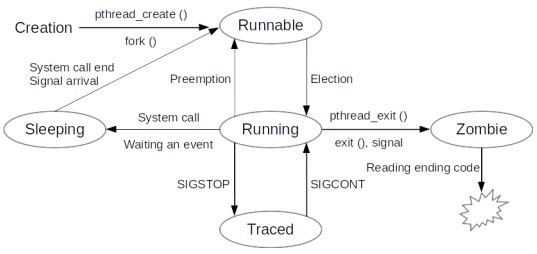

Figure 1: Main states, including Runnable state, and transitions between states, of a task, assuming Sleeping, Down and Idle are the same.

A hardware interrupt corresponds to a set of electrical signals from a peripheral to the scheduler to inform the latter of the occurring of an external event.

An initial signal is sent to system's interrupt controller that analyzes the change and in turn sends an electrical pulse to an IRQ (Interrupt ReQuest) pin on the processor. The latter interrupts the work in progress, stores in the stack the state of the registers and the address of the instruction being executed, consults the interrupt vector table which contains the address of the routine to be executed according to the IRQ number, then executes the code located at the address corresponding to that IRQ number. Once the interrupt processing is complete, the processor resumes the processing it was doing before. Interrupt-based apps are much better than interrupt-free (or polling) apps where a set of tasks is ran sequentially: response-time is predictable and energy consumption is reduced. Hardware interrupt is a key concept of real-time.

Usually, software interrupts are used to deal with system calls that are implemented not in the OS but in the C library (`GlibC` or `uClibC`). A software interrupt is an assembler instruction that the program normally executes from the user space. However, when the instruction is decoded by the processor, everything happens as if a hardware interrupt had occurred: saving the registers, switching to kernel mode, and branching to the address given the interrupt vector table. Exceptions are similar to hardware interrupts. Sometimes called as asynchronous interrupts, they arise when certain error conditions are encountered in the running code: they are handled by the processor without any external intervention.

Most systems under severe time constraints rely on periodic tasks, which must be performed with the best possible granularity and variability. To perform this type of task, POSIX timers have been developed as well as time measurement capabilities. A timer is programmed with a given initial delay and repeat period, when it is triggered, the kernel delivers a signal to the task and reset the timer. The expression timer interrupt is sometimes used because timer acts like hardware interrupt. Functions are available in the `time.h` header to deal with time: `timer_create()`, `imer_settime()`, `timer_delete()`, `clock_gettime()`, `clock_settime()`, etc. (Blaess (2019) Blaess (2005)).

Without any special extension Linux offers real-time processing possibilities with a real-time scheduler: real-time tasks (i.e. with real-time scheduling) have a specific priority scale, completely independent of standard tasks (i.e. with shared-time scheduling):

- the value 0 is for shared-time tasks (it is given by default, then shared-time scheduling is used),
- the values 1 to 99 are for real-time tasks (the higher the level the higher the priority of the task).

The scheduling and priority level are chosen task by task: either at the process or thread level. The general principle is that a task of a given priority can never be preempted or left in agreement (Runnable) while a task of lower priority has the processor. When several tasks have the same priority level, there are two solutions of real-time scheduling in the POSIX norm: FIFO (First In, First Out) in which the first waiting task is served, and RR (Round Robin) in which every task is served a quantum of time.

As a general rule, when designing a real-time system, one tries to assign only one task per priority level: the most urgent tasks and those tolerating the least time fluctuations are assigned high priority levels, while the others are assigned lower priority levels. If tasks have equivalent levels of criticality or urgency, they are given close priorities. And then FIFO is used in order to be sure that a high level task will be proceeded first.

Functions to deal with scheduling configuration for processes and threads are available in the `sched.h` header: `sched_setscheduler()`, `pthread_setschedparam()`, etc.

Linux with real-time scheduling works well for periodic tasks as well switching between tasks. But there is an issue with long interrupt processing at the expense of real-time. In that case, it is preferable to patch Linux with PREEMP_RT. Then, when a driver wants to manage an interrupt, it provides two functions. The first function, called directly when the IRQ arrives, must check if the IRQ concerns its driver and return IRQ_WAKE_THREAD if it's the case and IRQ_NONE if not. The second function is executed in a threaded kernel with a default FIFO 50 scheduling. PREEMPT_RT also improves kernel preemptibility (Blaess (2019), Reghenzani et al. (2019)).

4 Design and development of system

The need is for a low-cost educational temperature control system allowing to illustrate RTOS-based computer system programming, RT-bar-metal-based computer system programming, data transmissions. The environment temperature will be measured and when it exceeds a set point, a fan is controlled by a controller to regulate the difference between set and current temperatures towards zero. A camera is used to monitor the rotation of the fan. The set temperature, the image of the fan as well as the various measurements taken, such as temperature, fan speed and the value of the command sent to the fan, are displayed on a console.

4.1 Technical requirements

The operating temperature range of the system is between 3°C and 100°C. The temperature sensor must be low cost, with a conversion coefficient of the order of 10 mV/°C to achieve a measurement accuracy of ±2°C. A fan of the type used on computers fits. It is

built around a brushless motor and incorporates a control card so that it has 4 signals sufficient for its use: power supply, ground, control and speed measurement. A Raspberry camera module (PiCam) capable of producing VGA format images (640×480 pixels) will be used for the fan rotation monitoring.
The solution will involve a distributed computer system: an Arduino for the control and automation part, a first Raspberry Pi (RPi) for controlling the camera, and a second RPi for managing the both former systems and displays signals and images on its console. Both RPis will run Linux-RT.
The communication between RPi A and Arduino will be achieved with a serial link (in fact, RS-232 protocol emulated in USB protocol) with RPi acting as client and Arduino as server. The communication between RPi A and RPi B will be achieved in TCP protocol with the former acting as client of images and the latter as server of images (figure 2).

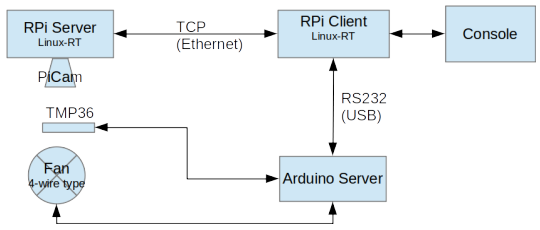

Figure 2: Specifications of the system.

4.2 Organic design

For the temperature measurement, an analog probe of the type TMP36 from Analog Devices, can be used. It is supplied with low voltage (2.7 V to 5.5 V) and consumes little current (about 50 μA), and provides a linear output voltage proportional to the measured temperature. At a temperature of -50 °C, it provides an 0V voltage at the output, then the voltage increases by 10 mV per °C, i.e. a resolution of ± 1 to 2 °C. It is therefore necessary to read an input voltage varying between 0 V and 1.75 V, corresponding linearly to a temperature read from -50°C to 125°C: the slope is 10 mV per °C with an offset of 0.5 mV (Devices (2022)).

For the fan, a processor fan of the model AUB0912VH-CX09 can satisfy the specifications. It has two wires for the power supply (12 V and earth), a wire for the speed control, accepting a PWM signal at 25 kHz, and an open collector wire sending a PWM pulse by half-turn of the fan for fan speed measurement (tachometer/speed sensor) (Electronics (2022)).

For the Arduino, the board Uno satisfies problem requirements: MCU ATmega328, 14 Digital I/O (of which 6 provide PWM output, 6 Analog I, etc. (Arduino (2022)). For RPis, the model 3B achieves a good balance between computation power and price. It features include: 1.2 GHZ quad-core ARM Cortex A53 (ARMv8 Instruction Set) CPU, Broadcom VideoCore IV @ 400 MHz GPU,1 GB LPDDR2-900 SDRAM, 10/100 MBPS Ethernet, etc. (Pi (2022a)).

The camera module (v1) is associated with RPi 3B. Its specifications include: 1/4 ” sensor format, 3.76×2.74 mm sensor size, 640×480p 60/90 (i.e. up to 60/90 frame per second), etc. (Pi (2022b)).

The communication between RPiA and RPiB on one hand and RPiA and Arduino on the other hand is achieved through Ethernet link with TCP/IP protocol and Serial link with RS-232 protocol, respectively.

Arduino will act as a controller of temperature and a server of signals. RPiB will act as a server of images obtained from the RPi camera module. Every second, the RPi client (RPiA) will ask for a VGA size image the server will send back, and every 100 ms it will get signals from Arduino.

Electronic connections of the control part includes:

- pin `A0` of Arduino receives the analog ouput of the TMP36 temperature sensor;
- pin `3` of Arduino connected to Timer 2 delivers fan PWM control signal;
- pin `2` of Arduino receives tachometer output and generates interrupts for speed measurement.

Figure 3 displays a view of the system.

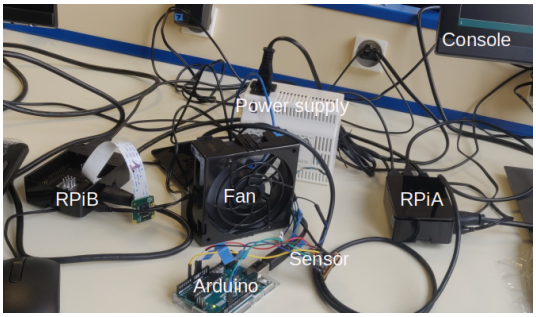

Figure 3: A view of the system.

5 Development of application

5.1 Functional design

The system will achieve the following functions:

- `Regulate_temperature`: Regulate environment temperature;
- `Inform_user`: Inform the user with fan image, set temperature, current temperature, current fan speed, control of fan (i.e. **monitoring**).

`Regulate_temperature` can be decomposed into two more functions: `Read_temperature` that reads the temperature from the sensor, and `Control_fan` that controls fan speed.
`Read_temperature` can be decomposed into `Acquire_temperature` that acquires the voltage of the sensor and `Convert_degrees` that converts the obtaining voltage into degrees.
`Control_fan` can be decomposed into `Calculate_control` that calculates the value of the control, and `Apply_control` that applies the control onto the fan.

`Inform_user` can be decomposed into: `Inform_user_signals` that displays the signals involved in the temperature regulation, and `Inform_user_image` that displays images of fan.
`Inform_user_signals` can be decomposed into above `Read_temperature`, `Read_speed` that reads fan speed, `Collect_signals` that gathers the signals (current temperature, set temperature, speed) to be send to user, `Send_RPi_signals` that sends to RPiA the signals, `Receive_signals` that receives signals from Arduino and `Send_console_signals` that sends the signals to console.

By considering input-output peripherals, temperature sensor, fan (speed sensor, motor), camera and console, one obtains the functional architecture of the system (figure 4).

With the knowledge of the physical system, it is now possible to assign functions to the different organs and to define the types of data described in functional analysis.
The application can be broken down into two parts:

1. a client-server application between the both RPis, RPiA will be the client and implements `Request_image`, `Receive_image` and `Send_console_image` whereas RPi B will be the server that will implement `Receive_request`, `Acquire_image` and `Send_RPi_image`;

2. a client-server application between RPiA and Arduino, RPi will implement `Receive_signals` and `Send_console_signals`, whereas Arduino will implement resting functions (figure 4).

5.2 Client-server application between RPiA and RPiB using TCP

RPiA runs client application. It is a periodic task: every seconde `Request_image`, `Receive_image` and `Send_console_image` are called.

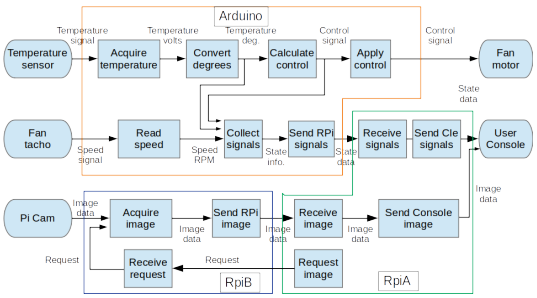

Figure 4: Assignment of functions to organs.

RPiB runs server application: it implements the functions `Receive_request`, `Acquire_image` and `Send_RPi_image`.

As stated in the specifications, TCP/IP protocol is used for the communication.

Application is a matter of two processes that communicates using sockets, the fundamentals network interfaces (Abbott (2011), Roux (2009)).

5.3 Client-server application between RPiA and Arduino using RS-232

RPiA runs client application that implements `Receive_signals` and `Send_console_signals`.

Arduino runs server application that comprises:

- implementation of `Acquire_temperature`, `Convert_degrees`, `Calculate_control` and `Apply_Control`;
- an **interruption** to deal with `Read_speed`;
- implementation of `Collect_signals` and `Send_RPi_Signals`.

The choice of interrupt is relative to the feature of the fan regarding frequency generator for rotation speed measurement (Electronics (2022)). The output circuit is an open collector that generates a 50% duty cycle (100 that multiplies high value duration over period). Assuming 4 poles fan, the output signal contains 2 periods per tour, then the speed can be computed knowing the period. The interrupt is used to measure the laps of time between the rising edge and the falling edge.

The fan control signal is of the type PWM at 25 KHz. As a consequence a timer is required to deal with it.

For serial communication the data structure `termios` available in the header `termios.h` allows the port configuration with respect to the communication protocol

(stop bit, parity bit, etc.), along with functions to set port output and input speeds, etc. (Hunter (2022)).

6 Discussion

The developed system consists of a fan, a temperature sensor, a camera, a Linux-based computer (Raspberry) and a bare-metal computer (Arduino), and a power supply. It enables to control environment temperature and to monitor all signals involved and environment image (figure 5). Associated application allows to illustrate key real-time concepts: multiple task (processes as well as threads) with different priorities, hardware interrupt with a bare-metal, timer.

After validation, the system was duplicated and has been used by the students of University of Franche-Comté for the class of real-time.

The system is cheap. Using a fan taken from an old PC, very cheap temperature sensor, open-source hardware (Raspberry and Arduino) and software (Linux, C), its final cost is less than 100 Euros.

Current version of system may be improved by assembling all components on a rigid support, that can be a simple wooden board, using 3D printed fasteners. It is also possible to replace the serial link by an Ethercat or CAN link, and the RPi server of images by a powerful board allowing to really monitor fan rotation.

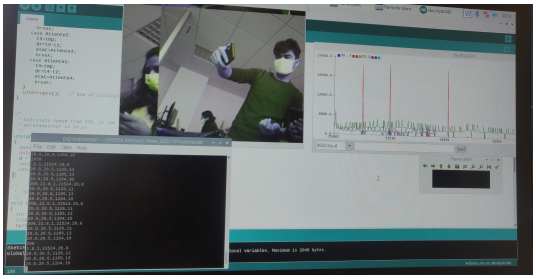

Figure 5: A screenshot showing fan speed, signals and image displays.

7 Acknowledgements

This work has been supported by the EIPHI Graduate school (contract "ANR-17-EURE-0002")

References

Abbott, D. (2011). *Linux for embedded and real-time applications*. Elsevier.

Arduino (2022). Arduino uno specifications. `https://www.arduino.cc/en/Main/arduinoBoardUno>`. (Online; accessed 2022-January-22).

Blaess, C. (2005). *Programmation système en C sous Linux: Signaux, processus, threads, IPC et sockets*. Editions Eyrolles.

Blaess, C. (2019). *Solutions temps réel sous Linux*. Eyrolles.

Devices, A. (2022). Tmp36, voltage output temperature sensors. `https://www.analog.com/en/products/tmp36.html#product-overview`. (Online; accessed 2022-January-22).

Didalab (2022). Véhicule multiplexé didactique. `http://www.didalab-didactique.fr/site/upload/FR_130115_125011_ME_Y8vXbH.pdf`. (Online; accessed 2022-January-25).

Electronics, D. (2022). Aub0912vh-cx09 datasheet. `https://pdf1.alldatasheet.com/datasheet-pdf/view/943936/DELTA/AUB0912VH-CX09.html`. (Online; accessed 2022-January-22).

Grolleau, E., Hugues, J., Yassine, O., and Henri, B. (2018). *Introduction aux systèmes embarqués temps réel: Conception et mise en oeuvre*. Dunod.

Hunter, G. (2022). Linux serial ports using c/c++. `https://blog.mbedded.ninja/programming/operating-systems/linux/linux-serial-ports-using-c-cpp/`. (Online; accessed 2022-January-23).

Oliveira, P. M. and Hedengren, J. D. (2019). An apmonitor temperature lab pid control experiment for undergraduate students. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 790–797.

Pi, R. (2022a). Raspberry pi 3 model b. `https://www.raspberrypi.com/products/raspberry-pi-3-model-b/`. (Online; accessed 2022-January-23).

Pi, R. (2022b). Raspberry pi camera module 2. `https://www.raspberrypi.com/products/camera-module-v2/`. (Online; accessed 2022-January-23).

Reghenzani, F., Massari, G., and Fornaciari, W. (2019). The real-time linux kernel: A survey on preempt_rt. *ACM Computing Surveys (CSUR)*, 52(1):18.

Roux, B. (2009). *Les sockets en C*. Developpez.com.