EMBEDDED SYSTEM SOFTWARE ARCHITECTURES IN MULTIPROCESSOR SYSTEMS

By Andreu Sabé i Cruixent
Software Architect in *SALICRU*

salicru

Introduction

For some years now, due to the increase in the level of integration of the devices and the decrease in their cost, it has been possible to see a proliferation of embedded electronic systems with more than one processor. This has meant distributing the intelligence over one or several electronic boards in the same system through various combinations of MPU, MCU, DSP or DSC:

- Single-core processors.
- Multi-core processors.
- A combination of the previous two.

This has provided, among others, the following benefits:

- Increased computing power.
- Redundancy.
- Use of processors specialised in different tasks, as for example, combining a MCU and a DSP.

From the firmware point of view, it has been approached in different ways:

- Developing totally independent tasks, whether they are real time or not.
- Working as a peripheral of another processor.
- Allowing real parallelism in the same code.
- Running different firmware with coordinated tasks between multiple processors.

Several methods have been used for the communication between devices:

- Communication buses: UART, I2C, SPI, CAN…
- Shared RAM memory.
- Specialised Inter-Processor Communication Peripherals (IPC).

In previous documents [1] [2], the advantages of using Embedded System Software Architectures (ESSA) were highlighted. In the present one we will address, in an introductory way, the role that these architectures can play in systems with more than one processor, which, as we said, are increasingly common.

[1] Embedded Software Architecture.

[2] Virtualisation and Embedded System Software Architectures.

Short description of an Embedded System Software Architecture (ESSA)

Before going into the main subject, let us make a brief review of the main building blocks of an ESSA. For a more detailed description, read the above-mentioned documents [1] [2

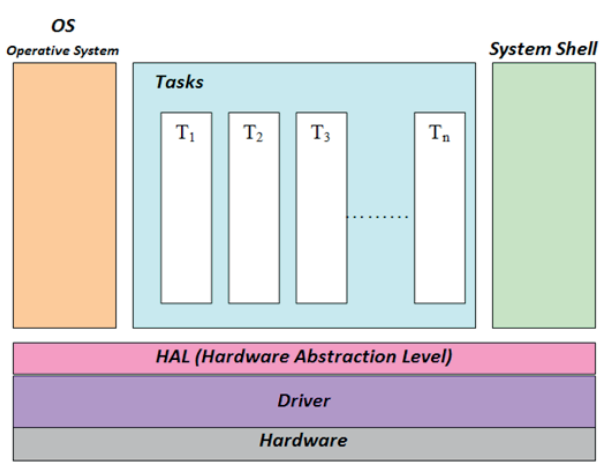

Figure 1: ESSA block diagram

As we saw, ESSA is constituted by a module O.S. which is responsible for executing the different tasks of the embedded application in addition to providing various services (file system, RTC, timers, access control, etc.), the already mentioned tasks, a Shell module responsible for the interaction of architecture with the external world and a Hardware Abstraction Layer (HAL) that interacts with the Drivers and eases the portability of the software architecture between different devices.

ESSA in multiprocessor systems

If the embedded system has more than one processor, either in the same device or in different devices in the same electronic board, it is logical to think of applying the same architecture to all of them. However, in a system like this, the different processors often interact with each other to carry out a common task and here it is possible to ask what else an ESSA can do for us. To begin with, let us consider a system with two processors, each of them with the same architecture:

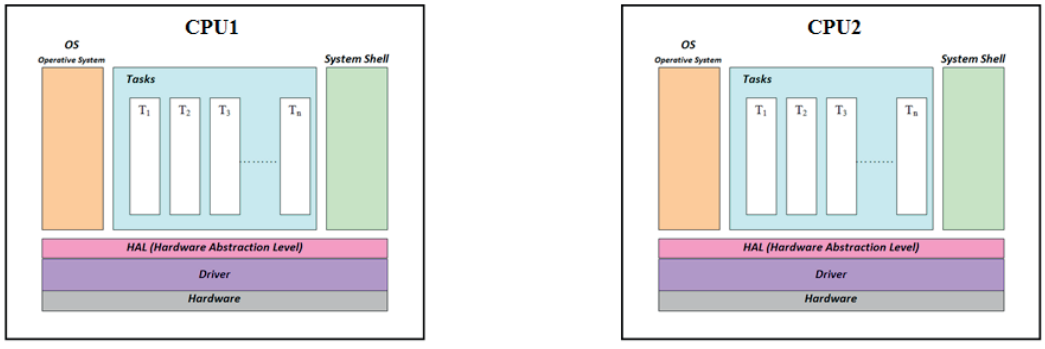

Figure 2: Diagram of a two-processor system with the same ESSA

[1] Embedded Software Architecture.

[2] Virtualisation and Embedded System Software Architectures.

The first step is to provide the ESSA with a messaging system between both processors. This system allows the communication between different tasks, no matter which of the two CPUs they are running in. This adds a new abstraction layer so the tasks do not need to be aware of where the other tasks they are communicating with are located. Then, from the point of view of the tasks, the system can be considered as single one.

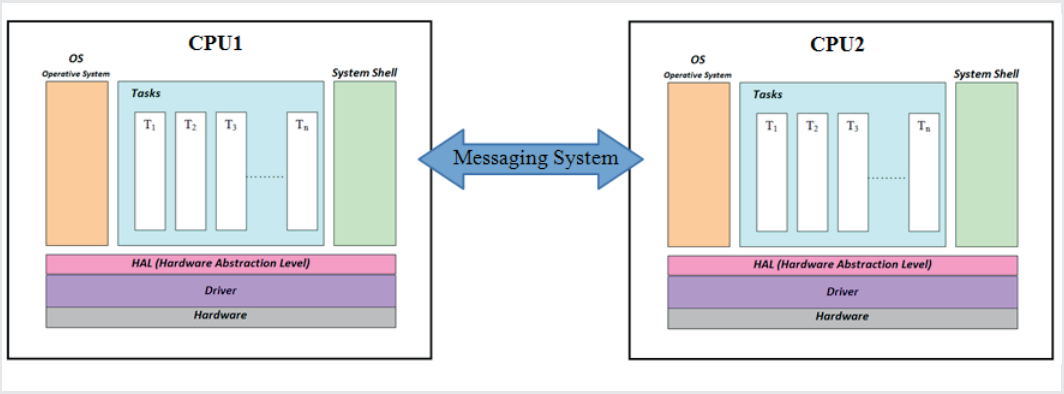

Figure 3: Figure 2 with added messaging system

The next step is to interconnect the HAL layers of both systems, what allows peripheral sharing between the devices. Then, if a task in a certain CPU requires a peripheral that is in the other one, this is not a problem at all, since the HAL is shared.

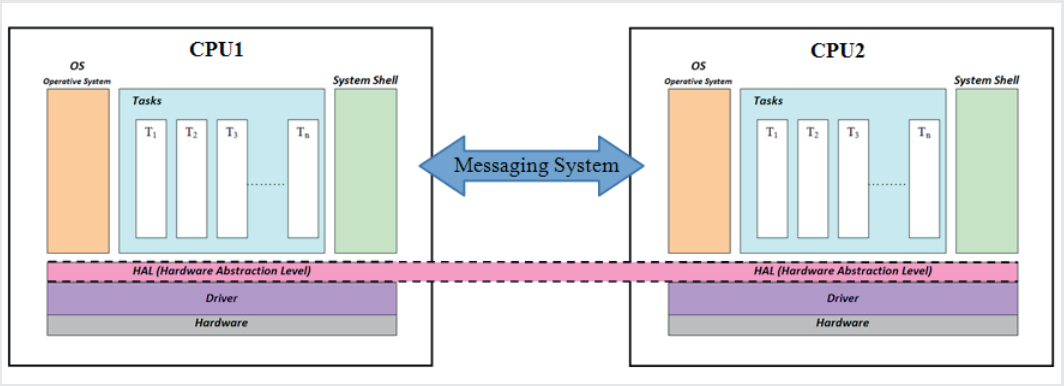

Figure 4: Figure 3 with added shared HAL layer

It is clear that in a system like this, the different processors cannot either start/stop randomly. A synchronization procedure must be established that, in our opinion, it is only possible at O.S level since this is the only one that holds the complete control of the system and can start/stop the tasks in due course. For that purpose, the messaging system will be used.

One of the issues that can generate the most controversy is the existence of two Shell interfaces in the system, which poses a contradiction with the idea of the system as a single one. Then, ¿it is not a single Shell enough? In our opinion, it depends on which stage of the development we are in. If the development has already finished, one can think that one is quite enough.

However, mainly during the first stages of the development, it will be very interesting to be able to work with single CPUs in order to focus on the implementation of the specific task of each one of them. In this case, having a Shell in each processor is a real need.

Anyway, a system like this can be highly configurable and (in due course) disable one of the Shell interfaces leaving the other one active, which can be used to interact with the whole system from the outside.

So far, the explanation has been made taking a two-processor system as a base, even though, its philosophy is, as a matter of fact, generalizable to more complex systems with a wide variety of multiprocessor system topologies. As an example, a system with four parts (CPU1, CPU2, CPU3 and CPU4) is given. Here CPU1 acts as system Master with two CPUs slaves, CPU2 and CPU3, which is in turn Master of CPU4:

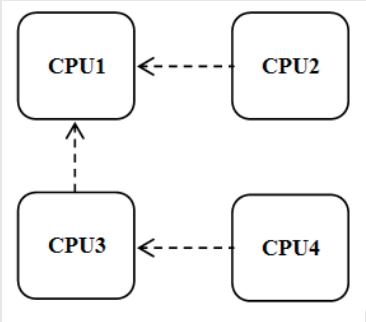

Figure 5: Example of a four-CPU system. The direction of the arrows shows the dependency between them

From this description, the system can be defined depending on the hierarchy between the different processors:

Device	Hierarchy
CPU1	System Master
CPU2	Slave of CPU1
CPU3	Slave of CPU1
CPU4	Slave of CPU3

Table 1: Definition of the system in figure 5 taking its hierarchy into account

Another way to understand the system is that CPU2 and CPU3 work as CPU1 complex peripherals and CPU4 plays the same role for CPU3

Nevertheless, just as it was said above, a system like this cannot boot randomly. When powering up the system, all the CPUs start and they are aware of its position in the hierarchy of the system (stated in previous table), which will define its behaviour. Here, the simplification of a possible boot sequence, from the point of

Figure 6

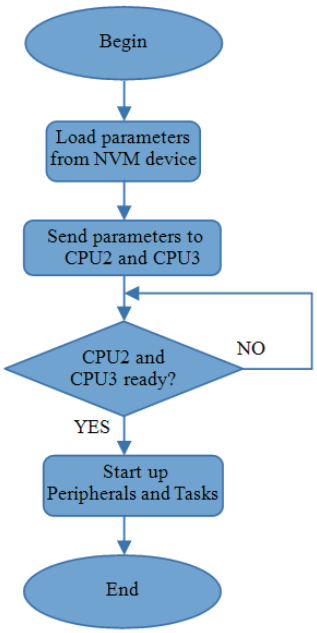

Figure 6. CPU1 boot sequence

Figure 7

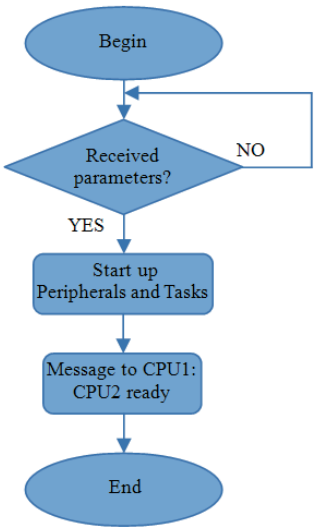

Figure 7. CPU2 boot sequence

Figure 8

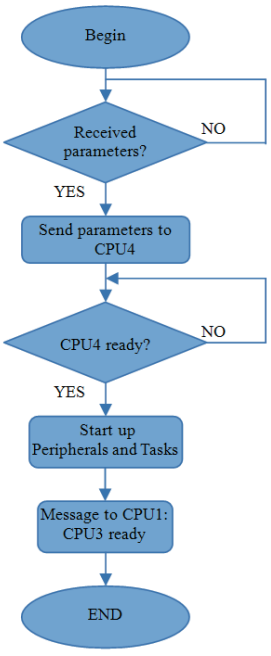

Figure 8. CPU3 boot sequence

Figure 9

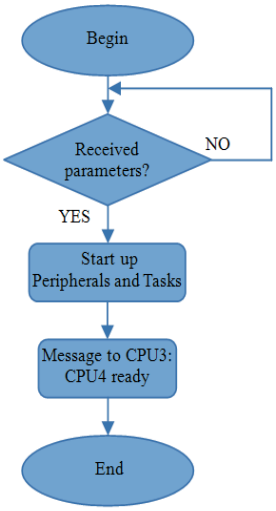

Figure 9. CPU4 boot sequence

From these four diagrams, the following can be underlined:

- CPU1, as a system Master, is in charge of loading the setting parameters from a NVM device and pass them (directly or indirectly) to the other processors.
- CPU2 and CPU4 diagrams are the same.
- As for CPU3, given its status as CPU1 slave and CPU4 master, its diagram is based on a combination of the diagrams of these two.
- The start-up of peripherals and tasks is performed in reverse order with respect to their hierarchy in the system: CPU2, CPU4, CPU3 and finally CPU1.

From these diagrams it is also possible to deduce a possible stop sequence of the system, which we leave for the reader.

One of the keys of ESSA is the reuse of the software in the different developments of the company. This implies that the management of the multiprocessor system has to be considered in a generic way and be able to work with diverse hardware topologies. It must also provide an easy method of defining the system and the relationships between the different CPUs. A basic example of this would be the format shown in Table 1, which could be enhanced with other data: priority within the same hierarchical level, data sharing system, etc.

Other considerations in the development of the system are those intended to ease the debugging of the whole system:

- With ESSA intrinsic debugging capabilities (stop/start of tasks/drivers, simulation of events through the Shell, etc.).
- Making the detection and management of errors less strict.
- Allowing the incorporation of new CPUs to the system, even after finishing the start/stop sequences.
- With the virtualization of the different CPUs and their communications systems on a PC, allowing the debugging of the system without the need of the definitive hardware.

.