ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ARCHITECTURE AND ENGINEERING

DEPARTEMENT OF

Electrical, Electronic, and Information Engineering "Guglielmo Marconi" - DEI

MASTER'S DEGREE IN

ADVANCED AUTOMOTIVE ELECTRONIC ENGINEERING

MULTICORE SOFTWARE DEVELOPMENT FOR ENGINE CONTROL UNITS

Master Thesis
in
Hardware-Software Design of Embedded Systems M.I.C. – Real Time OS

Candidate
Massimo Toscanelli

Supervisor
Chiar.mo Prof. Paolo Torroni

Co-supervisor
Ing. Saverio Cortese

Academic Year 2018/2019

Session II

Contents

1. Introduction

1.1. The context

System requirements in automotive context are becoming increasingly more restrictive. Their complexity is growing up very fast and this has a remarkable impact on functional elements used in software. The integration of the different software modules, that represent the operations of a control device, is a very expensive and error prone phase of the entire system production line. Moreover, the limitation of system components is often translated in scalability or maintenance difficulties. Specific adjustments and several versions are the reason why reusability and standardization are hardly achievable and often many producers reinvest in new concept platforms to solve this problem. For this purpose, the AUTOSAR partnership was born, gathering the consent and participation of many companies in the automotive sector.

1.2. Magneti Marelli use case

Faced with the need of car manufacturers to have control units capable of performing increasingly difficult real-time tasks, Magneti Marelli, as tier one supplier, has made several hardware and software upgrades to their systems during the years. One of the most important was the switch to multi-core microcontrollers. This change implied also a significant software adaptation to exploit as much as possible this new hardware potential. Therefore, the operating systems have been suitably modified by introducing inter-core communication services that allow the cooperation of the cores and the consequent increase in the workload of the system. The company adopted AUTOSAR architecture first in single-core ECUs (Engine Control Units) and then in multi-core ones, that are natively supported since AUTOSAR version 4.0, offering its own standard for inter-core communication: the “IOC”. However, Magneti Marelli wanted to develop a better-fitted solution for its use cases: a custom “IOC” component that could substitute the standard one and speed up executions that requires a multi-core utilization.

1.3. The project

The aim of this project was to design a proper inter-core communicator (ICC) for Magneti Marelli ECUs with AUTOSAR architecture. Once studied the best algorithm to optimize the data transfer, we tested it in a simulated environment that we created ad-hoc. Verified its correctness, we implemented it in embedded C code to be flashed in control unit, so that we could also validate our solution.

At the end of this process, we designed and developed a code generator, based on that code structure, that can automatically read configuration files of an AUTOSAR project and produce the C code of a properly configured ICC.

This project will be described in the thesis with the following structure:

- Initially AUTOSAR is introduced, together with its architecture, its methodology, the implementation used in Magneti Marelli and the related development tools.
- Afterwards, we focus on the study of the inter-core communication. In particular, the current solution is compared with the one we propose.
- The embedded software development process is the next topic, in which we show all the implementation steps that led us to test our model directly in the control unit.
- In the end, all the design and development phases of the code generator are described and a brief guide to its use is also provided.

2. AUTOSAR

2.1. What is AUTOSAR

As specified in [2] and [3], the AUTOSAR (AUTomotive Open System ARchitecture) platform is born from the activity of the homonymous consortium born in 2003 from a group of important actors in the automotive scene. From that time, the industrial realities that joined the consortium are so many, sharing the common need for regulation of the sector.

AUTOSAR is a standard born to divide the dependency of the applicative functions from the hardware platform where they are running. This allows to simplify their transferability and to reduce adjustment expenses for producers' requirements.

Thanks to well defined interfaces and a unified architecture, maintenance, update and interchangeability of software components can be easily guaranteed for the entire system lifecycle.

Having the same applicative modules on different hardware platforms, increases the growing of software suppliers that are specialized into single sectors. In fact, in AUTOSAR systems we often find components provided by different suppliers and car manufacturers act just as integrators. With an adequate organization of the process chain, a fruitful collaboration and communication channels defined in partnership, it is possible to reduce the iteration cycles and management costs and, consequently, general costs and development time.

AUTOSAR promises benefits also in software quality, which is becoming an aspect of considerable size in a context where certifications are increasingly required according to the Spice automotive standard (ISO / IEC 15504) or CMMI (Capability Maturity Model Integration). Indeed, the combination of qualitative structures within the partnership between semiconductor manufacturers and software houses contributes to reduce the percentage of errors and make a solid integration between software and hardware.

Finally, adequate partnerships and fruitful collaborative relationships further pave the way and facilitate the creation and market introduction of a complete AUTOSAR car.

The consortium members tend to the theoretical limit in which, if the process of abstraction concerns all the devices of a motor vehicle, the same management software can equip many types of cars with a simple configuration of some performances.

2.2. History and organization

The consortium, born from an almost exclusively Teutonic will (BMW, Bosch, Continental, Daimler, Chrysler, Siemens VDO and Volkswagen), has expanded, gathering more and more support and involving today more than two hundred industrial companies that participate in fundamental way to develop and define the platform ([4]).

However, the contribution of partners varies depending on the type of partnership:

- **Core Partners**
- **Premium Partners**
- **Development Partners**
- **Associate Partners**
- **Attendee**

As core partners we find BMW, Bosch, Continental, Daimler AG, Ford, General Motors, PSA Peugeot Citroën, Toyota and Volkswagen. They are responsible for organization, administration and control of the AUTOSAR development partnership.

Premium and Development members are allowed to work on packages coordinated and monitored by the Project Leader Team, established by the Core Partners. Associate partners make use of the standard documents AUTOSAR has already released. Attendees participate with Academic collaboration and non-commercial projects.

AUTOSAR project milestones are phase 1 (from 2003 to 2006), phase 2 (from 2007 to 2009), phase 3 (from 2010 to 2012). After them, we find continuous further development that includes the stabilization of Classic Platform and, since 2017, the improvement of the Adaptive Platform. The first kind of platform addresses the needs of deeply embedded ECUs, whose software is designed and implemented for a target vehicle and does not change fundamentally during vehicle lifetime. Future vehicle functions, such as highly automated driving, will introduce highly complex and

computing resource demanding software into the vehicles and must fulfil strict integrity and security requirements. Therefore, AUTOSAR specifies Adaptive Platform, which provides mainly high-performance computing and communication mechanisms and offers flexible software configuration, e.g. to support software update over-the-air.

Since 2009 (version 4.0), AUTOSAR has supported systems with multicore processors. However, the OS can still only execute a single thread at a time, which means that the OS has to be replicated on each core. Moreover, AUTOSAR only allows static task allocation, meaning that tasks are not allowed to migrate between cores.

However, what is interesting to note is the growing acceptance of the standard at manufacturers and suppliers that today allows us to say that more than 80% of the cars sold in the world are built by members of the AUTOSAR consortium.

2.3. AUTOSAR Architecture

The structure, based on the AUTOSAR software components, uses a layered architecture (defined in [5]) to free the functionality from the hardware and from software services of the system.

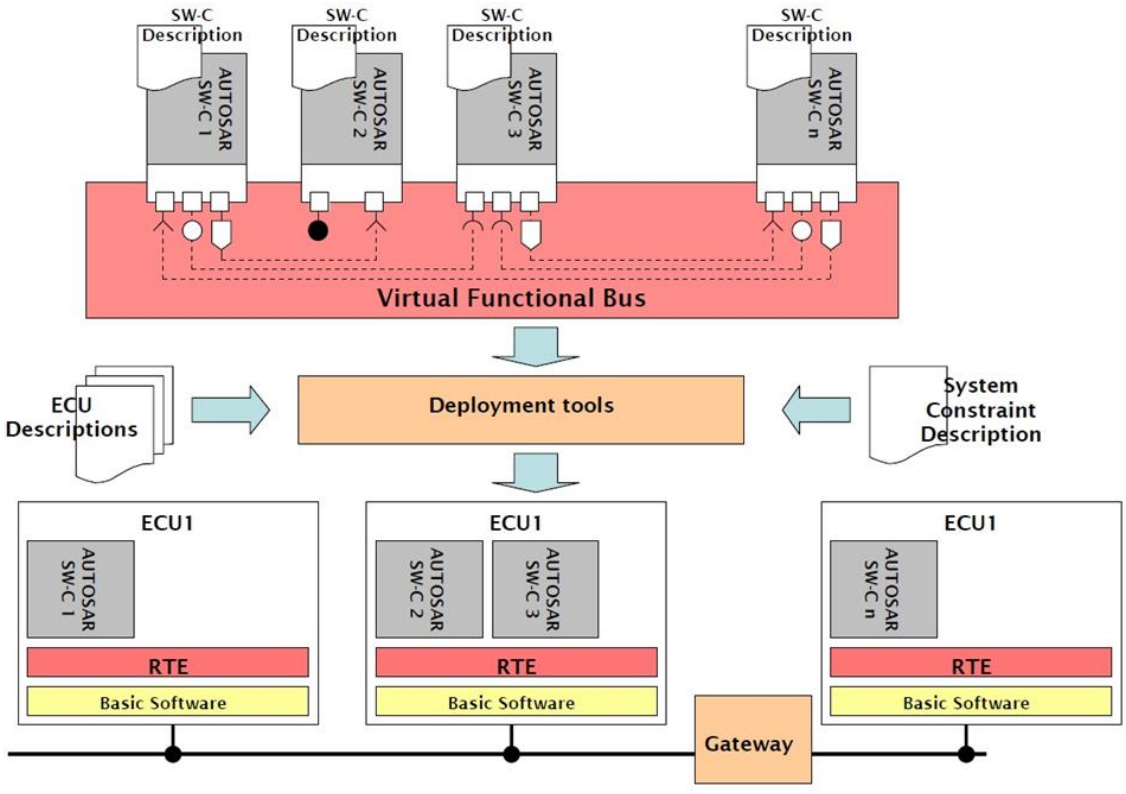

Figure 1: AUTOSAR Layered Software Architecture

2.3.1. Software Components (SW-C)

In AUTOSAR infrastructures, applications run on Software Components (AUTOSAR SW-C) that have well defined and standardized interfaces. Their standard description format is called SW-C Description.

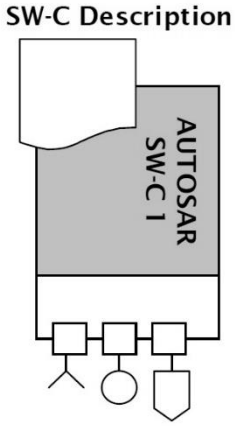

Figure 2: SW-C

Each SW-C is “atomic”, meaning that its instance is statically assigned to one ECU. Moreover, AUTOSAR does not specifies whether a component must be handwritten or automatically generated.

A Software Component Description that specifies the infrastructure configuration for the component, and an implementation, that can be given as “object code or source code”, compose a shipment of a SW-C.

A SW-C Description is structured with a “software component template” that includes:

- PortInterfaces that describe operations and data elements that the SW-C needs
- Requirements on the infrastructure
- Resources required by SW-C
- Information regarding the specific implantation of the SW-C

The source code component implementation is independent on the type of ECU it is mapped in, on the number of its instances and on the location of the other components with which it interacts.

Components interact each other through well-defined ports. Interface concept is introduced to define services or data that a port provides or requires. An AUTOSAR Interface can be Client-Server, defining a set of operations that can be invoked, or Sender-Receiver, for data-oriented communication.

In Client-Server communication, the server is a provider and the client is a user of a service. Who starts the communication is the client that requests a service to the server and then it can be blocked (synchronous communication) or non-blocked (asynchronous communication) until a response is received. On the other side, the server waits for incoming requests, executes the service and send back a response to the client.

The Sender-Receiver communication is asynchronous: the sender distributes information to one or more receivers and, meanwhile, it can continue its execution not expecting any response. The sender is unaware of number of receivers, it just provides information and the communication infrastructure is in charge of distributing it.

Requirements and capabilities of data exchanged between components are defined through Communication attributes and Application level attributes. The first ones specify parameters of the communication that are meaningful for the RTE generation or the real runtime communication (ex. transfer time over a connector). Application level attributes instead, are just indications for developer on how data must be processed, for example if data is “filtered” or “raw”.

2.3.2. Runnable Entities

Runnable Entities are software functions that implement the component behaviour and, at the same time, are the smallest pieces of code that a component can reference. They are subject to OS scheduling as part of OS tasks.

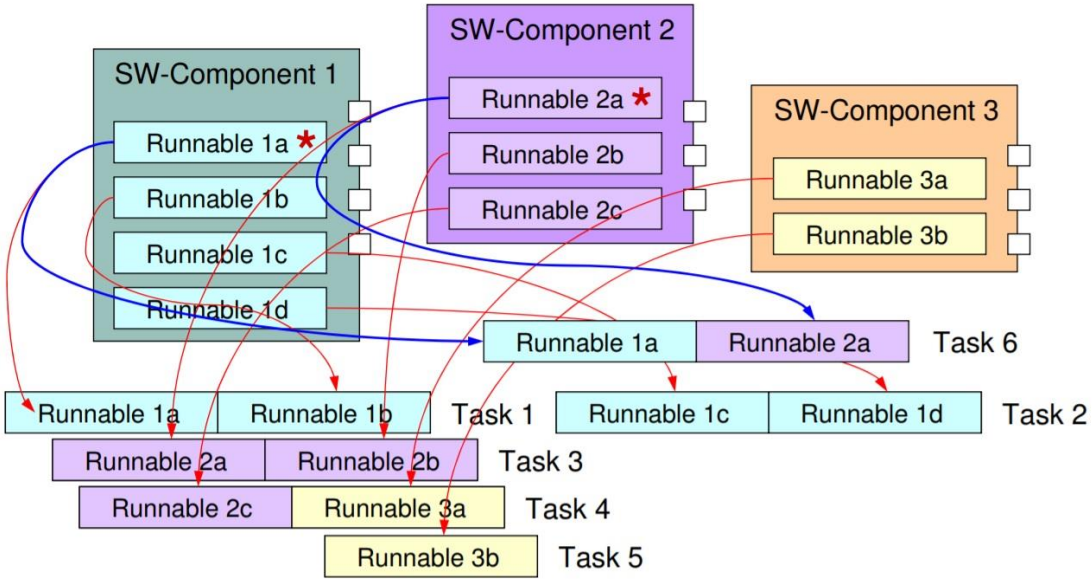

Figure 3: Runnables' mapping into tasks

Each Runnable has a “canBeInvokedConcurrently” option that is FALSE by default. If it is set TRUE, we can have more instances of the same Runnable that run at the same time in different tasks with no single state associated. Without any explicit constraint imposed to the OS, it can freely preempt every Runnable. Furthermore, all code called by every Runnable must be reentrant.

Each Runnable has access to the port interfaces and can read/write data signals from/to other software components. A Runnable execution is triggered by a data receive event (when new data is available on its sender-receiver port) or by a timing event (timer trigger).

2.3.3. Virtual Functional Bus (VFB)

Every communication mechanism provided by the architecture is abstracted with this technology independent level. VFB allows virtual connections between components in order to define a system since its early development phase. Here we find the

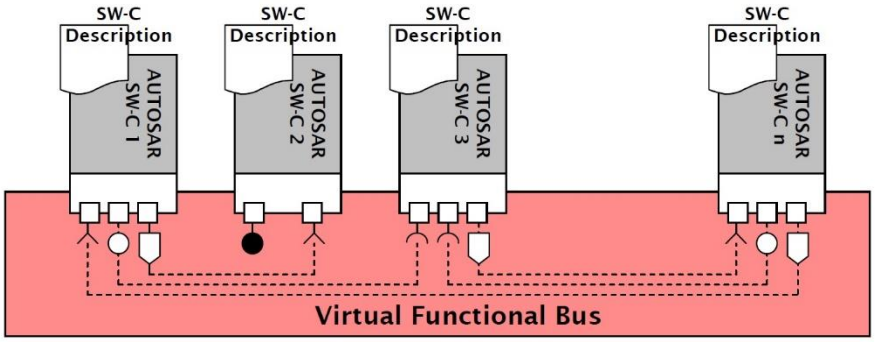

Figure 4: Virtual Functional Bus

description of components with the means of datatypes, interfaces, hierarchical components, ports and connections between them.

The functionality of the VFB is provided by communication patterns.

2.3.4. System Constraint and ECU Descriptions

In vehicles, we find many interconnected ECUs with different resources and configurations. To integrate them with SW-Components, AUTOSAR provides their description formats together with the system description. It also defines the methodology and the tool support to build a concrete system of ECUs, meaning the configuration and generation of RTE and BSW on each ECU.

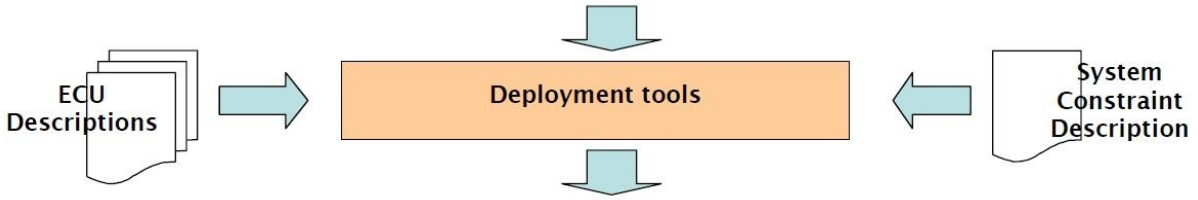

Figure 5: System Constraint and ECU Descriptions

2.3.5. Basic Software (BSW)

This layer has no other specific features besides making the top layer (Runtime Environment) independent of the system hardware. This function is implemented through specific APIs. Obviously, this layer is dependent on the system hardware.

2.3.5.1. BSW Conformance classes

During the migration period to next-generation automotive systems AUTOSAR and NON-AUTOSAR software are mixed together. That is why three implementation conformance classes (ICCs) are defined for the BSW, where we find modules' interfaces that are AUTOSAR-compliant, so that it is not necessary to implement each module as unit of its own. In this way, ICCs affect BSW and RTE, but not the ASW (Application Software).

- **ICC1**: It is the first step of the migration, in which RTE and BSW are inside the same cluster, and only the interface between RTE and ASW and the one to the bus must be AUTOSAR-compliant. RTE and BSW implementations are proprietary, but we need to take care that they have a standardized AUTOSAR behaviour.
- **ICC2**: Clusters divide related modules that must have AUTOSAR-compliant interfaces. RTE has its own cluster. BSW clusters from different vendors can be integrated together.
- **ICC3**: No clustering of modules. It is the most compatible AUTOSAR level: all AUTOSAR compliant BSW modules are present with the specified interface.

2.3.6. Runtime Environment

It manages the data exchange between the software components and the connections between the application, the system hardware and the various software components.

2.4. ECU Architecture

AUTOSAR has a layered architecture that allows clear and structured interface definition and a precise hardware abstraction. We have five main layers plus the Complex Drivers.

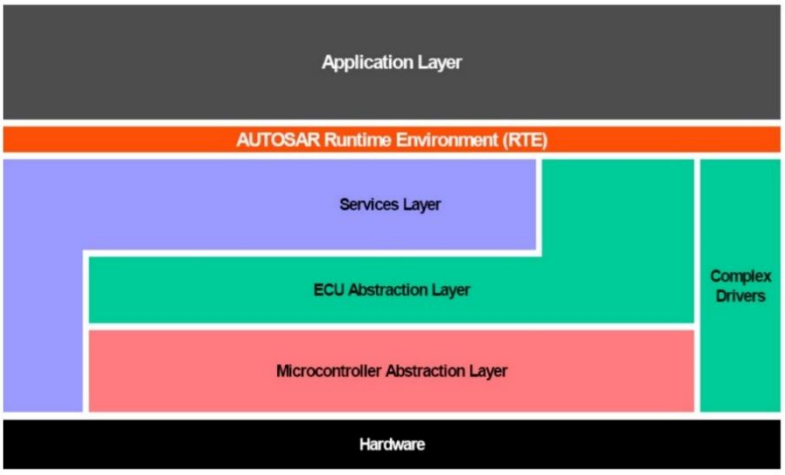

Figure 6: ECU Architecture

2.4.1. Application layer

This layer consists of AUTOSAR Software Components that are mapped on the ECUs. Their interaction is routed through the AUTOSAR Runtime Environment. The AUTOSAR Interface specification assures the connectivity.

This is the only layer not composed of standardized software, because it is the one in which the application resides. This approach, based on software functionality, allows the definition of the "vehicle system" ignoring whether two software components are belonging to the same ECU or not. "Low" software layers have the responsibility to connect the components and to guarantee their access to hardware resources. The sequence of operations used to define the "vehicle system" in all its components can be summarized in the following steps.

2.4.2. Inputs description

The inputs description can be divided into three sections: the first is the formal description of the software components (independent of the implementation of the software component itself), whose interfaces and hardware requirements are specified. Then follows the description of the system topology (interconnection between the various ECUs) described together with the available data buses, the used protocols, the clustering of functions and the specific characteristics of the devices such as bus speed, timing, latencies etc. It is then necessary to define the hardware structure of the system (processors, actuators, sensors) and any particularities regarding signal processing and device programmability.

2.4.3. System configuration

In this phase the software components are attributed to the various ECUs through an iterative process that must take into account the resources available and the limits of the system (for example if the communication speeds allow the subdivision of a software component on two different ECUs and so on).

2.4.4. ECU configuration

In this phase the Basic Software and Runtime Environment layers of each ECU are configured. Obviously, this configuration is based on the assignment of the SW components to each ECU.

2.4.5. Executables generation

At this point, it is possible to generate executables for each ECU and, of course, it is necessary to define the specific behaviour of each SW component. This methodology is automated thanks to the use of special software that allows the management of every single step of the process. All actions taken up to the generation of executables are supported by the definition of standard data interchange formats using XML. To support the AUTOSAR method, a meta-model in UML was developed containing the formal description of all methods and related information. This methodology allows a clear and immediate visualization of the information structure, the guarantee of the information consistency and the enormous facilitation of system software maintenance.

2.4.6. Runtime Environment

The Runtime Environment is the runtime representation of the Virtual Function Bus for a specific ECU. It abstracts the connection of Software Components providing the same interface and services for inter-ECU or intra-ECU communication. Being communication requirements very application dependent, RTE is tailored generated to offer desired services and, at the same time, to be resource-efficient. Therefore, RTE is usually tool-generated, statically configured and very ECU dependent.

The RTE-generator creates the right APIs based on the definition of each Software Component Template. Not to change components' code when mapping is modified, the API has to be independent from mapping. The API names must be compliant to a

naming convention and are read from XML files. RTE-generator also implements connectors between ports; this piece of generated code is dependent on the mapping of SW-C to the ECU and. It creates a communication stub that can be local, if two connected components are on the same ECU, or, otherwise, it can use network communication. The last one is also responsible for parameter marshalling, so the serialization of complex data to a byte stream, even if who eventually performs the endian connection is the Basic Software.

RTE is also responsible for the lifecycle management of AUTOSAR Software Components invoking their start-up and shutdown functions. Furthermore, SW-Cs cannot directly access Basic Software, but thanks to RTE generated APIs it can become possible.

2.4.7. Basic Software

AUTOSAR Basic Software is below the RTE, which provides services to SW-Cs, but does not fulfil any functional job. It includes standardized components (about services and communication) and ECU specific ones (Operating system, Microcontroller abstraction, Complex Device Drivers).

We can find a further refined layered architecture inside the Basic Software: there are around 80 Basic Software modules subdivided into 11 main blocks plus Complex Drivers.

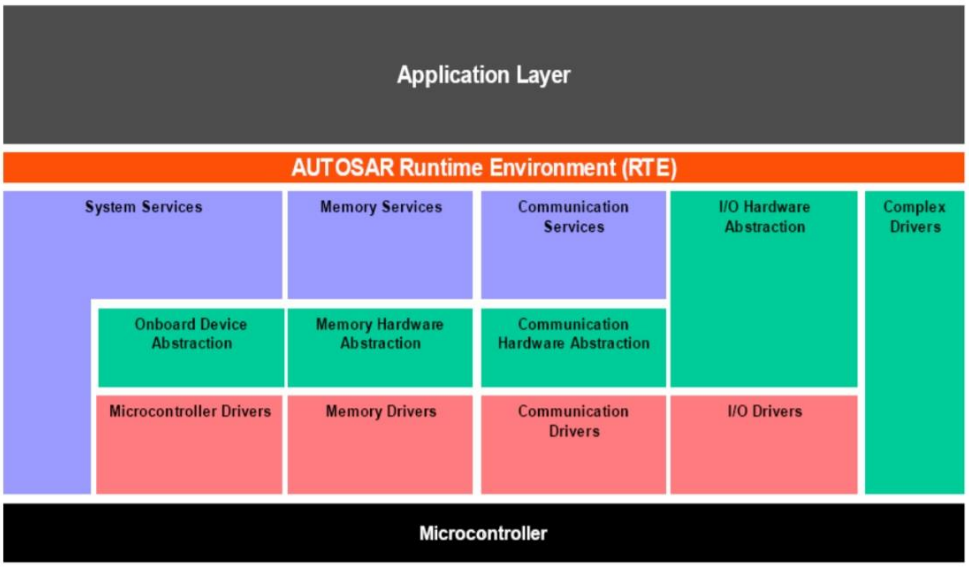

Figure 7: BSW layered architecture

2.4.8. Operating System

The OS that we can find inside is compliant with AUTOSAR Operating System requirements. It must be a real-time OS (RTOS), with priority-based scheduling and support to protective functions at run-time; it must be configured and scaled statically, and hostable on low-end controllers with and without external resources.

The basis for AUTOSAR OS is the standard OSEK OS (ISO 17356-3), but a proprietary OS can be also allowed as long as it is abstracted to an AUTOSAR OS, this means having interfaces to AUTOSAR components that are AUTOSAR compliant.

AUTOSAR has adopted a fixed priority preemptive scheduling policy. The unit of execution of the OS is called OS-Task, it has an assigned priority and can be preempted by OS-Tasks with higher priority.

There are two types of OS-Task:

- Basic: it can be in one of three states: ready (it waits for the allocation of the processor), running (it is executing its instruction), or suspended (when it has returned or terminated)
- Extended: it has one more state with respect to the ones of the basic task. A system service can block and put into ready state the task. This one can only be activated and put into ready state by an event like a received data or an expired timer.

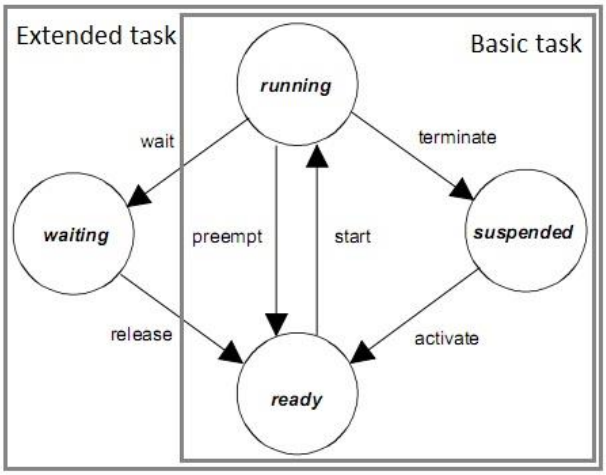

Figure 8: Types of tasks

Every Runnable defined in the system must be mapped to an OS-Task that can accept a multiple Runnable assignment. The simplest solution could be mapping each Runnable to its own OS-Task, but actually it is not feasible, because in many systems the number of tasks is limited and task switching would imply a considerable utilization overhead of the core.

In multi-core ECUs, the standard specifies that each core is independently scheduled and a task of different cores cannot preempt each other.

The OS-Application is a collection of OS-objects: OS-Tasks, ISR (Interrupt Service Routines), alarms, events, etc. It can be trusted, if its objects have unrestricted access to the API and hardware resources, or untrusted, if the access is limited and they run in non-privileged mode.

An OS-Application has its own memory partition, separate stack, data and code. AUTOSAR assures that a code executed in the context of an OS-Application cannot corrupt memory area of another OS-Application.

2.4.9. Microcontroller Abstraction Layer

The Microcontroller Abstraction Layer (MCAL) is a very hardware specific layer, the lowest one of the Basic Software. It acts as standard interface that manages microcontroller peripherals and provides BSW components with microcontroller independent values. Thanks to the notification mechanism, it also supports distribution of commands, responses and information to processes.

The MCAL is composed by:

- I/O Drivers: Drivers for analog and digital I/O (e.g. ADC, PWM, DIO)
- Communication Drivers: Drivers for ECU onboard (e.g. SPI, I2C) and vehicle communication (e.g. CAN). OSI-Layer: Part of Data Link Layer
- Memory Drivers: Drivers for on-chip memory devices (e.g. internal Flash, internal EEPROM) and memory mapped external memory devices (e.g. external Flash).
- Microcontroller Drivers: Drivers for internal peripherals (e.g. Watchdog, Clock Unit) and functions with direct µC access (e.g. RAM test, Core test)

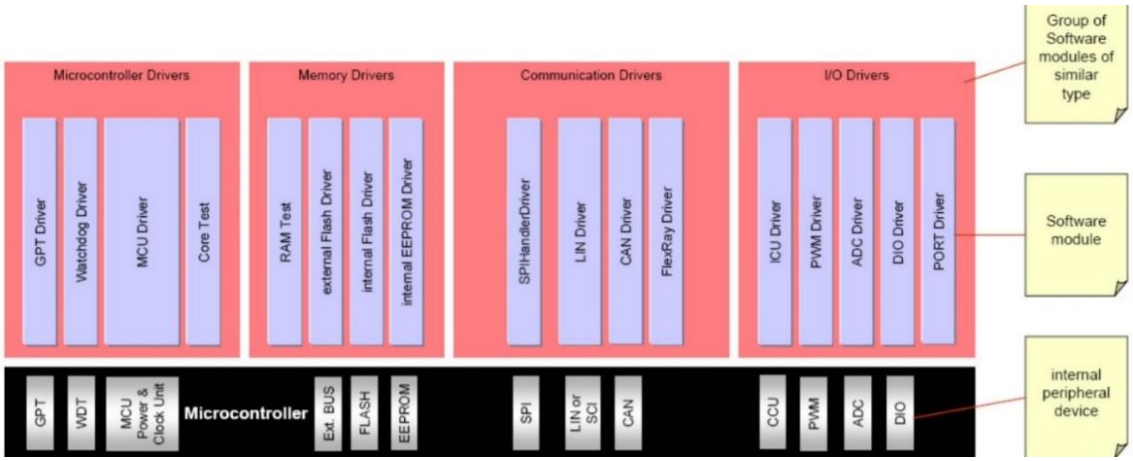

Figure 9: MCAL schema

2.4.10. ECU Abstraction layer

The ECU Abstraction Layer is the interface to electrical values of any specific ECU. It provides the complete separation between hardware dependencies and higher software level.

This layer is subdivided into:

- I/O Hardware Abstraction: this section is in charge of representing I/O signals as they are connected to the ECU hardware (e.g. current, voltage, frequency) and it hides ECU hardware and layout properties from higher software layers.
- Communication HW abstraction: it is a group of modules that provide equal mechanisms to access a bus channel regardless of its location (on-chip / onboard).
- Memory HW Abstraction: The task of this group of modules is to provide equal mechanisms to access internal (on-chip) and external (onboard) memory devices.
- Onboard Device Abstraction: its task is to abstract from ECU specific onboard devices.

2.4.11. Service layer

Service Layer is composed by:

- Communication Services: these modules make use of drivers through the Communication HW Abstraction. Their role is to hide protocol and message properties from the application and to provide a uniform interface to the vehicle network (for communication between different applications and for diagnostic communication) and uniform services for network management.

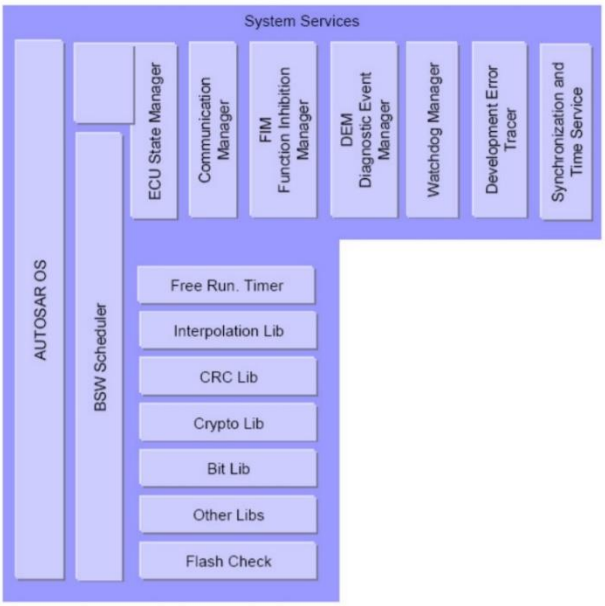

Figure 10: Service Layer

- Memory Services: they manage non-volatile data, being responsible of read/write operations from different memory drivers. A fast-read access can be performed thanks to the NVRAM manager that, with a RAM mirroring, provides a data interface to the application.
- System Services: the task of this group of modules is, in general, to provide basic services that can be µC dependent (like OS), ECU hardware and/or application dependent (like ECU state manager, DCM) or hardware and µC independent.

2.4.12. Complex Drivers

A Complex Driver is a container where specific software implementations can be placed, provided that their port and interfaces are compliant with the AUTOSAR specification. Complex Drivers are mostly used to perform complex sensor evaluation and actuator control with direct access to specific interrupts and complex microcontroller peripherals. In addition, we can use Complex Drivers to implement drivers for hardware not supported by AUTOSAR or to extend the AUTOSAR standard adding software that will not force the OEM nor the supplier to reengineer all existing applications.

2.4.13. Communication

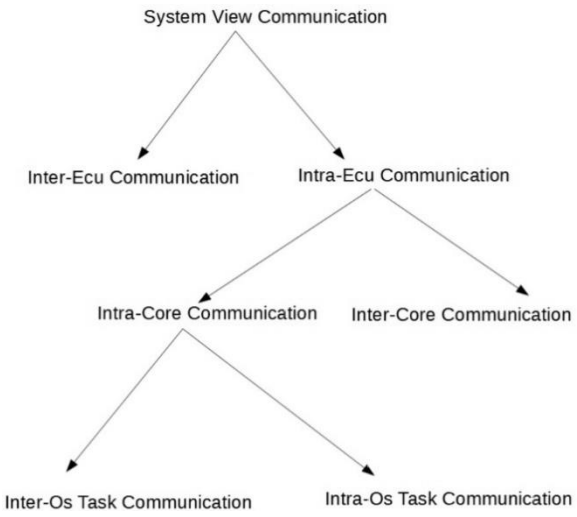

Figure 11: Communication hierarchy

As already mentioned, the sender-receiver pattern is an asynchronous type of communication in which the sender Runnable transmits data through its component P-Port and one or more receivers consume what received through their component R-Port.

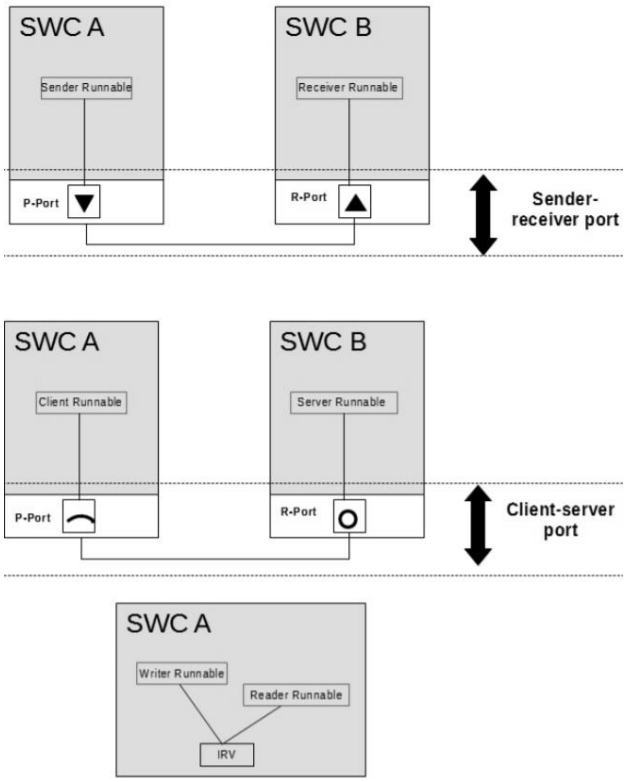

Figure 12: Sender-Receiver and Client-Server ports

Sender-receiver can be:

Implicit: sender Runnable sends just the latest data of the signal after its execution, the RTE generates a copy of the information and then starts the receiver Runnable that will use that copy for its entire execution time.

Explicit: a sender Runnable can transmit data whenever he wants, calling the RTE API. Each call corresponds to a different data transmission that can be queued or unqueued. In the first case data signal is retrieved in FIFO (first-in first-out) order, otherwise its latest value is read (last-is-best semantic). The reading happens each time a receiver Runnable decide to read data.

The client-server paradigm, instead, provides a communication in which one or more client Runnable invoke the service of a server Runnable that executes requests in FIFO order. The communication can be synchronous (blocking) or asynchronous (non-blocking).

The inter-runnable pattern is considered as a special case of sender-receiver. Runnables belonging to the same software component communicate asynchronously accessing the same inter-runnable variable.

Communication types can be classified depending on SW-Cs' and Runnables' mapping to ECUs, cores and OS-Tasks.

2.4.13.1. Inter-ECU communication

If Runnables are mapped to different ECUs, RTE layer performs an inter-ECU communication relying on modules such as Communication Stack (COM) to send data over a physical network

2.4.13.2. Inter-Core communication

In multi-core systems, BSW modules can be subdivided (or repeated) in several partitions and each of them can only be present in one core. Not standardized communication services allow an inter-partition linking.

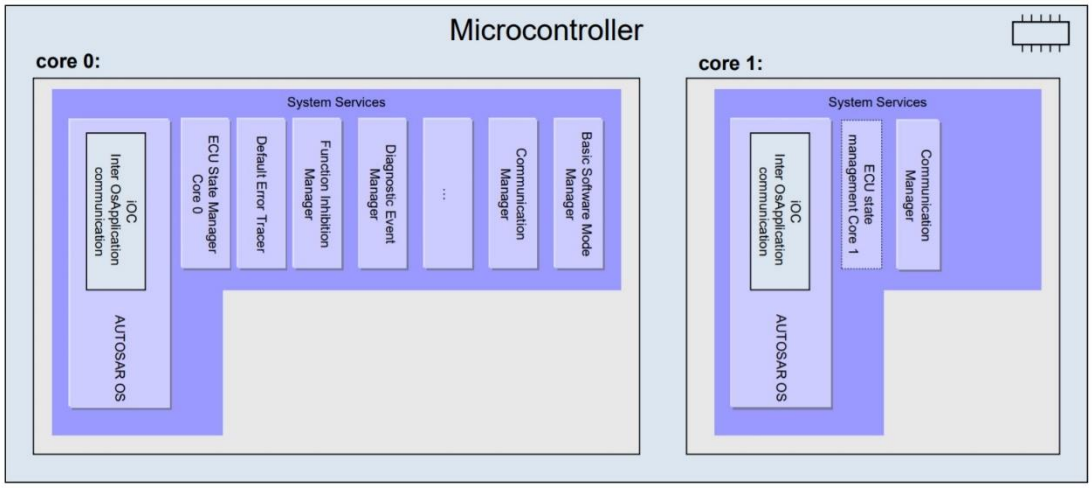

Figure 13: System services in two cores

Every OS-Application is connected with the others thanks to the IOC (Inter OS-Application Communication), which provides proper services for crossing core communication and memory protection boundaries. Therefore, Runnables mapped to different cores communicate between them with the help of the RTE and the IOC layer.

Considering that OS-Applications can be or not in different cores, the inter-core communication is always an inter-OS-Application communication, but not vice versa.

IOC internal functionality is dependent on hardware architecture properties, in particular on the memory architecture. To guarantee data consistency, the content of all data sent in one communication operation and (in queued communication) the sequential order of communication operations shall remain unchanged.

The IOC provides sender-receiver communication only. Therefore, the RTE translates ClientServer invocations and response transmissions into Sender-Receiver communication.

1:1, N:1 and N:M (unqueued only) communication are supported by the IOC.

The IOC allows the transfer of one data item (that can be a data structure) per atomic communication operation. It does not need to know the internal data structure, the

basic memory address and length is sufficient. Transferring more than one data item in one operation is only supported for 1:1 communication. The advantage compared to sequential IOC calls is that mechanisms to open memory protection boundaries and to notify the receiver have to be executed just once. Additionally, all data items are guaranteed to be consistent, because they are transferred in one atomic operation.

The IOC provides both, unqueued (Last-is-Best) or queued (First-InFirst-Out) communication operations. It can optionally notifie the receiver as soon as the transferred data is available for access on the receiver side by calling a configured callback function.

Depending on the hardware architecture and other constraints, different implementation options might be available within the IOC. In systems with shared memory, there can be a specific communication buffer for each data item in a memory section, which is shared between the sending and receiving OS-Applications.

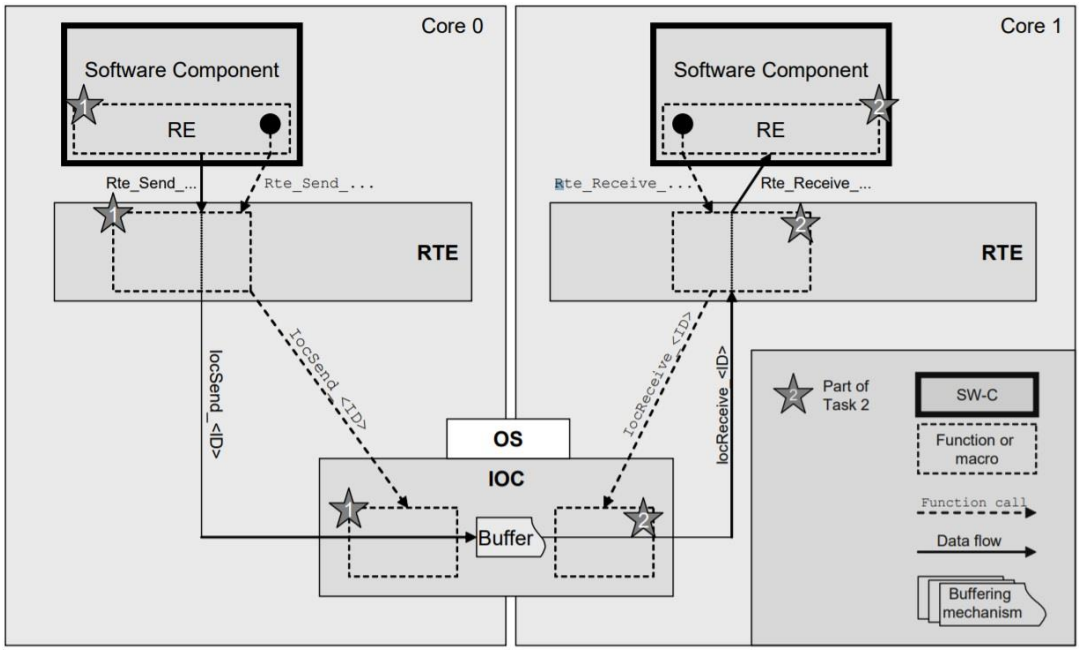

Figure 14: IOC schema

2.4.13.3. Intra-task and Inter-task communication

Intra-task communication is provided by the RTE when Runnables that are exchanging data are mapped on the same OS-Task. Inter-task communication, instead, happens when Runnables are mapped to different OS-Tasks on the same core. It is worth to mention that the mapping order into the OS-Task is very important; data sending, for example, must be performed before data reading.

2.5. AUTOSAR Methodology

AUTOSAR Methodology is the description of principal steps of system development required by AUTOSAR standard. Design phases go from the system-level configuration to the generation of ECU Executable. AUTOSAR Methodology does not include a complete process description and does not specify the precise order in which activities must be executed; it just defines their dependencies on work-products.

XML files are used to store models and descriptions. They are compliant with the W3C XML schema specific for AUTOSAR models: the AUTOSAR XML Schema. That is why every AUTOSAR XML file is characterized by the ".arxml" extension.

2.5.1. System and ECU configuration

In "System Configuration Input" phase, the overall system constraints are identified and software components and hardware are selected. Information exchange format are used as formal description and their structure is different depending on the specific data type:

- Software Components require a software API description.
- ECU Resources require definitions like the processor unit, memory, peripherals, sensors and actuators.
- System Constraints require information about bus signals, topology and mapping of connected software components.

"Configure System" is an activity that contains a collection of complex algorithms and engineering work. System configuration tools can support the mapping operation of software components to ECUs. This configuration must obviously satisfy the restrictions specified in the System Configuration Input, matching resources and timing requirements. As output of this activity, we find the System Configuration Description, which includes information about system and mapping of software components to ECUs. Next steps need to be performed for each system ECU.

"Extract ECU-Specific Information" extracts information of a specific ECU from the System Configuration Description and automatically generates the ECU Extract of system Configuration.

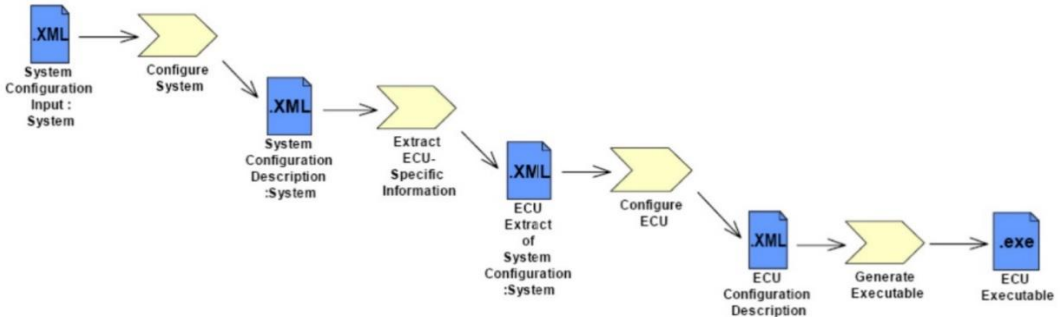

Figure 15: System and ECU configuration

"Configure ECU" is a phase that mainly deals with RTE and BSW configuration. It includes information that is strictly related to the implementation, e.g. task scheduling, required Basic Software modules, configuration of the Basic Software, assignment of runnable entities to tasks… At the end of this activity, we find an ECU Configuration Description containing ECU specific information that can be exploited to build the runnable software.

The ECU configuration step must not be underestimated; it requires engineering competences that are not needed, for example, in an information extraction phase. In this activity, indeed, detailed scheduling information or the configuration data for e.g. the communication module, the operating system, or AUTOSAR services have to be defined.

The last step is the "Build Executable" one, in which, starting from the ECU Configuration Description, code is usually generated, compiled and linked in an executable file.

2.5.2. Application Software Components configuration

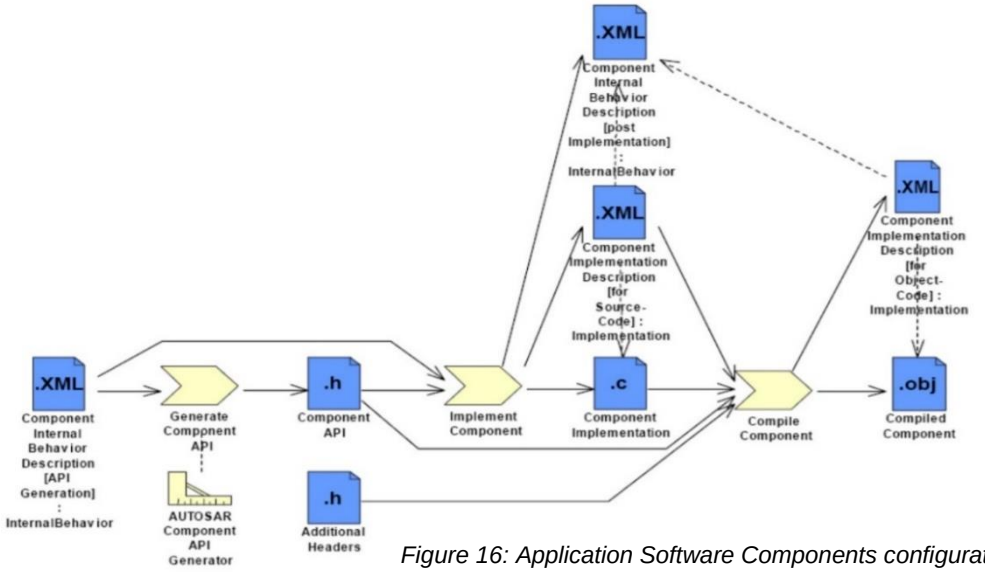

Figure 16: Application Software Components configuration

Configuration flow of Application Software Components is parallel to previous steps.

First, we find Component Internal Behaviour Description, which illustrates how a component responds to events like received data elements and describes the scheduling relevant aspects of a component.

After that, AUTOSAR Component API Generator reads the provided component description and creates a Component API containing all header declarations for the RTE communication.

In the “Implement Component” phase, the developer can implement the component independently from the external system design. As result, we obtain the Component Implementation (typically “.c” files), the Component Internal Behavior Description (more descriptive than the one generated at the beginning) and the Component implementation Description (to collect information regarding next build process).

At the end of this process, Compile Component generates Compiled Component using Component Implementation Description, Component API and Additional Headers. In addition, a new refined Component Implementation Description comes out, containing last process information, like linker settings.

2.6. MICROSAR

2.6.1. What is MICROSAR

As a promoting member of the AUTOSAR consortium, Vector Informatik is able to offer a wide range of design and development tools, as well as basic software modules specific to the AUTOSAR ECUs. Vector products for the development, distribution, generation and configuration of AUTOSAR software can be integrated with the DaVinci Tool Suite. Moreover, they help engineers to design distributed systems and software components compliant with AUTOSAR technology and to shorten the development time of automotive networks.

MICROSAR ([9]) is the Vector implementation of an embedded software for AUTOSAR ECUs that covers the standard and contains many useful extensions. It consists of the runtime environment MICROSAR RTE and MICROSAR basic software modules (BSW). Each BSW module is assigned to a MICROSAR package. Vector combines

and releases the BSW modules needed in individual “software integration packages” (SIP).

2.6.2. MICROSAR Architecture

The BSW modules of the MICROSAR packages assure basic functionality of the ECU. They contain implementations of AUTOSAR standard services needed for functional software that can be developed independently, because the AUTOSAR architecture follows a consistent strategy of hardware abstraction.

MICROSAR.OS and MICROSAR.MCAL packages contain hardware-dependent modules that Vector release for a large number of different hardware platforms and compilers. The operating system MICROSAR.OS is available for single core and multi core-processors. Based on its ongoing contacts with OEMs, Vector is able to offer a number of OEM-specific BSW modules and extensions such as the diagnostic modules.

To produces a complete set of ECU software, functional software can be integrated after the generation of preconfigured MICROSAR BSW modules that satisfies project’s requirements.

If the functional software consists of AUTOSAR-conformant SWCs, a run-time environment (RTE) is needed. The MICROSAR.RTE implements communication between the SWCs and their access to data and services from the BSW modules. Along with managing the entire flow of events and information, the MICROSAR.RTE also assures consistency in the exchange of information and coordinates accesses across core or memory protection boundaries.

ECU projects without SWC architecture (and therefore also without Rte) are optionally supported by the Vector vBre (Vector Basic Runtime Environment).

2.6.2.1. MICROSAR.OS

MICROSAR.OS is a pre-emptive real-time multitasking operating system with optimized properties for microcontrollers. It is based on AUTOSAR OS specification, as extension of the OSEK/VDX-OS standard, including functions for time monitoring and memory protection.

Memory Protection Unit (MPU) protects the OS partitions, which can run without the risk of mutual interference due to incorrect data changes, so that the system can operate in parallel partitions with different ASILs. LeanHypervisor is a module to ensure a safe startup of multiple operating system partitions in a multicore processor or SoC. It is compliant with ISO26262 ASIL D standard and it is in charge of programming the system MPU during system startup and then starting the operating system partitions.

2.6.2.2. MICROSAR.SIP

The Software Integration Package (SIP) is a fundamental component of every MICROSAR delivery that can be a prototype, beta, update or production one. Vector lists its customer requirements in advance of delivery and then it develop the SIP as individually as possible, also testing it. This allows companies like Magneti Marelli to put the entire package into operation within just few days. MICROSAR package is implemented so that it can cover as many additional variants to the initial configuration as possible. However, Vector tries to strictly satisfy project-specific constraints in order to ease the product integration for the customer. The aim is running the delivery on as many devices as possible from the preselected processor line. If it is technically possible for the project, MICROSAR SIP includes as the Extension "Start Application". It is based on the ECU specific input data for communication and diagnostics.

2.6.3. DaVinci Tool Suite

As already mentioned, DaVinci tools are useful to configure BSW modules in a user-friendly and well-coordinated way, instead of handwriting them. Moreover, multiple users can simultaneously work on a project thanks to the Multi User Support. DaVinci tools require an “ECU Extract of System Description” file as input and then assist the user in configuring the RTE and the BSW modules.

Automatic code generation relieves the programmer of tasks that recur frequently and are prone to errors when performed manually. This of course allows time and costs savings.

In particular, DaVinci Developer is a tool for designing the architecture of SW-Cs, including ports, data types, connectors and internal behavior. It ease the engineering

process thanks to graphical or textual grid views and to the automatic verification of AUTOSAR compliance of the project.

This tool can work in combination with DaVinci Configurator Pro, that offers a customized user interface to configure, validate and generate BSW and RTE.

Considering the entire system configuration DaVinci Developer comes into play when the XML of the “ECU extract of System Configuration” is already provided. In this phase SW-Cs are manged and saved into XML files that are part of the “ECU extract of System Configuration” and the “Component Internal Behaviour Description”. After that, DaVinci Configurator can be used to read the produced XMLs and configure the ECU generating the BSW, RTE and the “ECU Configuration Description”, but also to generate “Component API” (“.h” files) and component templates (“.c” files) that will be later implemented.

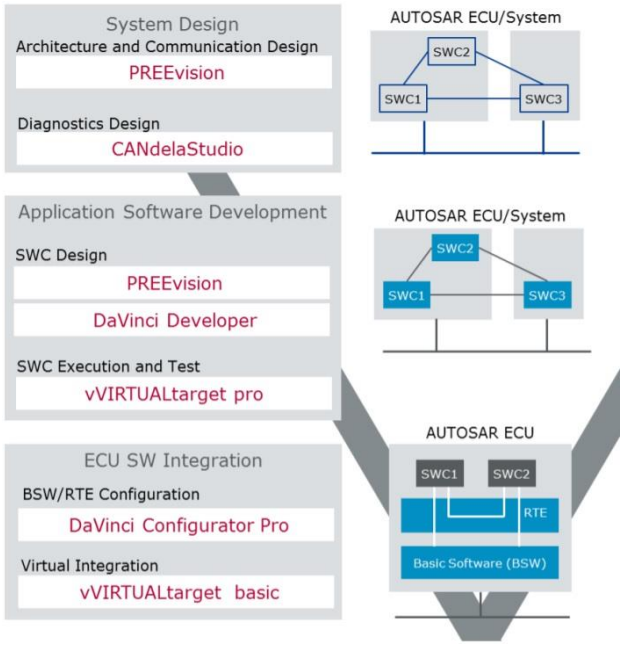

Figure 17: DaVinci Tools

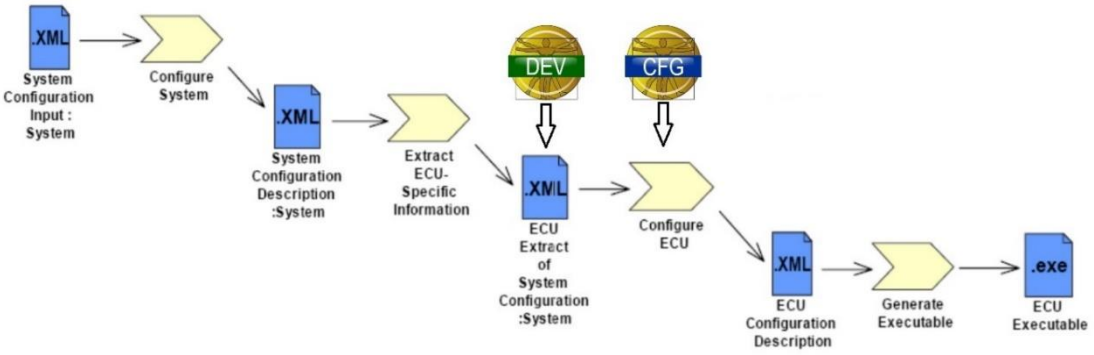

Figure 18: DaVinci roles in System and ECU configuration

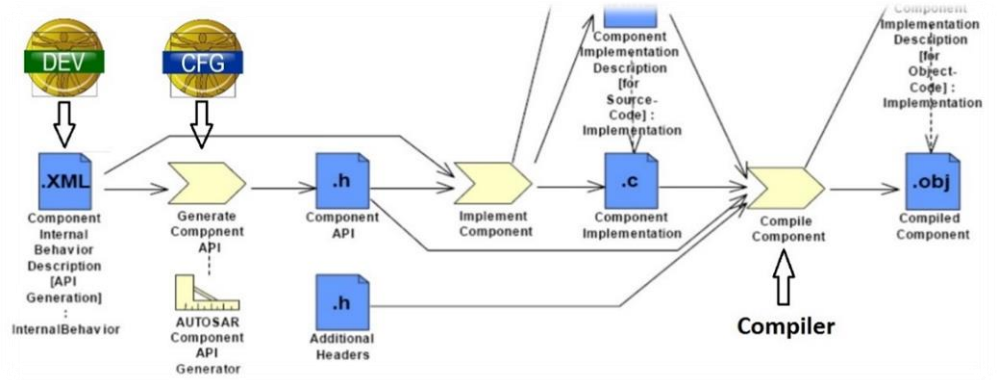

Figure 19: DaVinci roles in Application Software Components configuration

3. Inter-Core communication case of study

3.1. Inter-OsApplication-Communication

As already mentioned, AUTOSAR supports multi-core systems since 4.0 version. This means that it allows different OS applications to be statically allocated to the different cores and supports data exchange among these cores by means of the IOC sub-module.

The MICROSAR implementation of the IOC follows the AUTOSAR recommendations, using Spinlocks wrapped by a suspend-all-interrupt function. Spinlock is a programming technique that provides a lock variable acquisition and performs a busy-wait routine until the lock is released, allowing to enter the critical section. Therefore, if a core acquires a lock, the others cannot proceed with their execution. It is easily understandable that such an implementation can be very inefficient if the communication between cores is repeated very often or if the size of the transferred data is considerable. Another limitation of the IOC is that it does not allow to consistently communicate data elements produced by different SW-Cs.

Considering that these negative aspects of the already existing inter-core communication could be solved in its own control units, Magneti Marelli has decided to study a new solution, based on its specific use cases.

As alternative to the IOC, a Cyclical Asynchronous Buffers approach has been chosen, based on the older NON-AUTOSAR multi-core architecture already used by the company.

3.2. Cyclical Asynchronous Buffers

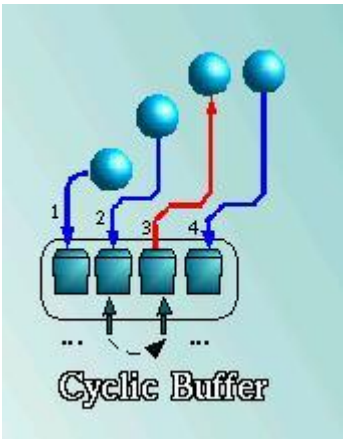

Figure 20: Cyclic Buffer

As defined in [1] and [12], the Cyclical Asynchronous Buffer (CAB) is a One-to-Many (in general Many-to-Many) Asynchronous Communication System purposely designed for the cooperation among periodic activities with different activation rates: sensory acquisition, control loops, etc.

CAB Mechanism guarantee that the last/newest message (data), after the first write operation, is available at any instant for reading. The message is not consumed by the reader, but is maintained into CAB System until a new message is overwritten. As a consequence, the same data is read more than once if the receiver is faster than the sender and, of course, messages are lost if the sender is faster than the receiver. However, this eliminates unpredictable delays due to synchronization and allows a continuous fetch of fresh data that is a satisfactory condition for many applications.

A CAB is created with a specific name and the dimension, parameter that corresponds to the maximum number of messages contained in the CAB (max_buff), multiplied by their single dimension (dim_buff). In its structure we also find a pointer to the list of free buffers and one to the most recent buffer (mrb). A buffer is composed by three fields: the pointer to the next free buffer (next), a counter that memorizes how many tasks are accessing that task (use) and the stored message (data).

CAB messages are always accessed through a pointer to a message buffer that first must be reserved, then filled with the data content and finally made available to be read.

CAB message write is performed with following paradigm:

```
buff_ptr = reserve(cab_id);

<write message to *buff_ptr>

putmess(cab_id, buff_ptr);
```

CAB message read is very similar: a task gets the pointer to the most recent message, use it and release the pointer. It is performed in the following way:

```
buff_ptr = getmess(cab_id);

<read message from *buff_ptr>

unget(cab_id, buff_ptr);
```

Can be noticed that simultaneous read and write operations are allowed without critical sections because of multiple memory buffers managed via Cyclic Array of Buffer Pointers. If a task reserves a buffer to write in a CAB and also another task wants to write inside it, the last one must use a free buffer that is different from the ones already reserved by who is writing and who is eventually reading. That is why, to avoid blocking, the number of buffers inside a CAB must be at least equal to the number of tasks that use it, plus one (num_tasks+1).

3.3. Proposed solutions

In the old Magneti Marelli NON-AUTOSAR architecture, a classical Cyclical Asynchronous Buffers approach has been used in order to meet time constraints and maintain higher priority of safety critical tasks, without losing data consistency. By the way, in order to improve the speed of next generation AUTOSAR based ECUs, we decided to develop a better-fitted CAB implementation. Thus, we started analysing what is the role of each core and how it communicates with the others in a generic application workflow, then we made assumptions that allowed us to reduce system interruptions and therefore achieve better performance.

3.3.1. Multi-core utilization analysis

Magneti Marelli multi-core ECUs are based on Infineon AURIX TriCore 32-bit MCUs. In the company architecture, two of the three cores are dedicated to hard real-time tasks and one to safety tasks. Divided functionalities allow an easier management of tasks and shared resources between cores. However, this is disadvantageous from a dynamic load-balancing point of view. In fact, a balanced processor load distributes workload evenly on each core, so that the system can be better optimized to perform its set of operations.

Even if this can be a reasonable approach, Magneti Marelli has preferred a task mapping solution that is coherent for each core, allowing a better scheduling tracking and therefore more consciousness on system behaviour. Tasks with the same

functionality are executed on the same core, implying that real-time requirements are often independently satisfied by each core, not by its coordinated use with the others.

This separation of roles between cores involves that their communication is not as frequent as in balanced load solutions, where very dependent tasks running on different cores need to exchange data very often to proceed their execution.

In Magneti Marelli applications, tasks are linked by a very simple relation: one task (in one core) produces data and one or more tasks use it (one task per core); so, the CAB rule becomes: one plus the number of reading cores plus one (1+num_readingCores+1). This means that a shared variable can just be written by one core, but read by all the others, acting as a one-way communication channel between them. We will see that this consideration becomes very important when tasks are running on different cores.

What must be also pointed out is that, for Magneti Marelli use cases, readers do not need to know all the history of written data, therefore a data can be overwritten even if it has never been read by anyone. However, the consistent reading of messages is the critical aspect of the inter-core communication; in fact, reading tasks must always find consistent data available and this implies that they cannot read the same memory area that the writer is updating. This can be relevant for a CAB implementation, where the writer is cycling buffers and, if readers are not fast enough to fetch data, it must wait until everyone has read the buffer with the oldest value.

3.3.2. Scheduling schemes

In order to become more conscious on how to perform a new CAB design, the analysis proceeded simulating the behaviour of the system with scheduling schemes. To do that, we have considered a periodic task per core (three in total): one that writes a CAB (t_a) and the others that read it (t_b and t_c).

In a first approach, we set t_a, t_b and t_c, respectively with a high, medium and low speed. As result, we saw how the CAB theory always guarantee (thanks to the “max_buff=num_tasks+1” formula) at least an available buffer to write and, as expected, reading tasks lose some data history (t_c more than t_b), because of their slowness.

After that, we tried to exchange the speed of t_a and t_b. Being t_a still faster than t_c, this one continues to lose some data history at a certain point in time, but t_b, being faster than t_a, can always read every value t_a release into the buffers.

In the last scheme instead, we considered two tasks per core, so three that are dedicated to inter-core communication (the same as before) and three that preempt them with a certain periodicity. From this model, we understood that tasks belongings to the same core (in particular if they are preemptive) have a relevant impact on communication timings, even if they do not participate to the data exchange.

3.3.3. Two different designs

As already said each core has:

- a different functionality
- a unidirectional communication with the others

These assumptions allow us to simplify the old implementation, but, depending on other additional considerations, we have developed two different design models.

3.3.3.1. First design model

We assume the case in which the read operation is much slower than the write one. This situation can be due to higher priority interrupts that can block the reading task for a long period, or caused by the read access time of the specific MCU.

Therefore, if “max_buff-1” (look at Cyclical Asynchronous Buffers chapter) buffers are occupied by reading tasks and the writer has written in the only available buffer, we can still have tasks that, despite the “num_tasks+1” formula, still need the resources. In this case, the writing task must cycle the CAB until it finds an available buffer. To keep track of buffer availability, we have used a shared array between cores (accessed through Spinlock) that collect the number of readers for each buffer.

Newer solution, as will be shown below, is very simple in “CAB Write” (function used by the writing task), but not in “CAB Read” (function used by the reading task), that is still lighter than the older implementation, having smaller critical sections.

CAB Write

OLD implementation:

<SUSPEND INTERRUPTS>
<GET SPINLOCK>
Free buffer reserved
Buffer pointer obtained
<RELEASE SPINLOCK>
<RESUME INTERRUPTS>

Write message into buffer

<SUSPEND INTERRUPTS>
<GET SPINLOCK>
Buffer becomes available (mrb updated)
< RELEASE SPINLOCK>
< RESUME INTERRUPTS>

First implementation:

Find a free buffer

Write message into buffer

Buffer becomes available (mrb updated)

Implementation differences:

1) The old version reserves a buffer because accepts multiple writers on the same CAB, the new solution accepts just one writer and multiple readers (so, no reservation needed)
2) The old version manages pointers inside critical sections, the new solution has no critical sections
3) The old version uses a stack (not cyclic) of buffers that grows up or goes down, the new solution has cyclic buffers

CAB Read

OLD implementation:

<SUSPEND INTERRUPTS>
<GET SPINLOCK>
Added a user to the most recent buffer
Buffer pointer (mrb) obtained
< RELEASE SPINLOCK>
< RESUME INTERRUPTS>

Read message from buffer

<SUSPEND INTERRUPTS>
<GET SPINLOCK>
Removed a user to the most recent buffer
< RELEASE SPINLOCK>
< RESUME INTERRUPTS>

First implementation:

< ENTER CORE CRITICAL SECTION>

Retrieved mrb value

<GET SPINLOCK>
Added a user to the most recent buffer
< RELEASE SPINLOCK>

Read message from buffer

<GET SPINLOCK>
Removed a user to the most recent buffer
< RELEASE SPINLOCK>

< EXIT CORE CRITICAL SECTION>

Implementation differences:

1) The old version adds user to the buffer and obtains the pointer in the same critical section. In new solution, we have changed the order of the operations, so that we have a spinlock just for the user addition. (Similarly happens for the user removal)
2) The old version suspends and not disables interrupts.
3) The old version has two “SUSPEND INTERRUPTS” and two nested “GET SPINLOCK”. We have instead created a big (configurable) “CORE CRITICAL SECTION” block and two small “GET SPINLOCK” ones.

Alternative CAB Read (not chosen)

OLD implementation:

<SUSPEND INTERRUPTS>
<GET SPINLOCK>
Added a user to the most recent buffer
Buffer pointer (mrb) obtained
< RELEASE SPINLOCK>
< RESUME INTERRUPTS>

Read message from buffer

<SUSPEND INTERRUPTS>
<GET SPINLOCK>
Removed a user to the most recent buffer
< RELEASE SPINLOCK>
< RESUME INTERRUPTS>

First implementation:

< ENTER CORE CRITICAL SECTION>

Retrieved mrb value

<GET SPINLOCK>
Added a user to the most recent buffer
< RELEASE SPINLOCK>

< EXIT CORE CRITICAL SECTION>

Read message from buffer

< ENTER CORE CRITICAL SECTION>

<GET SPINLOCK>
Removed a user to the most recent buffer
< RELEASE SPINLOCK>

< EXIT CORE CRITICAL SECTION>

Implementation differences:

1) The alternative to our proposed solution has two smaller “CORE CRITICAL SECTION” instead of a larger one.
2) In both the old and the new implementation, the reader can be blocked during the message reading by the same core tasks. This can delay the operation but cannot change the result.

We chose the previous “CAB Read” because we prefer to prevent the reading interruption of the message, obtaining a more deterministic data transfer.

3.3.3.2. First design optimization

In this first design we can point out that, if a CAB has just one reader, it is also the only one that can write the most recent buffer, so Spinlocks are not needed for that CAB. Spinlocks are often limited resources and, moreover, they are also a limiting factor for the cores’ execution. That is why, removing them, we can obtain an optimized version of this first design.

3.3.3.3. Second design model

We assume the case in which the read and write operations have almost the same execution time. This allows us to assume that a writer will always have an available buffer to write (cyclically the next one).

Therefore, we don't need to count readers of each buffer (no shared array, no Spinlocks) and, moreover, in write operation the free buffer must not be found cycling among all buffers (as in first alternative), because it is for sure the next one.

In this solution, we are never using Spinlocks, neither to write nor to read. However, writer must disable interrupts, something that is useless in the first alternative.

CAB Write

OLD implementation:	**Second implementation:**
<SUSPEND INTERRUPTS> <GET SPINLOCK> Free buffer reserved Buffer pointer obtained <RELEASE SPINLOCK> <RESUME INTERRUPTS>	<DISABLE INTERRUPTS>
	Pointer to next buffer
Write message into buffer	Write message into buffer
	Buffer becomes available (mrb updated)
<SUSPEND INTERRUPTS> <GET SPINLOCK> Buffer becomes available (mrb updated) < RELEASE SPINLOCK> < RESUME INTERRUPTS>	<DISABLE INTERRUPTS>

Implementation differences:

1) The old version reserves a buffer because accepts multiple writers on the same CAB, the new solution accepts just one writer and multiple readers (so no reservation needed)
2) The old version manages pointers inside critical sections; the new solution has just the interrupt disabling.
3) The old version seems to use a stack (not cyclic) of buffers that grows up or goes down, the new solution has cyclic buffers

CAB Read

OLD implementation:

<SUSPEND INTERRUPTS>
<GET SPINLOCK>
Added a user to the most recent buffer
Buffer pointer (mrb) obtained
< RELEASE SPINLOCK>
< RESUME INTERRUPTS>

Read message from buffer

<SUSPEND INTERRUPTS>
<GET SPINLOCK>
Removed a user to the most recent buffer
< RELEASE SPINLOCK>
< RESUME INTERRUPTS>

Second implementation:

<DISABLE INTERRUPTS>

Retrieved mrb value

Read message from buffer

<ENABLE INTERRUPTS>

Implementation differences:

1) The old version adds user to the buffer and obtains the pointer in the same critical section. In new solution, we just need to enter the “CORE CRITICAL SECTION” and get the pointer.
2) The old version has two “SUSPEND INTERRUPTS” and two nested “GET SPINLOCK”. We have just created a big (configurable) “CORE CRITICAL SECTION” block.

3.3.4. Designs comparison

As we can notice, both our alternative designs have smaller critical sections than the ones of the older implementation, without including a larger execution time and memory occupation. A core that waits for a Spinlock is blocked until its acquisition; this is the reason why we have reduced as much as possible this kind of critical section. The interrupt disabling instead, has not such a strong impact on the MCU performance, because it just blocks the execution of higher priority tasks on the same core, not affecting the other tasks running in the other cores.

3.4. Methodology

The development process we chose for our project is the V-Model, that allowed us to flow step by step from high level design to development phase and then to test everything going backwards. We needed two different “V” in order to first study the correct solution and its related embedded code, and then to develop the code generator.

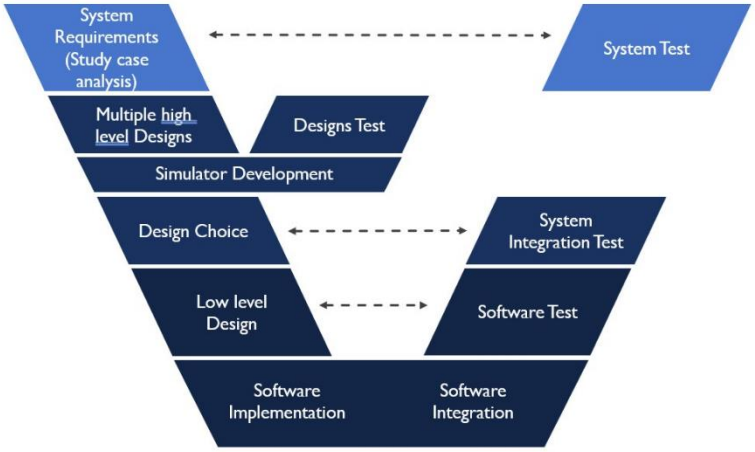

Figure 21: First V-process

Once defined the two CAB designs, we decided to test them to verify their correctness before implementing the real code for ECU deployment. Hence, we developed a software simulator in C language for Windows OS that also helped us in the choice of the design. After that, we introduced a low-level design that was a pseudo code very similar to the C code. The software implementation instead, is comprehensive of a SW-Cs Architecture design and a BSW Configuration, necessary steps to set a working

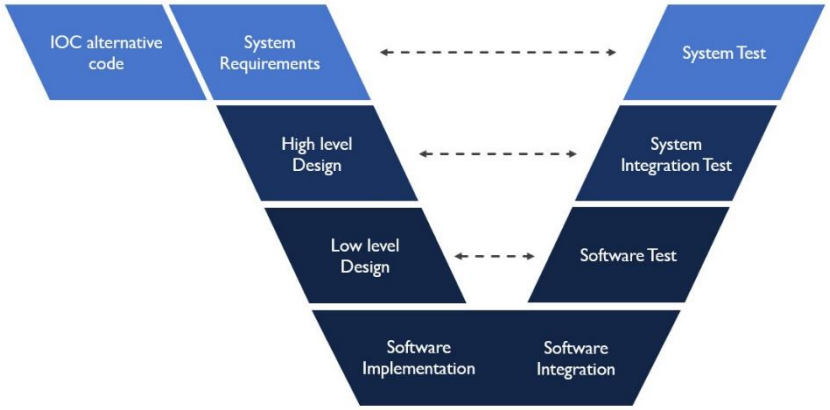

Figure 22: Second V-process

MICROSAR architecture and related C code templates that can be implementable.

As system requirements for code generator development, we consider the C code that we developed and tested in the previous process. From them we extracted the high and low level design of the generator: the analysis class diagram and the design class diagram. This time software implementation was just Java development.

3.5. Simulator

To build a simulated environment that could be as similar as possible to the original one, we decided to exploit threads provided by Windows, so that they could act as the three cores of the AURIX MCU. Therefore, we have generated three threads from the same process and we have imposed them the same priority. This allows Windows to schedule threads with a Round Robin algorithm, meaning that they are fairly executed. For simplicity, each thread represents also the unique task of the core, which can communicate with the others through the shared memory provided by their common process.

We defined a data type as a “struct” of two atomic data, to verify that the model could perfectly ensure data consistency. As relation between them, we have imposed to have two numbers: one the opposite of the other. “thread 0”, in fact, generates a random number, creates its opposite and writes them into the right buffer of the CAB. After that, “thread 1” and “thread 2” can verify data consistency simply summing them after the reading operation.

Simply implementing CAB algorithm into threads, we let them work at full speed (managed by the OS) and we cannot have a realistic emulation of the task behaviour. What really matters for our purpose is having tasks with different relative execution times, this means that we just want to control the speed of a thread with respect to the others, we do not care which can be the real execution time of each thread. Therefore, to impose this task characteristic we have inserted some “sleep” functions in strategic points of the code. In this way, we are stopping the thread for a certain period, simulating its execution, because, if we do not consider the processor workload, a sleeping thread can be seen as a running one, from a timing point of view. Where we extend the task execution is fundamental for our purpose, because it affects the way tasks interact. A “sleep” between two atomic writing operations simulates a longer writing operation; the same happens from the reading point of view. By the way, to

enlarge the task timing without changing the communication one, we have also paused threads before the end of their cycle.

Design 2 writing thread example:

```
Writing _Thread(){
        int i = 0;
        for(i=0; i<WRITER_ITERATIONS; i++){
                Find a free buffer
                Write "data" into buffer
                Sleep(WRITING_DURATION)
                Write "-data" into buffer
                Buffer becomes available (mrb updated)
                Sleep(ENDING_DURATION)
        }
}
```

Fast readers could access CAB before a first writing operation, for this reason we also needed to ensure a correct initialization of threads. To work around the problem we impose the MRB initialization value to “-1”, so that, if readers find a negative MRB, they skip the reading operation. As soon as the CAB is written, the writer updates the MRB to “0” and data is considered available.

In this simulated environment, we have not introduced the “core critical section” because what we have developed is a simplified model of the system in which there is just a task per core that cannot be pre-empted by anyone else.

The Spinlock present in “Design 1” has been implemented with Windows Mutex, that provides the exclusive access to the critical sections, as Spinlocks do in AUTOSAR architectures.

3.6. Simulation Results

Tests have been performed changing tasks' timings, in order to emulate different use cases. These different configurations have been obtained simply modifying sleeping parameters of threads in both design simulations, in order to compare their impact on the two models.

First, we noticed that the CAB theory was respected; in fact, the writer correctly cycles buffers and releases the updated MRB, while readers are always reading the buffer indexed by the current MRB.

As expected, enlarging the writing time, the number of readings performed on the same buffer is increased. However, this behaviour can be limited slowing down readers with a highest "ENDING_DURATION" or "READING_DURATION" (input parameters of the Sleep() function). Of course, if instead we reduce the "WRITING_DURATION" of the writer, the speed of released MRB becomes higher, the probability that a reader reads the same buffer is reduced. If the writer is fast enough, with a speed that is similar or greater than the one of readers, some buffers' readings are skipped.

Between the two designs, we expected that the first one, using Spinlocks, was the heavier. Indeed, the simulator demonstrates that Spinlocks have a considerable impact on the reader performance, even if it is blocked for a small critical section. To reach this outcome we have imposed the writing time a bit lower than the reading one, in order to see how fast readers can follow the MRB updates.

In this pseudo-real environment, the presence of critical sections that slow down the communication can be noticed. By the way, to actually understand their importance, we need to keep in mind that the target of this implementation is a multi-core embedded system with many tasks that exchange information through the shared memory.

4. Embedded software development

4.1. Inter Core Communicator

The right CAB design model has been chosen keeping in mind that reliability is a milestone in the industrial field, especially in the automotive context. In general, when a new software component is deployed, it must not compromise the execution of the others and, of course, it must do its own work. Speed or memory optimizations can be considered as a plus, not as a strict requirement that overcomes the safety. Therefore, our design had to optimize as much as possible the inter-core communication, being at the same time the more robust solution.

We considered that the way MCUs manage the access to the RAM is not deterministic, even if we disable all core interrupts.

After this assumption, we can notice that, among our two designs, just one of them can be considered as robust as the old solution, being at the same time more efficient: the first design. So, our choice fell on this model that, as already mentioned in the CAB design analysis chapter, provides that readers write a shared variable to increase the counter that keep track of buffer users. To do that, Spinlocks must be used, although for a short critical section with respect to the one of the old implementation.

The chosen test ECU to develop the new software is an ICC3 AUTOSAR compliant multi-core ECU that Magneti Marelli has prototyped. The principal BSW supplier of the company is Vector, which provides its own implementation of the AUTOSAR architecture: MICROSAR. Differently from the IOC present in MICROSAR.OS, that allows the communication between OS Applications, we defined our new software as Inter Core Communicator (ICC). Indeed, its aim is managing the physical communication between cores, not between different OS Applications that can be part of the same one. The ICC is located in the Complex Driver layer and it can be used as alternative to the IOC inside MICROSAR.OS, so that a system can be designed to let them work together, having some shared variables accessed through IOC and others through ICC.

4.2. SW-Cs Architecture Design

An ICC is a SW-C that is directly linked with other SW-Cs of the same core that require an inter-core communication, but also with other ICCs, thanks to their internal implementation. Therefore, to start the implementation of our CAB design, we first needed a pre-defined testing SW-Cs architecture that included ECU components. For MICROSAR architectures, components can be defined in DaVinci Developer modifying the “ECU extract of System Configuration” of the MICROSAR.SIP.

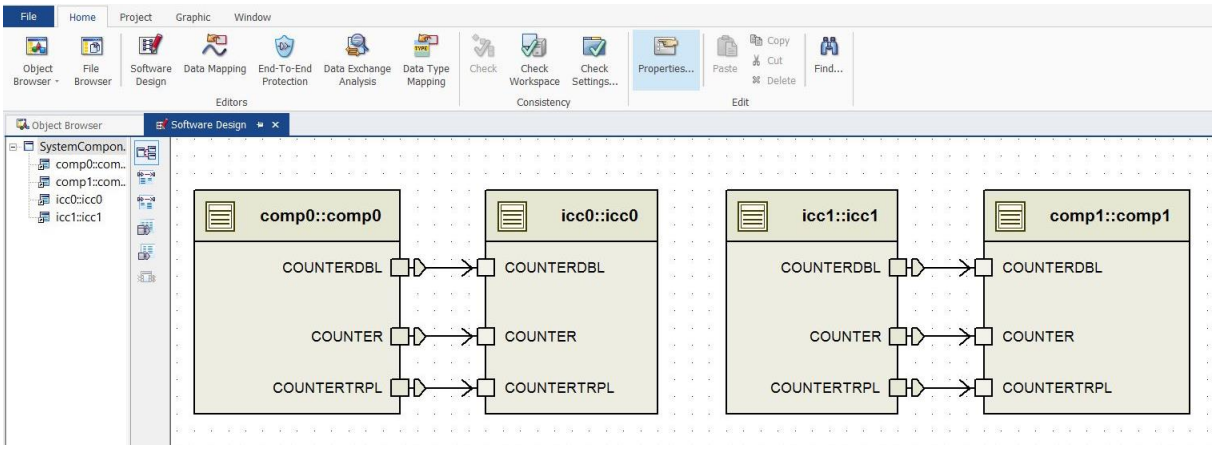

Figure 23: SW-Cs Architecture Design

We have created two “Application Components” (SW-Cs): “comp0” that writes data and “comp1” that reads it. They rely in two different cores and are connected to their relative ICC (named with a number corresponding to the core number). As we can notice in figures, every SW-C situated in one core is directly linked with the others situated on the same core, this means that they can communicate through connected “Application Ports” that must be of the same type. These connections will be later used by the DaVinci Configurator to automatically generate the RTE that implements connection at code level. The subdivision into cores allows to get rid of the additional contribution of the IOC, which is substituted with the interconnected ICCs. Their communication cannot be seen at components architecture level, because our internal implementation will actually be in charge of exchanging their data.

Inside components, “Application Ports” are associated to “Access Points” of Runnables in order to express what data they can access. “Application Ports” that we used are just the ones that in DaVinci are defined as Sender/Receiver, that allows components to provide or receive information. However, before we could use Application Ports we have defined Application Port Interfaces based on standard Magneti Marelli Application

Data Types. Each of them is mapped to the corresponding code variable by the DTMS (Data Type Mapping Set), information exploited by the DaVinci Configurator to generate code. This chain of dependencies is fundamental to make the SWC-Cs architecture modular and code independent.

As already mentioned, DaVinci Configurator Pro generates the RTE or, if configured, the IOC, basing on the port connections defined in DaVinci Developer. Thus, if two correlated data are sent independently through different ports, they are seen by the Configurator as uncorrelated and it cannot guarantee a consistent data transfer. To send correlated data, we should use Application Ports based on Application Port Interfaces that include in their definition every correlated Data Type.

With our ICC implementation instead, this operation can be avoided. Indeed, Runnables are aware of what must be kept consistent, because we designed them to consider as single data their complete set of Access Points, without grouping their related Data Types inside new Application Port Interfaces. To better understand this concept, let us analyse how we conceived the SW-Cs of this first architecture.

Inside comp0, we have collocated two Runnables: one that writes two atomic values, COUNTER and COUNTERDBL, and the other that writes COUNTERTRPL. In comp1, we find other two Runnables: one with the role of reading COUNTER and COUNTERDBL, while the other COUNTER and COUNTERTRPL. We wanted that our ICCs could always guarantee a consistent reading of such data couples. To do that, we inserted in each ICC a Runnable with Access Points COUNTER and COUNTERDBL and another one with COUNTER and COUNTERTRPL (in this case, icc0 Runnables just send data, while icc1 receives it). The consistency check will be defined in “Templates Implementation” phase, imposing COUNTERDBL as the doubled value of COUNTER and COUNTERTRPL as its tripled value.

4.3. BSW Configuration

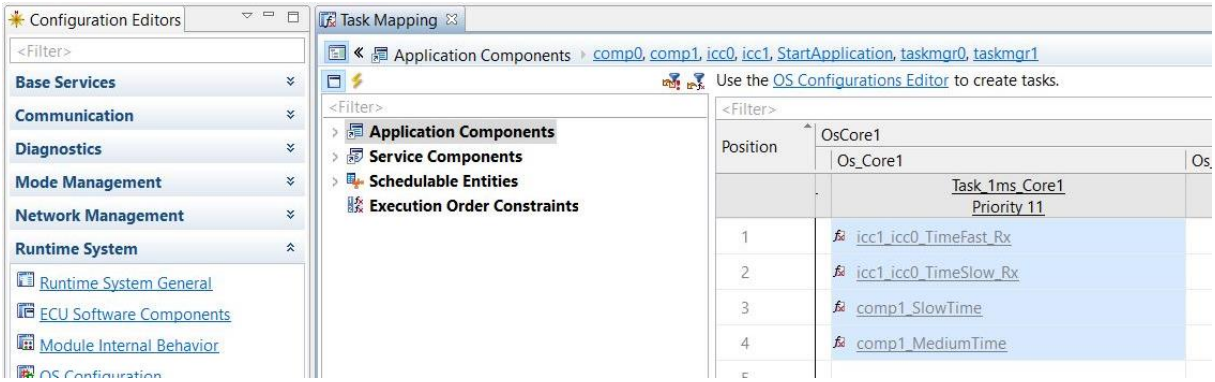

Figure 24: Task Mapping - Reading Runnables

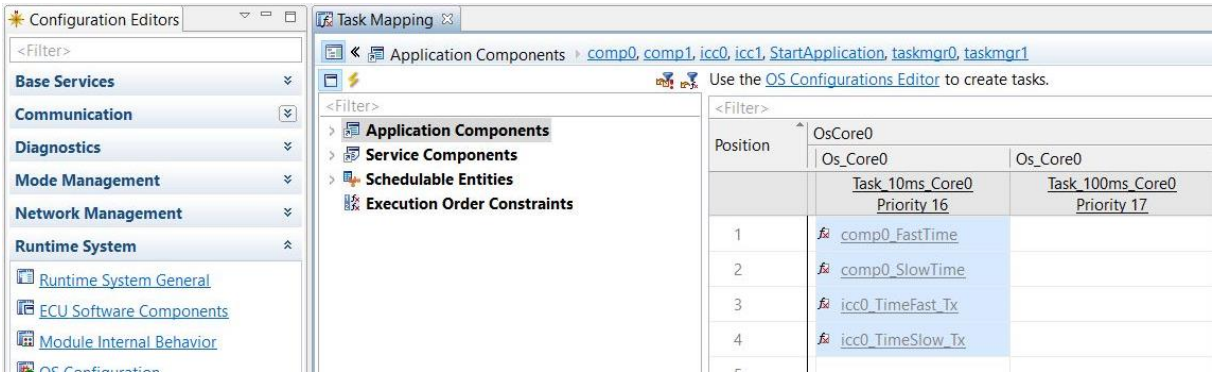

Figure 25: Task Mapping - Writing Runnables

Once saved the project in the DaVinci Developer, we can open it with the Configurator Pro. This tool reads the ARXML files of the previously modified “ECU extract of System Configuration” and warns the user that latest changes have to be configured.

In the “OS Configuration” section, we have created new tasks for OS Applications of “Core 0” and “Core 1”. After that, we entered the “Task Mapping” section and we mapped every Runnable we added in the SW-Cs Architecture with the tasks previously inserted in the OS Applications. Each transmitting ICC Runnable has to be mapped at the end of the task where we decide to make it run, each receiving Runnable instead, at the end of the task. In this way, we guarantee that every task cycle ends updating every modified variable or that starts with updated values, exploiting ICC functionalities.

To make our testing phase easier, comp0 and comp1 Runnables are in the same tasks of the ICC Runnables; in fact, we wanted them to have the same timings of the ICCs,

so that, at every task cycle, ICC0 can read the consistent data produced by comp0, while comp1 can read the consistent data transferred by ICC1. If comp0 and comp1 had been on different tasks, we would have had consistency errors not related to the ICC problems, but just to the tasks' synchronization.

In "OS Configuration" we also configured a Spinlock to be used in our code. In this case we did not need any Spinlock having just a reader for a unique CAB, but we used one anyway so that our implementation could be as generic as possible. To define a Spinlock, we set its name, the OsApplications that access it and other two parameters: "OsSpinlockLockMethod" and "OsSpinlockLockType". The first one allows to associate the Spinlock to a locking method that prevents the pre-emption of a task that holds it. We set it as "LOCK_NOTHING", because we decided to implement the "CORE CRITICAL SECTION" in our ICC code. "OsSpinlockLockType" instead, let us choose whether to use a standard Spinlock or an optimized one provided by MICROSAR.OS. Setting it as "OPTIMIZED", we decided to minimize the execution time of the Spinlock API, but to do it we had to make some assumptions. In fact, the standard Spinlock performs error checks on OS configuration, verifying that no deadlocks are occurring. In our use cases, Spinlocks are never nested and we cannot have deadlocks between them. The optimized Spinlock omits the API checks, so that its data are kept in user memory and the OS context change is eliminated.

Next step was the validation of the BSW configuration, operation that allows the DaVinci Configurator Pro to check if some generation phases are inconsistent. Ensured the correctness of the project, we generated the BSW, the RTE and, furthermore, the templates of our SW-Cs, so that we could manually implement them.

4.4. Templates Implementation

A template is an auto-generated ".c" file with a fixed structure that is divided in auto-generated sections (where code is completely overwritten, if we generate again the same template in the same folder) and manually implementable sections (where we can write our C code avoiding that it will be deleted or substituted after a further template generation). Of course, Runnables are defined with auto-generated names and Access Points, but being templates, their implementation must be manual.

Therefore, we converted our low-level design structure into C code, filling Runnables internal fields, including header files and defining variables of ICC files. As already mentioned in the “SW-Cs Architecture Design” chapter, comp0 was implemented to produce related variable couples, while comp1 to check that the relation between them is respected, so that data consistency can be verified.

4.5. “cabs” and “cab_shared_sec” files

During templates implementation, we understood that these generated templates were not enough to constitute our complex driver. We missed were to declare and define shared variables accessed by every ICC file. Hence, we imposed that they all must include a “cabs.h” file that contains:

- The macro definitions of “ENTER_CRITICAL_SECTION” and “EXIT_CRITICAL_SECTION” in order to be properly parametrized with the respective calls to the OS for core critical sections (for instance “SuspendAllInterrupts()” and “ResumeAllInterrupts()”).
- The macro definitions of every buffer dimension (following the CAB rule: number of reading tasks “+” number of writing ones “+” 1)
- The new datatypes based on the buffers’ structures.
- shared variables declarations.

We created this file together with the “cabs.c” one that includes it in order to define its variables.

We parametrized the memory location of these shared variables, so that the software integrator can decide which is the best RAM section where to put them. We need to keep in mind that, if they are accessed very often, this choice can have a great impact on the MCU performance.

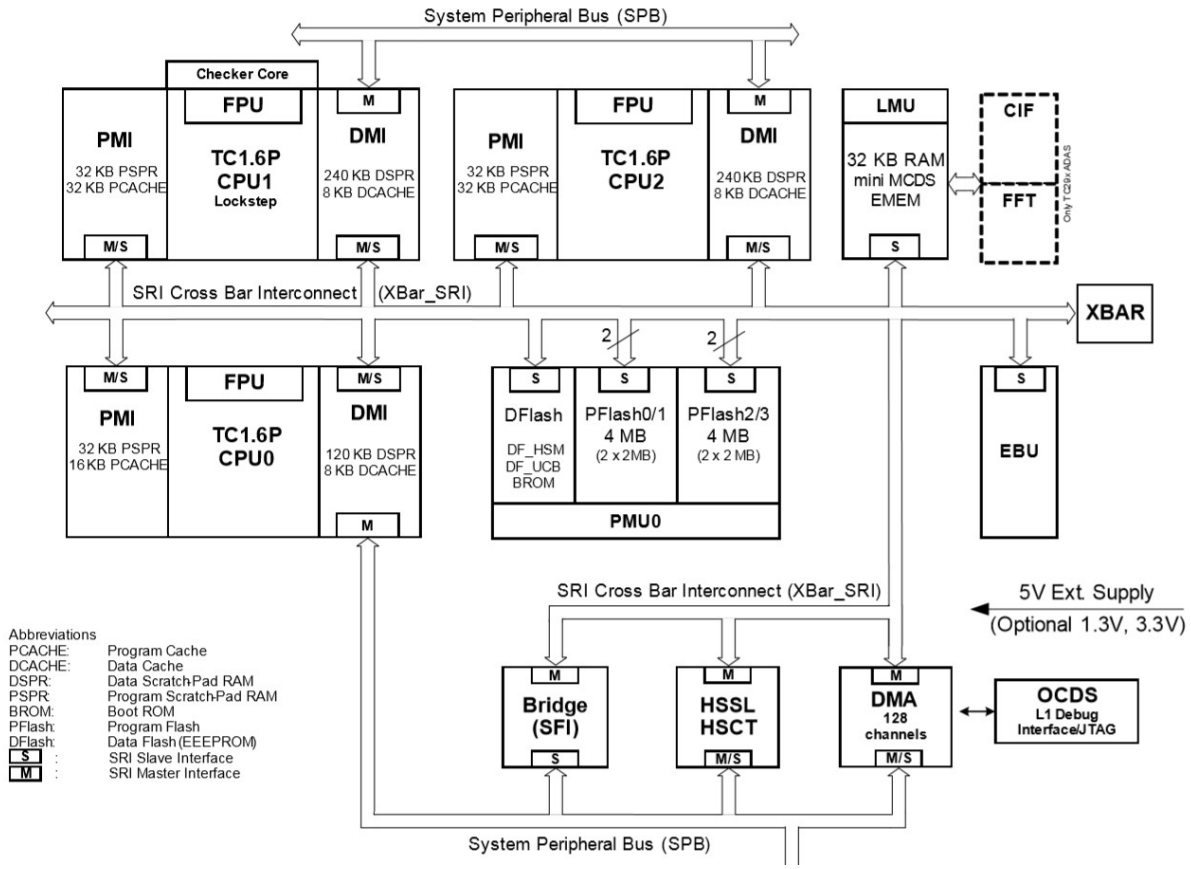

Figure 26: AURIX TC29 MCU - partial schematic

As an example, we can consider the TC29 MCU structure ([16]), where every core has its own RAM divided in PSPR (Program Scratch-Pad RAM) and DSPR (Data Scratch-Pad Ram), but there is also a global/shared one in the LMU (Local Memory Unit). Every RAM is accessible by every core, this is why it can be very important where we put shared variables. For example, if we know that a variable can be only accessed by core0 and core1, we can put it DMI belonging to core0 or core1. This solution is more efficient than the one we have if we store the same variable inside the LMU; this is due to the fact that we are shortening the path to access data, avoiding a useless congestion of the crossbar.

To parametrize the memory location, we surrounded shared variables' definitions of each core in this way:

```
#define CAB_CORE<NUM_CORE>
#include "cab_shared_sec_on.h"

<SHARED_VARIABLES>

#include "cab_shared_sec_off.h"
#undef CAB_CORE<NUM_CORE>
```

Including “cab_shared_sec_on.h” before the definitions we add this piece of code:

```
#ifdef CAB_CORE<NUM_CORE>
   #pragma section "CAB_CORE<NUM_CORE>_section"
#else
   #error No core definition found for pragma section of cabs.h or cabs.c elements
#endif
```

Including “cab_shared_sec_off.h” at the end of the declarations we are adding:

```
#ifdef CAB_CORE<NUM_CORE>
   #pragma section
#else
   #error No core definition found for pragma section of cabs.h or cabs.c elements
#endif
```

In this way we can call a pragma that can be redefined just changing the content of the “cab_shared_sec_on.h” files and leaving “cabs.h” and “cabs.c” untouched.

4.6. Software building

IBM Rational Synergy ([13]) is the tool adopted by the company as task-based software for configuration management that allows the cooperation of distributed development teams. Therefore, every project needs to be versioned with this system, which saves everything in a server accessed by selected users.

Once uploaded our project with this tool, we remotely accessed a UNIX based server that Magneti Marelli uses as compilation platform. From there, we opened Synergy in order to see our project files and we built the software with a predefined “make file”. This can be also considered a first bug correction step, because first compilation errors that came out showed some implementation problems that have been solved with few corrections.

4.7. Testing

After the building process, we obtained many files resulting from compilation and linking. The one that mostly interested us was the “.elf” file. ELF (Executable and Linkable Format, formerly called Extensible Linking Format) ([14]) is a common standard file format for executables, object code, shared libraries, and core dumps. Unlike many proprietary executable file formats, it is very flexible and extensible, and it is not bound to any particular processor or architecture.

To debug ECUs, Magneti Marelli uses emulators provided by Lauterbach (producer of microprocessor development tools) (15). These boards need to be controlled by their proprietary software Trace32, which allows to read ELF files, to flash the contained firmware inside the connected MCU and to have a complete debugging interface.

Therefore, we physically connected the ECU to the emulator and the emulator to the company intranet through an Ethernet connection. In this way, we could launch Trace32 from the remote server that directly managed the emulator. After that, we loaded the ELF file that came out from the build procedure and we selected the right memory partition of the microcontroller where to flash the firmware.

We added to the “watch window” every variable we needed, to understand if our implementation was working; for example, the MRBs variables showed if CABs were cyclically accessed, the “Consistent” variables instead, were used as counters

initialized to 0, that increment their value when the associated reading Runnable of comp1 receives inconsistent data.

The first result was quite disappointing because, even if the MRBs were correctly updating their values, “Consistent” counters were continuously increasing. Therefore, we set break points in every Runnable and we stepped into C code lines to keep track of what was happening. What we noticed was completely unexpected: in fact, variables that count the number of users per buffer were not changing their values. Hence, we visualized the correspondent Assembly code of the C code lines where the values had to change and indeed, there were no instructions that could modify the variables. From this result, we understood that the compiler was optimizing our code, “thinking” that the consecutive increase and decrease of the same variables were useless operations: it cannot see that a Runnable in another core needs the updated value of this shared resource. To solve this problem, we declared that shared variables as “volatile”, so that the compiler cannot optimize them.

4.8. Validating design

Once seen that the model was working with a very simple SW-Cs Architecture, we decided to test our design applying it to a more complex architecture in which the system has two components in three different cores. We chose this model to validate our design, because it can be considered closer to a real architecture, dealing with many different data exchanges between every core.

We created three “Composition Components” named “Core0”, “Core1” and “Core2” that represent the three cores of the Tricore MCU. Inside each of them, we have included Application Components that have to run on the specific core. Each Application Component contains Runnables with Sender/Receiver Access Points that can have either correlated Data Types or not (to better simulate a real use case). We repeated the same steps described in previous chapters: after the definition of the new SW-Cs Architecture Design, we configured the BSW, we extracted and implemented templates, we built the software and we tested it in ECU.

A peculiarity of the ICC that has not been previously specified is that it allows to consistently transfer multiple variables together without defining a new data type as an aggregation of multiple types. With the IOC, this does not happen because it

guarantees a consistent writing of a single data; therefore, if we want to send more related variables, we must create a new type that encloses them.

The facility, that we introduced, allows us to consistently send data elements produced by different SW-Cs; thus, with the validating design, we also checked this property.

Tests in ECU were performed several times changing the Runnables' mapping into tasks with different periodicity, to simulate many use cases where data are sent at different rates.

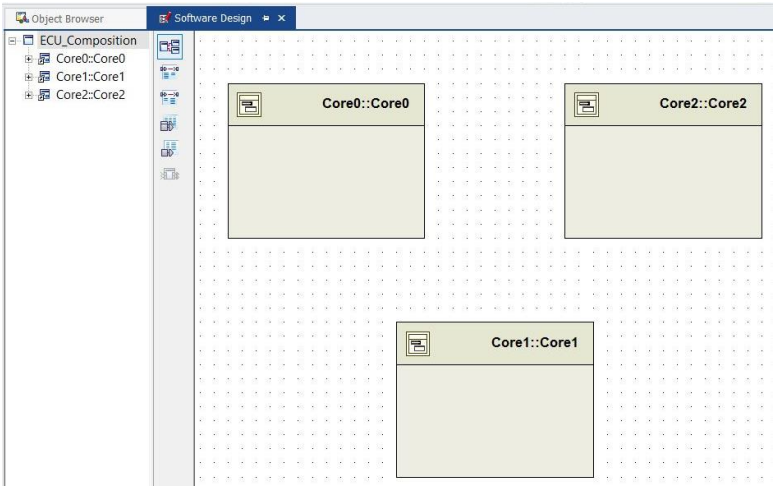

Figure 27: ECU Composition: Cores' subdivision

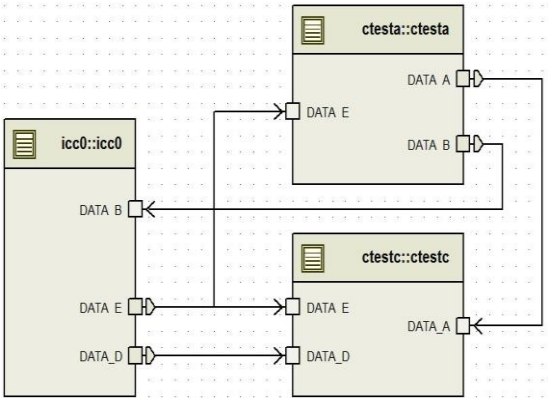

Figure 30: Core0 SW-Cs

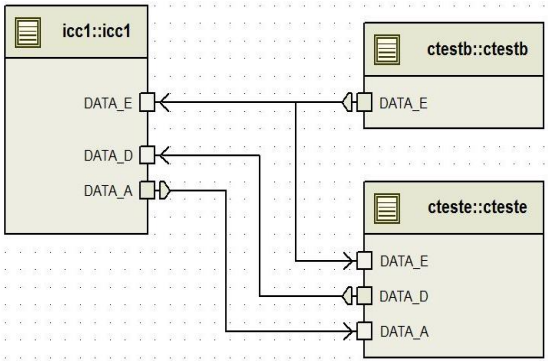

Figure 29: Core1 SW-Cs

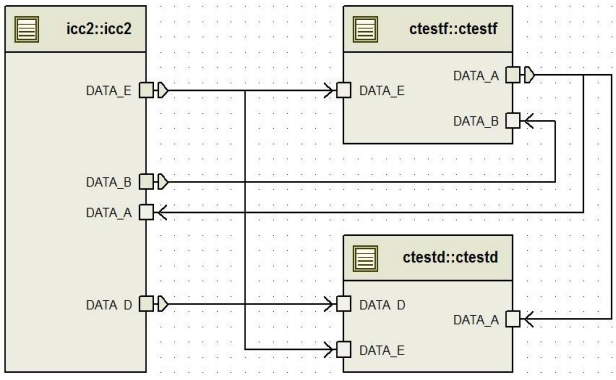

Figure 28: Core2 SW-Cs

5. Code generator development

5.1. Diagrams

5.1.1. Analysis class diagram

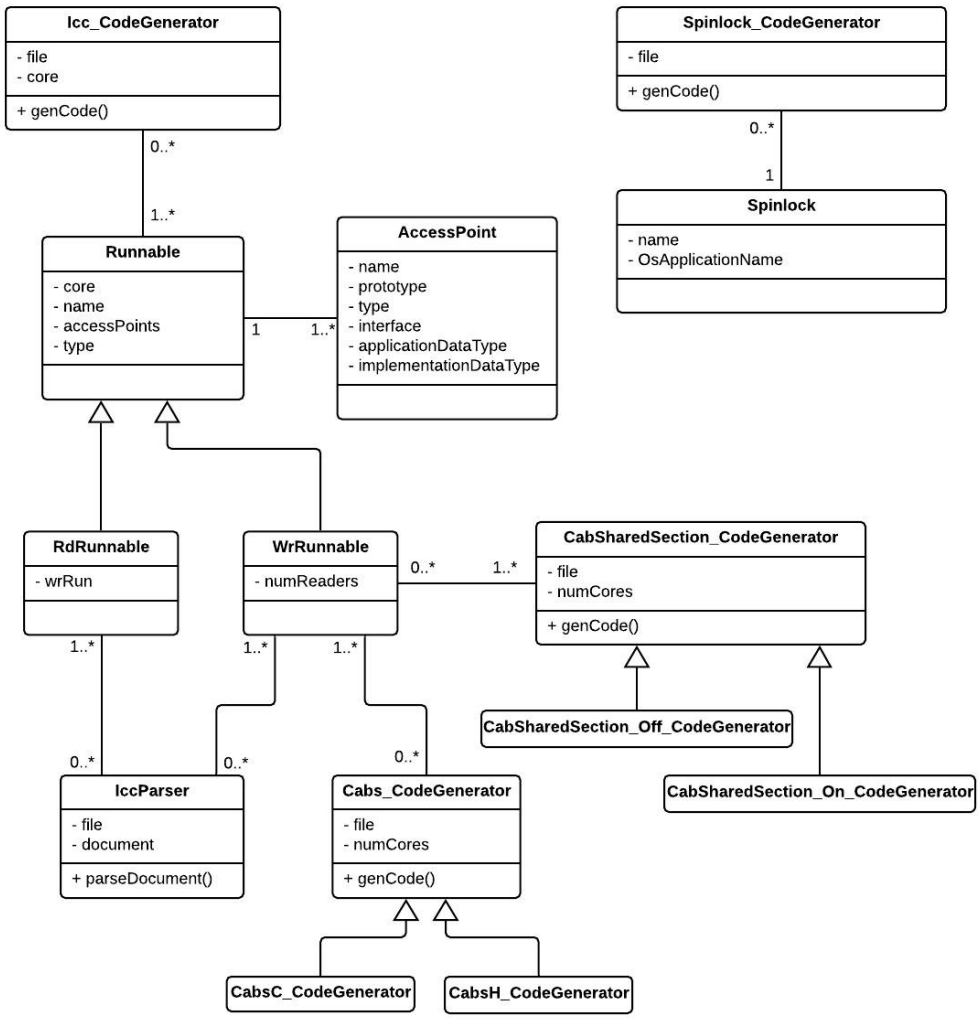

Figure 31: Analysis class diagram

5.1.2. Complete design class diagram

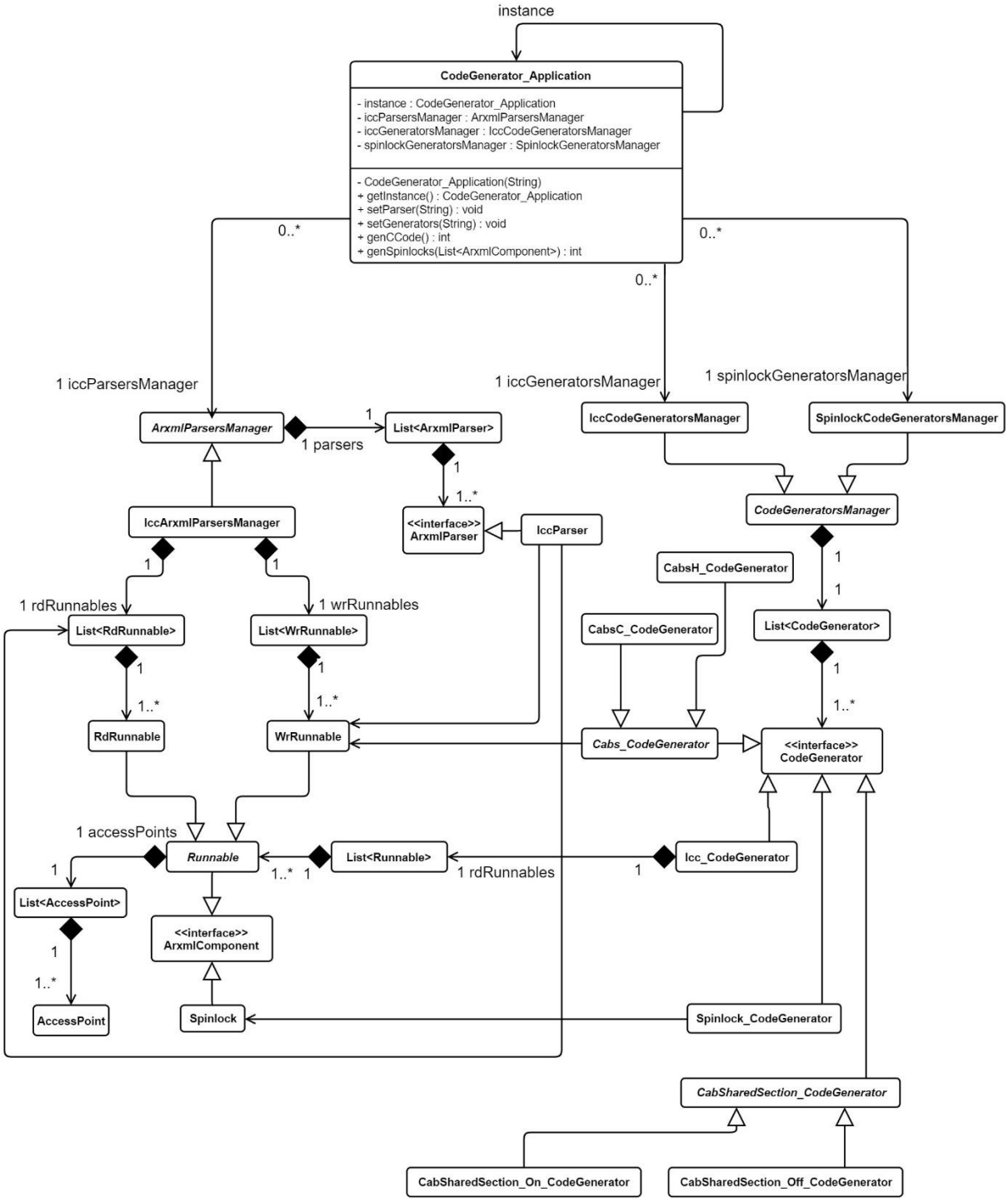

Figure 32: Complete design class diagram

5.1.3. ARXML parsing - design class diagram

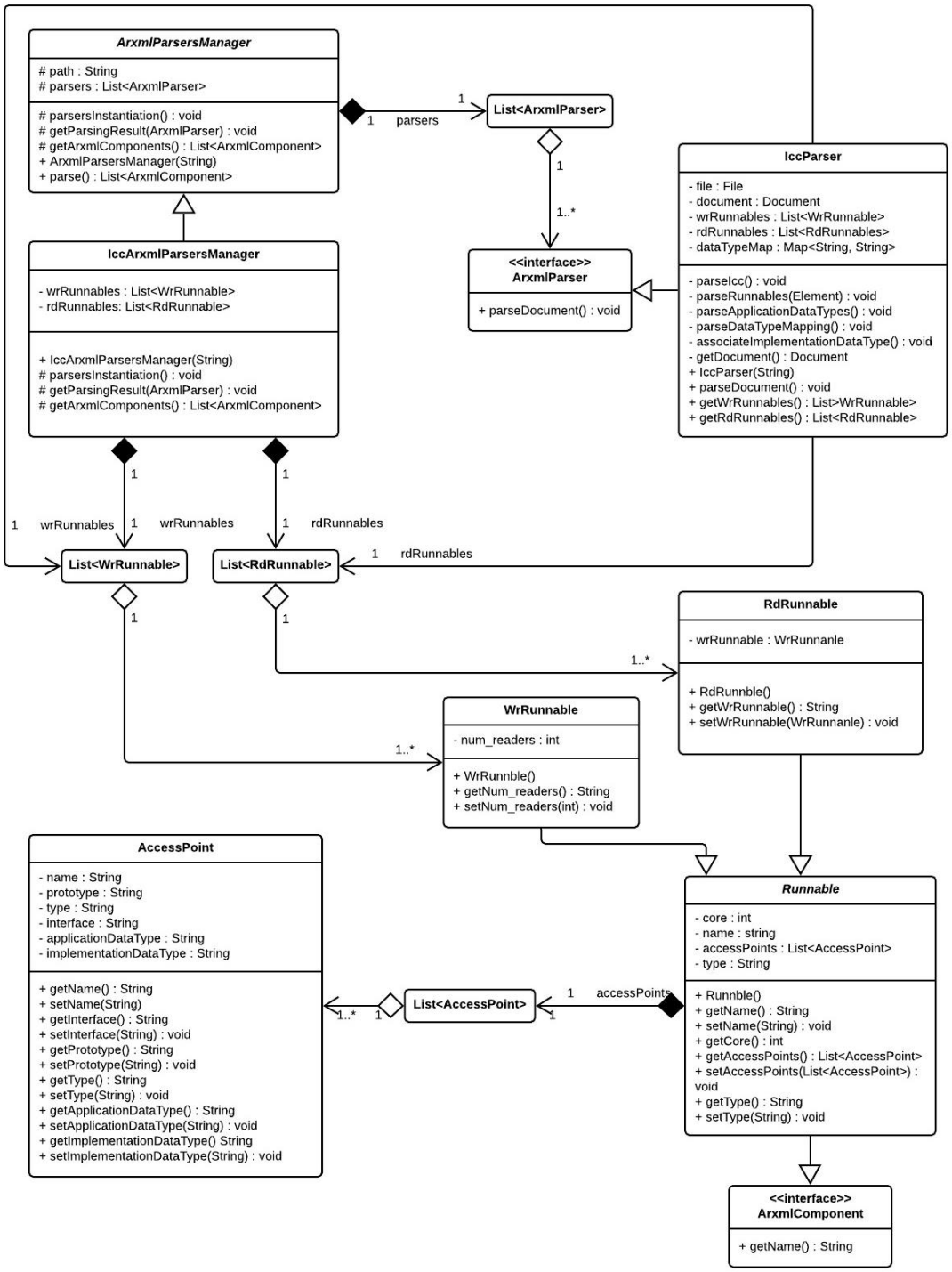

Figure 33: ARXML parsing - design class diagram

5.1.4. Code generation - design class diagram

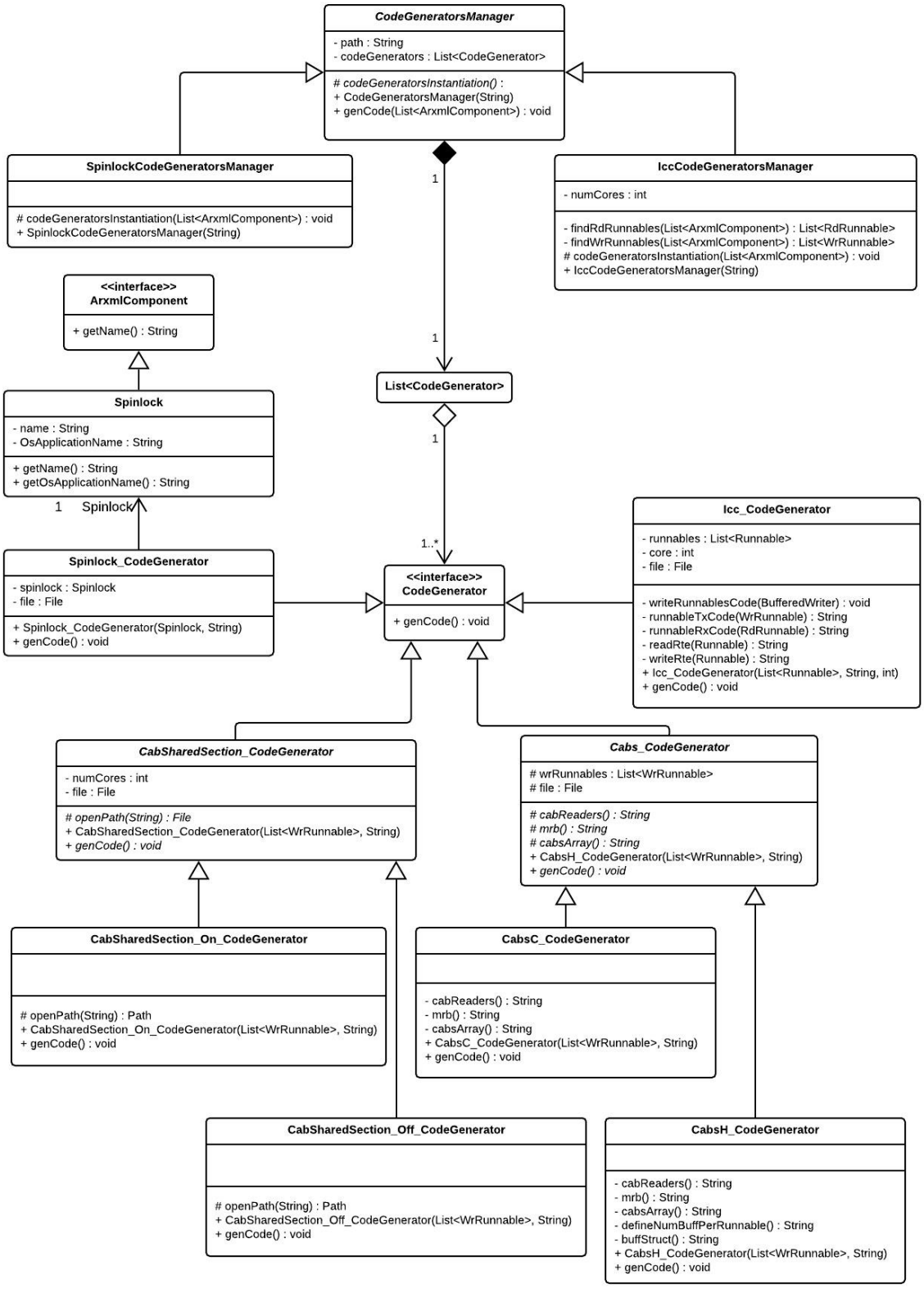

Figure 34: Code generation - design class diagram

5.2. General description

After the validation of CAB code that we performed in ECU, we were sure that the auto-generation tool should have been based on that structure. Therefore, we started the code generator development considering as system requirements what we obtained from the previous development process.

The aim of the generator was to automate as much as possible the procedure of the Inter Core Communicator implementation. To do that, we wanted that our application could substitute the template implementation phase (generating C code) and the Spinlocks' definition (generating an ARXML to be included in the BSW configuration).

Java was the programming language chosen for the generator, due to its WORA ("write once, run anywhere") characteristic that allows to run it on every platform which supports the Java virtual machine. The IDE mainly used for the development was Eclipse, but we preferred NetBeans to design the GUI (Graphical User Interface).

The software was originally designed to be run on a CLI (Command Line Interface), but, during the development, we realized that a GUI, even if trivial, could simplify the use of the generator for the end user. Thus, we decided to add it just when every Java "exception" had already been managed printing a message in the CLI. Redirecting every output to a GUI would have been a waste of time, so we decided to cope with this problem imposing the user to launch the program through an executable file (".bat" Windows, or ".sh" in Linux). In this way it will automatically open a CLI, where any errors can be printed, and then GUI to interact with the application. The output redirection could be a future improvement to make the software independent from the CLI, so that the user can open it simply double-clicking the ".jar" file.

As can be noticed by the analysis class diagram (in "Diagrams" section), our application contains two completely separated and independent generation phases: one that generates ICC, "cabs" and "cab_shared_sec" files (left part of the graph) and one that generates Spinlock ARXML (right part of the graph).

"CodeGenerator_Application" is the class that contains the logic of our application, for this reason we adopted the "Singleton" design pattern that guarantees its instance to

be unique for the entire execution of the program, requiring it through a static method “getInstance()” that returns a new instance just if not yet present. This class is in charge of instantiating manager objects that are used to parse and generate ARXMLs and also to generate C code. These managers are useful to hide the algorithmic and structural complexity of other classes, managing more parsers or code generators. Their characteristics will be better explained in next sections.

5.3. ARXML parsing

The “IccParser” that we find in the analysis graph is an XML parser that has to extract both writing and reading Runnables (“RdRunnable” and “WrRunnable”) information from an ARXML that has been exported from the DaVinci Developer. In fact, during the ICC development, we previously generated templates; with this procedure, instead, we have to export every ICC component that we have in the Developer project, so that our tool can read it.

5.3.1. ARXML components

In design phase we used the “Façade” pattern, defining the “ArxmlComponent” interface with just a symbolic method “getName()”. The aim of this pattern is to mask the interaction of complex components behind a simple one: Runnables and Spinlocks can be seen as ArxmlComponents (meaning they are generic elements that can be extracted from ARXML files). We will see that another ArxmlComponent will be a Spinlock, that we will create in order to generate an ARXML file, instead of being extracted from it.

Why are we distinguishing writing and reading Runnables? The reason is linked to the fact that we are considering a CAB as uniquely identified by its unique writer (basing on previous assumptions about our CAB definition), so a writer has, as particular property, the number of readers that receive its message. Reading Runnables instead, can read from just one writer, that is why it is their particular property.

Therefore, to better explain this concept: we don’t have CAB objects inside our program; by the way, we can identify CABs just with lists of writing and reading Runnables, linked each other.

A Runnable has associated its “AccessPoints”, objects representing Access Points and all their properties, such as the “implementationDataType”, which represents the real C code data type that the generator uses to define variables to be transferred.

5.3.2. ARXML parser

IccParser is implemented as a DOM (Document Object Model) parser, that is a good choice for documents like ICC ARXMLs because they are quite small (few megabytes) and have a complex structure. It returns parsed documents as tree structures, easier to be analysed and to extract useful data from. The “parseDocument()” method exploits parsing functionality to instantiate Runnable objects and fill the internal class lists.

With the “Façade” pattern, as before, we designed the “ArxmlParser” interface, to make implementing classes, like “IccParser” (the only one needed in our specific application), externally seen as a simpler class with just the “parseDocument()” method. In fact, this allowed us to define the abstract “ArxmlParsersManager” in charge of managing every generic associated ArxmlParser, that in this case is only IccParser. Its method “parse()” instantiates ArxmlParsers calling the abstract method “parsersInstantiation()” and then the “parseDocument()” of each parser. “IccArxmlParsersManager” is the necessary class that concretizes abstract methods of the ArxmlParsersManager with Runnables dependent algorithms and leaving it independent by them.

5.4. Code generation

The “Façade” pattern was also used to define a “CodeGenerator” interface that only shows the “genCode()” method, hiding the complexity of all the implementing classes that in our program are four:

- “Icc_CodeGenerator”: used to generate “icc<NumCore>.c” files;
- “Cabs_CodeGenerator”: abstract class concretized by “CabsC_CodeGenerator” and “CabsH_CodeGenerator”, to generate “cabs.c” and “cabs.h” files;
- “CabSharedSection_CodeGenerator”: abstract class concretized by “CabSharedSection_On_CodeGenerator” and “CabSharedSection_Off_CodeGenerator” to generate “cab_shared_sec_on.h” and “cab_shared_sec_off.h” files;

- “Spinlock_CodeGenerator”: that produces “Generated_Spinlocks.arxml” file using Spinlock objects that implement ArxmlComponent.

The “CodeGeneratorsManager”, with its “genCode(List<ArxmlComponent>)” is in charge of instantiating CodeGenerators and calling their “genCode()”, similarly at ArxmlParsersManager. It is an abstract class extended by “IccCodeGeneratorsManager” and “SpinlockCodeGeneratorsManager” to respectively perform a C code and an ARXML generation.

5.5. GUI

A GUI has been created with the aim of simplifying the use of the application for the end user. It is based on a single frame connected to the “CodeGenerator_Application” through a “FrameController” that manages the communication between them.

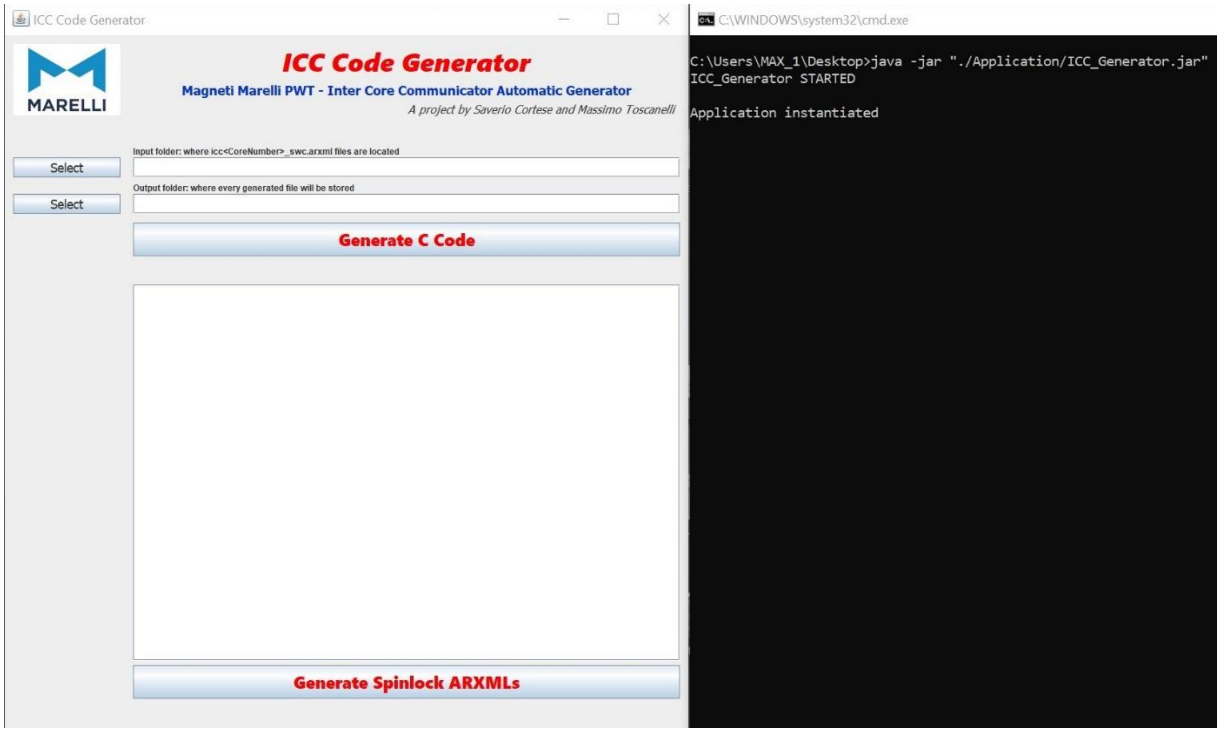

Figure 35: Code Generator Graphical User Interface

To generate C code, input and output folders must be selected either pressing “Select” and choosing the right ones in the appearing frames or writing the right paths in proper text fields. Then the “Generate C Code” button can be pressed and, of course, the generation of C code starts.

Once terminated the process, the application will notify the number of Spinlocks that have to be generated, considering that they are not used for CABs with one reader. Basing on this number, name and OsApplications associated to each Spinlock have to be provided typing them in the lower text field. A Spinlock can be defined writing its <SpinlockName> followed by a tabulation, followed by an <OsApplicationName>, followed by a tabulation, followed by an <OsApplicationName> and so on, for the number of <OsApplicationName> associated to the Spinlock. A new Spinlock must be defined in a new line (they are separated each other by an end-of-line).

After that, the “Generate Spinlock ARXMLs” button can be pressed and the “Generated_Spinlocks.arxml” file will be generated.

It can be noticed that the application has two completely separated sub-programs that have different functionalities, as defined in high level design phase; in fact, we can simply generate C code without Spinlocks’ ARXML, or vice versa, being not strictly linked operations (we just need to know how many Spinlocks we have to declare).

The way we define Spinlocks, writing in a text field, can be considered a rough solution to be substituted in future with a better graphical structure, maybe equipped with buttons to add singular Spinlock and OsApplication text fields. This implementation can reduce the likelihood of text formatting errors. However, the current solution, although very simple, is more flexible, because Spinlocks can be added copying and pasting them from any text file configured before the program opening.

5.6. User Guide

This chapter will explain how to properly use the software to obtain a correct ICC generation for our architecture.

1) Define a proper SW-C Architecture

Open the system project with DaVinci Developer to define a proper SW-Cs Architecture that integrates ICCs.

Depending on the number of cores that have to inter-communicate in our system, we define SW-Cs named **icc<CoreNumber>** that will represent the ICC for the related core.

Inside them, we name transmitting Runnables as **<TxRunnablePrefix>_Tx**, where **<TxRunnablePrefix>** is defined as **icc<CoreNumber>_<RunnableLabel>**.

Receiving Runnables, instead, are named **icc<CoreNumber>_<TxRunnablePrefix>_Rx**, where **<TxRunnablePrefix>** is the one of the related transmitting Runnable.

Runnables triggers are the periodic activations at 10 milliseconds adopted as standard in MMPWT.

Inside ICCs we have to create just Runnables necessary for components communication; Therefore, we do not need to set calls and events.

2) Export ICC components

Export ICC components in **AUTOSAR V4.4.0** ARXML format following the naming convention **icc<CoreNumber>_swc.arxml**.

3) Run the ICC_Generator

Run the ICC_Generator clicking on the “ICC_Generator.bat” file.

This will open a console (where generation results will be notified) and a GUI (where the user have to insert parameters).

In the GUI select the correct paths of the folders where **icc<CoreNumber>.c** files are located and where to store output files.

Click the “Generate C Code” button.

Once the generation is completed, write in the lower text field as many Spinlocks as the application will notify, respecting the naming convention defined in “GUI” chapter.

Inserted every spinlock we want to generate, the “Generate Spinlock ARXMLs” button can be pressed and the “Generated_Spinlocks.arxml” file will be generated.

NOTA BENE:

- Running the “.bat” (if using Windows OS) instead of the “.jar” file is very important because in this way the console is opened to show generation results and, eventually, errors.
- “.bat” file has been defined to be on the same folder of the “Application” folder that contains the “.jar” file.

4) Correctly map ICC Runnables into tasks

Open the system project with DaVinci Configurator Pro.

Each transmitting ICC Runnable has to be mapped at the end of the task where we decide to make it run, while each receiving one has to be mapped at the beginning.

In this way, we guarantee that every task cycle ends updating every modified variable or that starts with updated values, exploiting ICC functionalities.

5) Import the “Generated_Spinlocks.arxml” file into DaVinci Configurator Pro

In DaVinci Configurator Pro go into “File”, then “Import” and the “Module Configuration Import” dialog window will appear.

Add the “Generated_Spinlocks.arxml” file ignoring UUIDs.

At the end of the process, DaVinci will switch into “Comparison Mode”, from which we need to add Spinlocks and ignore all the removed elements.

This passage is very important, if we do not ignore everything that is removed, we can lose configuration data. This is due to the fact that DaVinci computes the differences among the imported file and the rest of the project and we just have to add what is contained in the “Generated_Spinlocks.arxml” file, not to substitute it with the project configuration data.

6. Conclusions

The conducted work in this thesis was a chance to study in deep the AUTOSAR architecture, the main standard for embedded software development in automotive industry.

We focused our attention on the inter-core communication features provided by AUTOSAR, to understand what needed to be improved according to the Magneti Marelli use cases. The study of an old implementation, already present in the company and based on an alternative communication mechanism, allowed us to design our solution. Therefore, many software architectural levels have been analysed and several cutting-edge development tools have been used.

The Inter Core Communicator model resulted to be a satisfactory solution for the company, since it is based on a solid and configurable model to be reusable on generic Magneti Marelli multi-core implementations, independently on the microcontroller used.

To effectively understand which performance increase a control unit can have thanks to our solution, we should have verified the difference in speed between the implementation of a commercial product that adopts the AUTOSAR IOC and the same one that instead uses the ICC. Unfortunately this work would have taken too long, because we would have to deactivate the IOC of an already existing ICC3 project and reimplement its multi-core functionalities adding our ICC as a complex driver. What we did, was to take the basic software, as released by Vector, and then create simple components to test our code.

After the ICC definition, we developed a code generator that automatically generate it from project configuration files. Our tool has been tested and works fine. However, it can be also improved to add new features and ease the user experience. For example, its modular structure allows a possible further increment of its functionalities adding different ArxmlParsers or CodeGenerators.

Acknowledgments

I thank Saverio Cortese (from Magneti Marelli) for his valuable advice and guidance during my internship work in the company and Prof. Paolo Torroni (from Alma Mater Studiorum – University of Bologna) for the useful suggestions he gave me to develop my thesis.

I also thank my friends and my parents that supported and believed in me during my university studies.

References

[1] Hard Real-Time Computing Systems – G. Buttazzo (Springer-Nature New York Inc; 3rd edition)

[2] Automotive Embedded Systems Handbook (Industrial Information Technology) - Nicolas Navet, Francoise Simonot-Lion (CRC Press; 1 edition)

Webliography

[3] Introduction to AUTOSAR https://www.autosar.org

[4] AUTOSAR – History – Concept and goals – Architecture https://en.wikipedia.org/wiki/AUTOSAR

[5] AUTOSAR – Layered Software Architecture https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[6] AUTOSAR – Explanation of Adaptive Platform Design https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-10/AUTOSAR_EXP_PlatformDesign.pdf

[7] AUTOSAR – Specification of Operating System https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_OS.pdf

[8] Summary of AUTOSAR Competence https://pdfs.semanticscholar.org/8e53/429e50fc52b767e50126f3d922305c555e7c.pdf

[9] MICROSAR – Product Information https://assets.vector.com/cms/content/products/microsar/Docs/MICROSAR_ProductInformation_EN.pdf

[10] Functionality assignment to partitioned multi-core architectures http://www.imm.dtu.dk/~paupo/publications/Maticu2015aa-Functionality%20assignment%20to%20pa-a.pdf

[11] Migrating a Single-core AUTOSAR Application to a Multi-core Platform: Challenges, Strategies and Recommendations http://publications.lib.chalmers.se/records/fulltext/250043/250043.pdf

[12] Kernel Overview http://hartik.sssup.it/overview.html

[13] IBM Rational Synergy V7.2.1 documentation
https://www.ibm.com/support/knowledgecenter/en/SSRNYG_7.2.1/com.ibm.rational.synergy.doc/helpindex_synergy.html

[14] Executable and Linkable Format (ELF)
https://elinux.org/Executable_and_Linkable_Format_(ELF)

[15] Integrazione Lauterbach e Vector Software
https://www.lauterbach.com/frames.html?tut-i_trace32-vectorcast.html

[16] TC29x B-Step – User's Manual
https://www.infineon.com/dgdl/Infineon-TC29x_B-step-UM-v01_03-EN.pdf?fileId=5546d46269bda8df0169ca1bdee424a2