Software Architectures in Embedded Systems

Iivo Raitahila
University of Helsinki
Department of Computer Science
https://www.cs.helsinki.fi/iivo.raitahila

***Abstract*—Embedded devices are tiny computers that are usually built into larger apparatus. They may for example control room temperature in air conditioners, tuners in televisions, networking devices and Internet of Things devices, such as toasters. It is typical that the devices interact with various sensors and actuators external to the computer and such it is normal to have hard real-time deadlines - for example in the braking system of a vehicle the code must be executed in a specific time. The devices may have cost, physical size, power consumption and heat emission constraints. Because there are no resources to be wasted, programmers have been unwilling to use any software architecture used with general purpose computers as the regular architectures come with too much overhead. The lack of a suitable architecture results in a "big ball of mud" architecture that is difficult to maintain. This paper presents simple architectures for very small programs and more complex component based architectures for larger programs that enable modularity, reusability and portability.**

I. Introduction

Most embedded devices are built into products, like televisions [1] and cars [2]. An example of a separately sold very low end device is the Arduino Uno that has an ATmega328 microchip, 2KB of RAM, 32KB of flash memory and 14 digital input/output pins and 6 analog pins for interacting with the outside world [3]. An example of a more powerful embedded device is the Arduino compatible Intel Galileo board with a single core 400MHz Intel Quark processor, 256MB of RAM, 8MB of flash memory, 20 digital input/output pins and 6 analog pins. The Galileo also has Ethernet connectivity [4].

The aim of this paper is to present some software architectures to use with such devices.

The main quality features for embedded software are modularity, reusability and portability [5]. Modularity means that independent modules can be combined into a full system reducing its complexity. Reusability means that parts of the system can be reused in similar systems or contexts reducing development effort. Portability means that the software can be easily ported to different operating systems and hardware. For embedded real-time systems also timeliness, predictability and efficiency are important. These attributes mean that the real-time constraints must be enforced, the system must work as intended with given input and that any limited resources are not wasted excessively. All of these quality features can be achieved with a small impact on efficiency and timeliness.

The fundamentals section covers some simple architectures as prerequisite information and operating systems that are necessary for some larger programs. The component-based architectures are discussed in the advanced architectures section. The architectures discussed in subsections are the Koala component model for consumer electronics, the AUTOSAR architecture for automotive applications and for general use the COMDES-II and FASA/FASAlight architectures. The architectures are compared using the aforementioned quality features.

II. Methods

The base of this paper was the article [5] from the WICSA 2016 conference that presents the FASA/FASAlight architecture. The references of that article were browsed through. Different databases were used in search for more articles. The databases linked to some random sites and to some more distinguished ones (mainly IEEE and ACM in this case). Some recurring authors were noticed, like Ivica Crnkovic (a professor of software engineering), and for example papers he had co-authored were considered as a reliable source of information. This paper was started from top to bottom: the WICSA paper represents the latest architecture and then similar or a bit older architectures were searched for comparison. Architectures that seemed outdated were not considered as candidates for this paper.

Some search strings used were "embedded software architectures" and "component based approach". After finding out some architectures, searches were conducted using the names of the architectures. In the AUTOSAR architecture the main source of information was their website, but some conference articles were found, too. Those articles used AUTOSAR's website as one of their references.

The architectures that were selected provide a small insight, but there are lots of other architectures not discussed in this paper.

III. Fundamentals

A. Simple architectures

Systems that have few tasks to do and lots of time can manage with a simple architecture as described in [6].

The simplest one is the round robin architecture depicted in listing 1. It has an infinite loop where each device is polled in turn and acted upon the result. After all the devices are checked through, the loop starts again from the first device. This works when the loop is executed very quickly so that the user does not mind the delay of for example when pressing a button and the screen updating as a consequence. If some devices have a higher priority, they can be checked multiple times in the loop.

The advantage of round robin is its simplicity but it scales up poorly and has performance constraints.

```
void main(void) {
   while(TRUE) {
      if (device_1 requires service)
         service device_1
      if (device_2 requires service)
         service device_2
      if (device_n requires service)
         service device_n
   }
}
```

Listing 1. Round robin, adapted from [6]

The round robin architecture can be complemented with interrupts, so that an interrupt service routine can handle urgent tasks when they occur. The interrupt service routine can also be implemented so that it does only the most essential bits of the urgent task and the main loop does the rest when the turn of the urgent task arrives. This approach reduces the response time of urgent tasks. Shared data problems must be taken care of, since an interrupt may occur while some other function is handling shared data.

In function queue scheduling architecture, after the interrupt service routine has executed the most essential tasks, it puts a function pointer into a queue of tasks for later processing. The main loop then goes through the queue if it is not empty. The queue can be sorted by priority for example. This way immediate response can be given promptly to all and the rest of the task is processed later on in a desired order. If the queue is sorted by priority, starvation of low priority tasks may occur if high priority tasks are inserted always before a low priority task has had a chance to execute.

Developers may end up with anti-patterns when trying to get the most out of the hardware [5]. Such bad practices are for example the use of global variables to exchange data between functions (might lead to race conditions) and using platform specific API calls (code is not portable). The use of an advanced architecture helps in this regard.

B. Operating systems

As described in [6], the simple loop architectures do not require an operating system. In fact such small systems do better without an operating system since it uses some of the scarce memory and CPU time. Real-time operating systems (RTOS) have a scheduler that is capable of handling real-time constraints. The flexibility of the scheduler is one of the benefits of using an operating system. Tasks can be easily added and removed and the scheduler selects a task to run based on its priority. With a preemptive multitasking system, a lower priority task can be suspended while running so that a higher priority task can start executing very quickly.

IV. Advanced architectures

Monolithic and platform-dependent software is hard to port, upgrade and reuse. Component technology brings many advantages [7]. The article [8] reports that component based development has been a success in many application domains, such as desktop and distributed applications. A component could be classified as a reusable, independently developed self-contained part used as a building block of a system. Component implementation is separated from component interface. Only the interface is visible to the rest of the system - this is called information hiding or encapsulation.

Large embedded systems are complex and thus interoperability is more important than a small performance gain. The applied architecture is similar to the ones used in general applications, such as with desktop applications. Large embedded systems are usually only soft real time systems, but a reduced component model is usable with a real time operating system.

Programmers may be unwilling to use any software architecture because of their overhead. The measurements in [5] report that the FASAlight architecture causes execution performance penalty of less than 14% compared to a native implementation without a modern architecture. With large amount of function blocks (50-100), FASAlight uses 10% more RAM, which in case of 100 blocks is just 160 bytes total. There is also a small initialization time when starting the program.

The subsections present a couple of component based architectures. Some architectures are useful in general purpose software (such as FASA) and some are for specialized use (for example AUTOSAR is for automotive use). A comparison based on the quality features of each architecture is shown in table I.

It seems that there is no clearly defined boundary to when a simple loop architecture is sufficient and when an advanced architecture is required. For example in the article [2] a case study with AUTOSAR is an application that controls five interior lights of a car. Even that system is complex enough to benefit from an advanced architecture.

A. Koala

The Koala Component Model was developed for consumer electronics [1]. The researchers found out that the size of embedded software doubles every year, the diversity of products increases and development time has to be decreased. They saw that the solution is to use and reuse components within an explicit software architecture. Reusing software saves development time and that is the key element of the Koala architecture. Encapsulating code as libraries is not sufficient with low level code reuse. Architectural description language is used to visualize the selected components and structure.

In the Koala model a component is a unit of design. A product is called a configuration and it consists of components. The component developers do not assume any configuration and configuration designers do not modify the internals of any component. Components are developed by various developers that can store the components into a web-based repository. Multiple components can be combined to form a compound component. A compound component acts just like a normal component and it simplifies large programs.

TABLE I
COMPARISON OF QUALITY FEATURES IN THE DISCUSSED ARCHITECTURES

Architecture	Reusability	Portability	Modularity	Real-time
Simple loop	Some functions	N/A	Function definitions	Small program / interrupts
Koala	Components	N/A	Interface definitions	Unknown
AUTOSAR	Application software components	Run time environment	Standardized interfaces	Task priority
COMDES-II	Actors and function blocks	N/A	I/O communication and physical drivers	Fixed-priority timed multitasking scheduling policy
FASA and FASAlight	Component and block model	Platform abstraction layer	Channels (interfaces)	Time-triggered cyclic execution and static schedules

A component communicates through an interface, which is a small set of functions defined in C syntax. A component is defined with component description language, where its provided interfaces and the interfaces it requires to function are listed. These two are illustrated in listing 2 code example that is part of figure 1. Components' requirements must be connected to a component that provides that service, unless the requirement is marked optional. One component can provide service to multiple components.

```
interface ITuner
{
        void SetFrequency(int f);
        int GetFrequency(void);
}

component CTunerDriver
{
        provides ITuner ptun;
                 IInit pini;
        requires II2c ri2c;
}
```

Listing 2. Interface and component definitions, adapted from [1]

Modules combine the interfaces of multiple components. It can be used for example to call the initialization interfaces of multiple components, or to combine multiple components to provide such an interface that any of the components do not provide on their own.

Figure 1 represents Koala's graphical notation and the general idea is applicable also to other architectures. The CTvPlatform is a compound component that contains CFrontEnd, CTunerDriver and CHipDriver components with instance names cfre, ctun and chip respectively. Interfaces are marked with arrows and the direction of the arrow tells whether the component provides service through that interface or requires it in order to function: the CTvPlatform provides IProgram and IInit interfaces and requires an II2c interface. For example the function of the CTunerDriver is to control a television tuner hardware via a serial I2C bus to tune in on a specific channel supplied by a front-end component through the ITuner

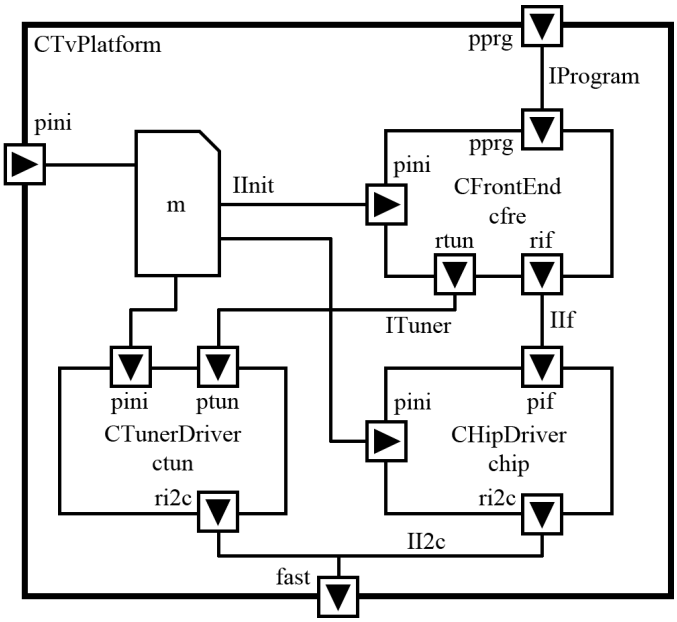

Fig. 1. Koala's graphical notation, adapted from [1]

interface. Note how the required II2c interface can be used by more than one component, but the provided IInit interface must be split using a module m.

B. AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is an open standard for automotive software architecture [2]. The key points of AUTOSAR are cost-efficiency, quality, reusability and managing complexity. Traditional software developed for automotives electronic control units is hardly reusable and the software is getting increasingly complex. It is possible to convert traditional software into AUTOSAR architecture by separating the existing software into AUTOSAR components and rearranging them accordingly.

The AUTOSAR architecture consists of application software components, the run time environment and basic software modules. The run time environment separates the application software components from the infrastructure that is the basic

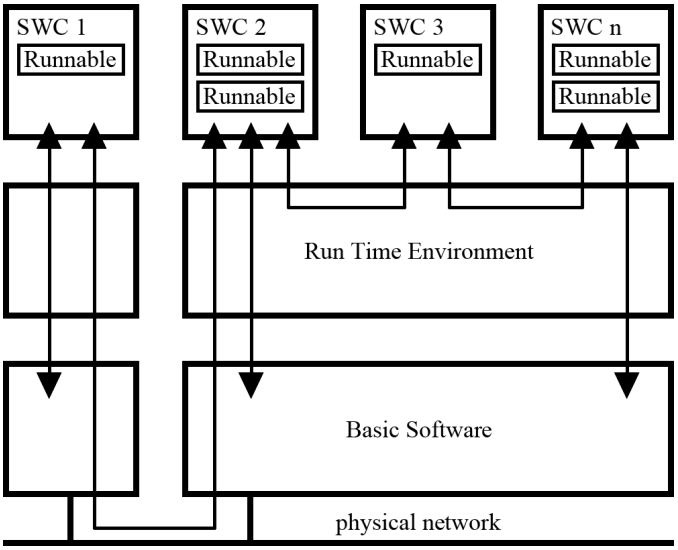

Fig. 2. Overview of the AUTOSAR architecture, adapted from [2]

software and physical network while improving reusability. Application software components are run in an electronic control unit (an embedded computer). The application software components consist of runnables that are grouped into tasks for the operating system.

The underlying basic software modules have an operating system and communication features that are clearly defined to provide compatibility between different vendors. The interfaces that the components use are standardized and XML (Extensible Markup Language) is used as the data format.

Figure 2 depicts the layers of the AUTOSAR architecture. Application software components (SWCs) communicate only via the run time environment - whether they communicate with each other or with the operating system, IO drivers, etc. in the basic software. If the application software components are running in different electronic control units, the system routes the calls accordingly as in the case of SWC1 and SWC2.

The AUTOSAR architecture supports multi-core systems, too [9]. With multi-core computing, one electronic control unit can run multiple applications and applications with higher system requirements are possible, for example vision-based advanced driver assistant systems that include video processing and image recognition. To run multiple applications on one device, embedded virtualization technology can be used. The device is then split into smaller virtual machines that can run different operating systems and software. The hypervisor can run either on top of or without an operating system, where the former can include significant overhead.

C. COMDES-II

The COMDES-II (COMponent-based design of software for Distributed Embedded Systems - version II) is also a component-based architecture that is designed for distributed embedded systems which need hard real-time support [10]. It also includes analysis techniques, advanced algorithms and data structures, automatic code generation and compilation techniques.

An application consists of actors that communicate using labeled messages. An actor has a task and multiple communication and physical drivers. The task contains multiple function blocks that contain the actual code and are reusable.

Actors contain the non-functional information. Actors are activated when a periodic or an aperiodic event is triggered. First input is read, then it is channelled to the task within the actor and finally output is written. No new input is read during the task execution. The task will be executed when the operating system schedules it to run. The task can be preempted during execution as long as it can finish before its deadline.

Function blocks that implement the concrete computation or algorithms have four different types. Basic type have inputs, outputs, parameters and internal variables. Composite function blocks include multiple basic or composite function blocks and are used to achieve complex features. While basic and composite function blocks model continuous behaviour, state machine function blocks preserve history data and thus can model sequential behaviour. Modal function blocks are jointly used with state machine function blocks, since modal function blocks act upon the state of a state machine function block. One state machine function block can control multiple modal function blocks.

The hierarchical nature of COMDES-II is shown in figure 3. On the top of the hierarchy is the environment consisting of data lines connected to the application. Circles denote input interfaces and squares denote output interfaces. The labeled messages the actors use to communicate are marked as msg_1-3. In the middle of the hierarchy is the actor. The actor of the example has three communication drivers (white triangles) and one physical driver (black triangle). The information from the drivers is transmitted to the task using local signals (im1_v etc.). At the bottom of the hierarchy is the task and its function block instances. In this example, the function blocks are either basic or composite function blocks since state machine and modal function blocks need to be connected to each other using two signal lanes - one for the state update message and one for the new state.

D. FASA and FASAlight

FASA (Future Automation Systems Architecture) and FASAlight, a version for low-end embedded devices, are proposed in [5]. FASA supports multi-core and distributed systems.

Applications are built as function blocks that are scheduled and monitored by a runtime framework layer. Below the runtime framework layer is a platform abstraction layer that can be accessed from the runtime framework and function blocks. The platform abstraction layer communicates with the operating system. These layers are illustrated in figure 4.

Function blocks contain the application code and have an interface called ports. Input ports and output ports are

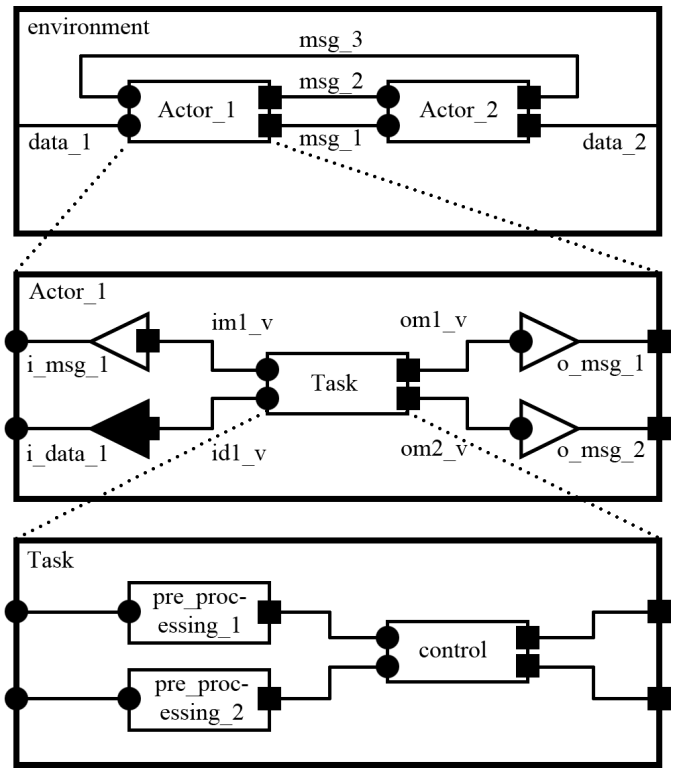

Fig. 3. Hierarchical architecture model of a COMDES-II system, adapted from [10]

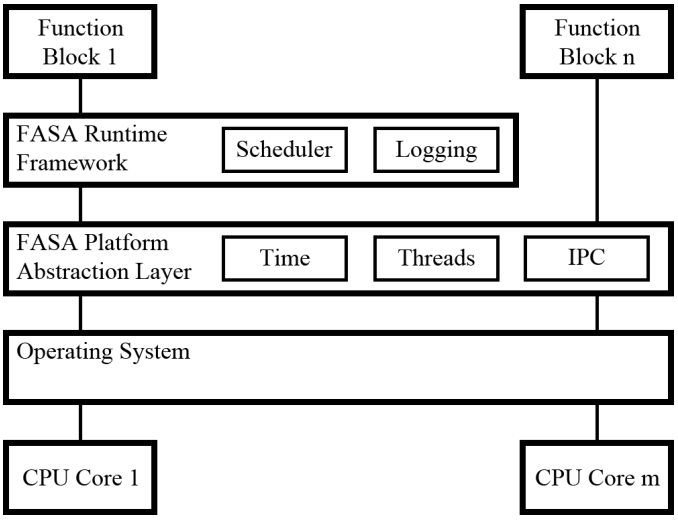

Fig. 4. FASA layered architecture, adapted from [5]

connected via a channel. A group of blocks is called a component. Components can be executed in parallel, so CPU bound function blocks should be organized into different components if the CPU has multiple cores.

The runtime framework consists of components such as a scheduling component that is also interchangeable. The reference implementation uses non-preemptive static cyclic scheduling (a fixed sequence), but other approaches are possible.

The FASAlight version is designed for devices that have as little as several kilobytes of RAM. Function blocks have only two methods: initialization and execution. Channels are pointers to a shared memory area. The platform abstraction layer has been implemented on Linux and Freescale MQX operating systems. Configuration is done using hard-coded initialization code.

The normal FASA has been implemented on Linux for x86 and ARM processors with multi-core support. Configuration is done using XML files, which was a too resource intensive approach for the FASAlight version. The FASA reference implementation has 20989 lines of code and the FASAlight has only 2402 lines.

The components and function blocks are portable if they do not use platform specific functionality, but only the platform abstraction layer. They are also interchangeable between FASA and FASAlight. The platform abstraction layer can be ported to different operating systems and processors with little effort. The programming language needs to have a compiler in the target system.

Both FASA and FASAlight have all the most important quality features plus the ones suitable for real-time systems (timeliness, predictability and efficiency).

V. Discussion

The architectures presented were compared in table I. Code reusability was present in each of the component based architectures - in simple loop architecture some functions might be reusable, but generally not. Proper portability is achieved with an effortlessly exchangeable platform abstraction layer that was present in AUTOSAR and FASA/FASAlight architectures. Achieving portability in other architectures means rewriting hardware or operating system dependent components or functions. Modularity can be achieved in every architecture - in fact the interface definitions of Koala use the same C syntax that can be used in the header files of a simple loop program written in C. In the simple loop architecture, real-time constraints can be met when the program is small enough and has a lot of spare time. AUTOSAR, COMDES-II and FASA/FASAlight use a real-time scheduler. Koala most likely has real-time support too, since for example televisions maintain a certain frames per second speed, but the technique it uses was not mentioned.

There are many more architectures and many more differences between them than discussed in this paper. The methods could have been different, for example IEEE hosts an IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) and its latest conference papers could have served as the basis of the architecture selection (the article [10] was published in that conference). There were also some architectures that were considered but weren't included because of their old age.

Some future work might include studying architectures that use a scripting language, since script languages offer flexibility usually with a performance penalty that may be unacceptable with embedded devices. Design patterns that are small enough

not to be considered as architectures but that provide some of the quality features discussed in the paper would also be interesting. For future systems, architectures that have emphasis on networking (Internet of Things) are something to study about. Background systems, such as real-time operating systems should not be forgotten about.

VI. Conclusions

A few software architectures for embedded systems were presented in this paper. The kind of software architecture that should be used when developing for an embedded device depends on the function of the apparatus, the performance of the embedded computer, special requirements etc. Very small programs that have not much to do and plenty of time can manage with only a simple loop architecture. If the program is not an extremely small one, a more advanced component based architecture should be used. Such architectures help manage the complexity of the program and allow the reuse of code in later projects or ports to different platforms.

References

[1] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The koala component model for consumer electronics software," *IEEE Computer*, vol. 33, no. 3, pp. 78–85, Mar 2000.

[2] D. Kum, G. M. Park, S. Lee, and W. Jung, "Autosar migration from existing automotive software," in *2008 International Conference on Control, Automation and Systems*, Oct 2008, pp. 558–562.

[3] "Arduino AG," http://arduino.org, accessed: 2017-03-01.

[4] "Intel Corporation," http://intel.com, accessed: 2017-03-01.

[5] A. Monot, M. Oriol, C. Schneider, and M. Wahler, "Modern software architecture for embedded real-time devices: High value, little overhead," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, April 2016, pp. 201–210.

[6] J. Cook and J. Freudenberg, *Embedded Software Architecture*. EECS University of Michigan, 2007. [Online]. Available: http://www.eecs.umich.edu/courses/eecs461/lecture/SWArchitecture.pdf

[7] I. Crnkovic, *Building Reliable Component-Based Software Systems*, M. Larsson, Ed. Norwood, MA, USA: Artech House, Inc., 2002.

[8] ——, "Component-based software engineering for embedded systems," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 712–713. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062631

[9] P. Gai and M. Violante, "Automotive embedded software architecture in the multi-core age," in *2016 21th IEEE European Test Symposium (ETS)*, May 2016, pp. 1–8.

[10] X. Ke, K. Sierszecki, and C. Angelov, "Comdes-ii: A component-based framework for generative development of distributed real-time control systems," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, Aug 2007, pp. 199–208.