

DASIA 2024 – FULL PAPER	
TITLE	SSLA: A standardized and reusable software framework for avionics and space applications
AUTHORS	Miguel López ¹ , Alba Rozas ¹ , Jesús Zurera ¹ , Rafael Polonio ¹ , Sergio Ramírez ¹
PRESENTER / CORRESPONDING AUTHOR	Alba Rozas Cid alba.rozas@aeroespacial.sener
AFFILIATION	¹ SENER Aeroespacial, S.A. (Severo Ochoa 4, 28760 Tres Cantos, Madrid, Spain)
SESSION	SOFTWARE – A06 presenting slot, May 28 th 2024.

ABSTRACT

Flight software development and testing has traditionally absorbed a significant part of a mission’s time and monetary resources. This is mostly due to its lack of modularity and excessive coupling to each mission’s particular requirements, which hinders its reuse in future projects. In recent years, modular flight software architectures have emerged to solve this issue and generate more modular and reusable software components for avionics and space applications. NASA’s core Flight System (cFS) is perhaps the most well-known and used framework for this purpose, following an open-source approach. Developing flight applications on top of cFS or any of its alternative frameworks, complying with their interfaces and following their architecture results in more modular, standardized, testable and ultimately reusable applications, reducing the costs of subsequent missions.

However, these flight software frameworks can still have a steep learning curve for application developers and mission system engineers, due to its increasing features and somewhat complex software structure. To address this, we have developed the SENER Service Layer API (SSLA), a new software framework that further abstracts the application developer from the underlying software and hardware execution platform. In addition to this enhanced abstraction, SSLA has been designed to cover other necessities of flight software development. For instance, it provides an innovative feature that supports native integration of FPGA cores and their access from the software applications. This makes it especially appropriate for its use in projects that use SoC processors with integrated FPGA fabric, greatly helping the process of allocating functionalities to FPGA cores or software applications depending on the mission requirements.

SSLA not only includes the abovementioned features built into its API, but also facilitates software creation in a more pragmatic way by providing a development and testing environment specifically tailored for it. In this paper we present the SSLA framework, introducing its features and its integration into a complete avionics execution platform. A quantitative analysis of its performance and some software-related metrics are presented based on a representative application use case.

1 INTRODUCTION

The development of software for avionics and space missions has usually consumed a great percentage of the projects’ budget and schedule, requiring a lot of human resources. This is because flight software has traditionally been created ad hoc to comply with each mission’s requirements and specific characteristics. Thus, it often lacked modularity and had components that were too tightly coupled to each other, which ultimately prevented or greatly complicated their reuse in successive missions. In the last decade, several flight software architectures have been presented to address this problem and facilitate reuse. NASA’s *core Flight System* (cFS) is arguably the most well-known in the space industry, with a very active open-source repository and continuous updates and enhancements to its code base [1]. In the European context, the *outpost-core* spacecraft platform developed by the German Aerospace Center (DLR) is also aimed at this modularization and enhanced reusability of space software [2][3]. The French *LVCUGEN* framework proposed by CNES is another partial example of these approaches [4].

However, at SENER we have identified some practical limitations in the use of these software frameworks when working on specific projects. Our conclusions have been focused on NASA's cFS but they are applicable to other alternative frameworks, as long as they are similar in their underlying approach (e.g. heavy modularization of components, high number of abstraction layers, enhanced parametrization and configurability). While these features are what actually supports the desired modularity and reusability of applications, they come with a cost in terms of the steep learning curve that these frameworks require to be proficient in their use. Flight software projects will benefit from a simplified application interface in which the underlying complexities of these architectures and frameworks are present and usable, but hidden to the application developer. In this manner, two perspectives can co-exist during the development of any flight software project. First, the application developer perspective consists only in having a simplified and abstracted API to build his/her applications that fulfill the mission functionalities, viewing the rest of the execution platform as a black box. On the other hand, the platform developers require a deeper knowledge of the underlying features and parameters of the used flight software framework, and even of the operating system and hardware, in order to fine tune it for the mission requirements. By splitting these perspectives, the workflow of both teams is greatly eased and expedited, benefitting the whole project.

To support this approach a new software module has been developed at SENER, called SENER Service Layer API or SSLA, that we present here. SSLA is itself a software component that presents the mentioned API for applications, but also includes some additional internal services and features that cover some identified necessities in the development of flight software. It is written in the C-language and is designed to be deployed on top of the sMart Integrated Avionics (MIA) platform, a software and hardware execution platform architecture also devised by SENER, acting as a single entry-point for applications. MIA's fundamental approach is having independent layers that provide the required execution platform functionalities and can be configured and selected decoupled from one another. Apart from other selectable software components (e.g. an operating system or a hardware virtualization layer), MIA supports the concept of a 'service layer'. This service layer consists of a set of flight-related functionalities (e.g. TM/TC, housekeeping, task management and data exchange, etc.) that are available to applications, so that they do not have to be implemented again in every mission. In principle, these service layer functionalities can be fulfilled by any potential flight software library or framework as long as it provides or can be expanded to support all expected functions. Specific details of the rest of the software and hardware layers of MIA are out of the scope of this paper, but an overview will be given in the following section.

On its early concept, SSLA has been designed to be compliant with a service layer based on NASA's core Flight Executive (cFE). This way, in its current version SSLA wraps cFE's core services and works alongside other cFS components and applications. However, its design allows it to be ported to work over an alternative flight software architecture, or even entirely substitute it by internally implementing its own flight-related services. Apart from this enhanced abstraction, SSLA includes an innovative built-in support for accessing FPGA cores from the software applications. Also, SSLA is provided alongside a software development kit (SDK) specifically tailored to its characteristics. Some useful tools of this development environment are an automated tool for auto-coding FPGA access drivers, and a semi-automated GUI tool for generating applications based on a standard application template.

As of now, the approach and concept of SSLA has been applied and is being successfully implemented in a couple of real space projects. One of them is the development of an Autonomous Flight Termination Unit, where SSLA-based applications are in charge of tracking the navigation of a launcher and detecting if/when it should be terminated based on a series of mission rules. Another example of SSLA usage is to host the data handling system of a small satellite platform currently under development for on-orbit repair.

This paper is structured as follows. The first section provides an introduction of the context in which the presented SSLA technology is used. Then, the second chapter briefly reviews some relevant works and the architecture of the execution platform in which SSLA is currently deployed. The third chapter is the central section of the paper, explaining the proposed SSLA framework in detail, along with its usage, features and some related development tools. Finally, some quantitative metrics of SSLA are foreseen to be presented in the fourth section, and some relevant conclusions will be extracted in the last one.

2 RELATED WORKS AND OVERVIEW OF THE MIA PLATFORM

As described in the introduction, SSLA is a software module designed to be deployed on top of the sMart Integrated Avionics (MIA) execution platform, serving as the single entry-point for applications into the platform. The MIA platform is described in detail in [5], and a depiction of its architecture is included in Figure 2-1.

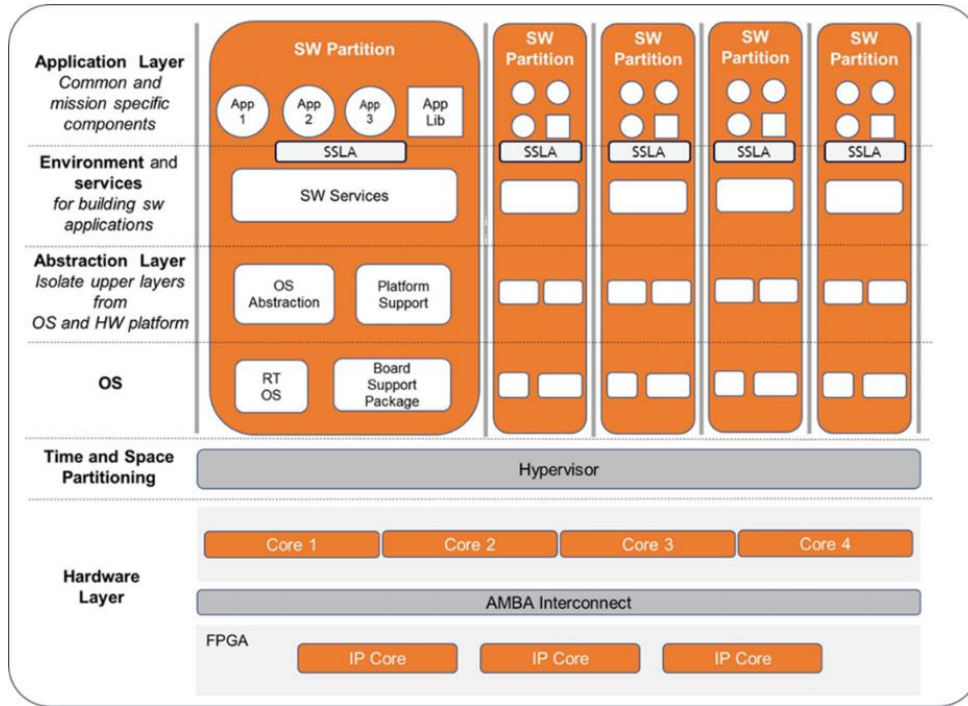


Figure 2-1: MIA Execution Platform Architecture

It is important to note that MIA is conceived as an architecture and framework to combine several software components into a ready-to-use execution platform in which applications can be run easily. In most contexts (and throughout this paper), when referring to the MIA platform, the hardware processor in which it is deployed is also considered part of the platform. A brief overview of the layers from bottom to top is described as follows:

- **Hardware layer:** it includes the actual hardware physical components in which the upper software layers are executed. MIA is specially designed to support multi-core execution and systems with an integrated FPGA, so it is particularly tailored to be deployed in Multi Processor System on Chip (MPSoC) hardware devices.
- **Time and Space Partitioning (TSP) layer:** if present, this layer provides the ability to split the missions' functionalities into different partitions guaranteeing independence between them. The layer's functionality is based on a hypervisor that spatially or temporally allocates the hardware resources to each partition.
- **Operating System (OS) layer:** it supports several operating systems or a bare-metal configuration if no OS is present.
- **Service Layer:** it provides generic flight software services and functionalities to the applications, as well as providing a standardized interface for application development. As of now, this layer is based on NASA's cFE and its core services and modules [6], such as the Software Bus (SB), Executive Services (ES), Event Services (EVS), Message (MSG) and Time modules to name a few. The SSLA module acts as the upper interface of this layer, enhancing the abstraction to the rest of the platform and wrapping the underlying services and functions for the applications.
- **Application Layer:** it contains the applications that provide the mission-specific functionalities. Ideally, this is the only layer where new software should be implemented from mission to mission.

The MIA approach followed by SENER is thus not focused on covering all the layers functionalities by developing every software component internally, but rather to support the integration of own and third-party software items. These software components may be procured in different ways depending on their availability and licensing characteristics. For instance, a licensed COTS hypervisor, such as the Xtratum Next Generation (XNG) from FentISS [7] can be deployed on the TSP layer, while a space-qualified open-source operating system such as RTEMS can be used in the OS layer [8].

As explained in the introduction, it is a task for the platform developer to integrate and configure MIA to fulfill the requirements of each particular mission and guarantee the compatibility among components. For example, if the service layer provider is selected to be NASA's cFE, the platform developer should make sure to select an OS to which NASA's Operating System Abstraction Layer (OSAL) is ported (e.g. Linux or RTEMS as of now). Understandably, this MIA configuration must be done with a top-down approach, starting from SSLA and going down layer by layer.

2.1 Two distinct perspectives: platform architect and mission application developer

As explained, SSLA is the upper layer of the MIA platform and the one that isolates the mission applications in the top layer from the rest of the platform software components. This allows MIA to support two distinct perspectives within a project's software life cycle: the platform architect/maintainer and the mission application developer, as depicted in Figure 2-2.

The former is in charge of developing, configuring and maintaining all the software layers of the MIA platform below the applications. Depending on the given project, this role may involve just setting up existing layers to the mission scenario specification (e.g. selecting an appropriate RTOS and/or configuring an existing hypervisor). On other projects the platform architect may need to develop new features in any of the layers, to update a given component to a more recent version or even to decide the FPGA-SW allocation of functionalities.

The latter is the mission application developer and his/her role just consists in implementing high-level applications to fulfill the mission specification and required functionalities. To do this, the developer only needs visibility of the SSLA API, seeing the rest of the platform as a black box. The existence of the SSLA layer allows us to provide this enhanced abstraction and leads to more modular, standardized and ultimately reusable applications. This greatly reduces the time and budget allocated to the software development task of a given mission.

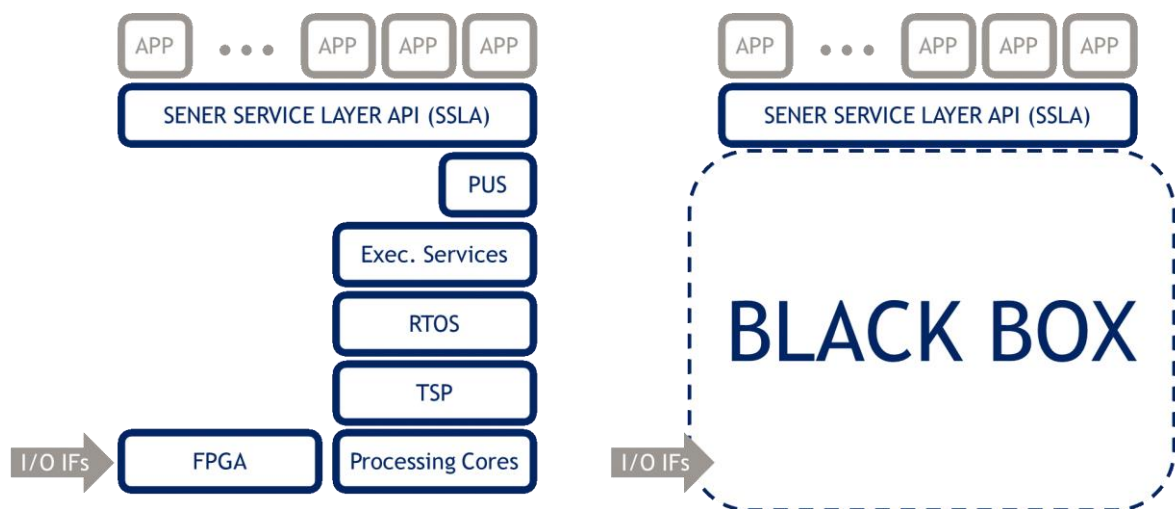


Figure 2-2: MIA distinct perspectives: platform architect/maintainer (left), mission application developer (right)

3 SENER SERVICE LAYER API (SSLA)

As described, SSLA has been introduced in the context of the MIA platform as an extra abstraction layer on top of the core software services. Even though cFE or its alternative flight software frameworks already provide their own API for application development, the introduction of SSLA has been motivated by the following aspects:

- 1) **Isolate mission applications from changes in the service layer provider.** It is the SSLA component the one that absorbs all these changes. For instance, an already developed and tested application that was deployed on top of an SSLA configured to be deployed on the Caelum version of cFE (2019), would not have to be changed at all to work on top of an SSLA ported to the Draco version of cFE (2023). This is especially important since cFE does not guarantee backwards compatibility in their subsequent releases. This greatly helps reuse of applications from mission to mission and helps avoid obsolescence.
- 2) **Ability to introduce new services or features not present in the existing flight software frameworks.** The most important example of these innovative features is the native support for FPGA-allocated functionalities. This is already implemented in the 'IP drivers' module, a built-in component of the SSLA library. In our experience, most recent avionics projects benefit from the existence of MPSoC processors with FPGA fabric. FPGA cores can implement mission functionalities with strict time execution constraints or intensive external interfacing (e.g. retrieving data from sensors), this way reducing the load and simplifying the software applications. However, the co-design and integration of the FPGA-SW interface is a time-consuming effort in this kind of projects. This built-in module standardizes the access to FPGA cores greatly facilitating their integration and isolating the workflow of the software and FPGA teams.
- 3) **Enhanced abstraction and simplified application design for app developers.** As described, the complexities of a complete flight software framework such as cFS can seem daunting for new developers and have a steep learning curve before acquiring all the necessary knowledge to properly use it. With SSLA, the structure of flight software applications is heavily standardized and simplified for the basic use cases. This greatly reduces the effort and time needed to have an application coded, running, and tested. In turn, the ability to customize the application structure and implement more complex functionalities is retained and can still be used by more expert developers.
- 4) **Support for a Software Development Kit with automated tools.** The enhanced standardization provided by SSLA allows us to generate a set of tools to support the development of flight software in a more practical way. Some of these tools greatly reduce the development and testing time by automating repetitive time-consuming tasks.

3.1 SSLA architecture

Since SSLA is designed as a part of the MIA approach, it is especially targeted to MPSoC processing devices. These devices generally contain multiple CPU processing cores as well as reprogrammable FPGA fabric in a single chip, with all the required peripherals, memory banks and interconnections. A common naming scheme in these devices refers to CPU cores as the Processing System (PS) and to FPGA fabric as the Programmable Logic (PL) of the device. The dual abstraction interface that SSLA presents to software applications that execute on the PS of the device can be seen in Figure 3-1. As depicted, SSLA serves as an interface for applications to both the underlying software services provider (cFE in the figure), and to the FPGA cores implemented in the PL.

3.1.1 SSLA components

As described, from the application developer's perspective, SSLA contains all the functionality required to create complete mission application software. In order to achieve this, the SSLA module is structured in three distinct components, as depicted in Figure 3-2. It is important to distinguish the scope of the term 'application' here, because a complete 'mission application' from the perspective of the whole system (e.g. a 'mission application' referring to the whole TM/TC system of a satellite), is almost always composed of several SSLA applications, each fulfilling part of the mission functionality (e.g. an SSLA app for reading telecommands from an external bus, another

The diagram illustrates a system architecture divided into three main vertical sections: **HW (External)**, **PROGRAMMABLE**, and **PROCESSING**.

HW (External): Contains two identical **HW 1 SENSOR/ACTUATOR** and **HW n SENSOR/ACTUATOR** blocks, each connected to the programmable logic below via four bidirectional arrows.

PROGRAMMABLE: This section contains multiple **FPGA BLOCK** units. Each block is divided into **LOGIC** and **DMA** components. The **LOGIC** part shows a state transition diagram with nodes S_i , S_j , and S_k . The **DMA** part contains a register table with the following structure:

ID	0x0000	DATA A	0x0000
READBACK	0x0008	DATA B	0x0008
...
Register Z	0xZZZZ	DATA Z	0xZZZZ

Each **FPGA BLOCK** is connected to the **AXI BUS** via a red double-headed arrow labeled **HP** (Host Port).

AXI BUS: A central yellow horizontal bar representing the communication bus.

PROCESSING: This section contains several layers and components:

- SSLA LIB:** A purple horizontal bar representing the Secure Software Library.
- SSLA APP 0, SSLA APP 1, ..., SSLA APP n:** Blue rounded rectangles representing individual applications, each connected to the SSLA LIB above and below via red double-headed arrows.
- SSLA LIB:** Another purple horizontal bar below the applications.
- SW BUS:** A yellow horizontal bar representing the Software Bus.
- cFS (Control File System):** An orange block on the left.
- cFE (Control Executive):** A large yellow block containing sub-components: **ES** (Executive Services), **EVS** (Event Services), **SB** (Software Bus), **TBL** (Table Services), and **TIME**.
- Operating System Abstraction Layer:** A purple bar at the bottom left, connected to the cFS and cFE.
- Platform Support Package:** A purple bar at the bottom right, connected to the cFE.

The diagram illustrates the SSLA system architecture. At the top is the **SSLA LIB** (purple bar). A purple arrow labeled **Pure facade** points to it from above. Below the SSLA LIB are two identical SSLA application stacks. Each stack consists of an **SSLA App** (yellow box) and an **SSLA Core** (red box), connected by a dashed purple line labeled **interface**. The left stack is labeled **SSLA_app_0** and the right stack is labeled **SSLA_app_n**. A thick red arrow points from the SSLA LIB down to the **cFE** (grey bar). Below the cFE are five blue boxes representing system components: **ES**, **EVS**, **SB**, **TBL**, and **TIME**. Bidirectional red arrows connect each SSLA Core to the cFE.

SSLA Core is the main component of the layer. It defines the structure and execution flow of applications, so that applications are as standard and repeatable as possible. It contains most of the foundation necessary for applications to be compliant with the underlying service layer, in this case the cFE framework. Thus, this is the component that is more coupled to the service layer provider. *SSLA Core* is coded in such a way that generic fixed

structures that individual applications need are initialized there, e.g. list of subscribed messages, housekeeping packets, etc.

The followed approach requires that each *SSLA App* is associated with its own instance of *SSLA Core* by means of an interface file as shown in Figure 3-2. The *SSLA Core* is itself an application in context of the underlying cFE Executive Services (ES) module, and subsequently mapped in the operating system as a single thread. It has the following real-time execution flow:

- 1) Upon being loaded by the ES from the startup script (or later if the particular app is not desired to start from power-on), an *Initialize* function is executed where:
 - a. Global variables, handles and counters are initialized or reset.
 - b. Two cFE Software Bus (SB) reception pipes (buffers) are created and allocated for command and data messages, respectively.
 - c. The *Setup* function of the *SSLA* application associated to the particular *SSLA Core* instance is executed. This *Setup* function is explained in the following *SSLA App* subsection since it contains application-specific code, but in general it consists of declaring and registering SB messages that the application publishes or is subscribed to.
 - d. The *SSLA Core* instance subscribes to every reception message that has been registered in the previous step.
 - e. The *SSLA Core* instance creates and starts any child task that has been registered in the previous step.
 - f. A performance monitoring service is started in the context of CFE.
- 2) After initialization, the *Main* application function is executed. It is based on a cyclic executive pattern, so after the first trigger it will continue running forever, unless the processor is powered off, reset, or the ES service or the operating system kill the application (not foreseen unless in FDIR circumstances).
 - a. The *Main* execution flow is based on a while loop that contains a 'receive message' callback from the Software Bus component. Thus, the execution of the main thread gets halted pending the arrival of any command SB message that the application is subscribed to (through the command pipe created in step 1.b).
 - i. When a command message arrives, it gets processed, and the *Core* goes back to waiting for command messages. The processing function dispatches each incoming message to a specific module, either the ground commands handler or the inter-application command handler. These components contain app-specific functions coded by the mission developer and mapped to each command that's registered in the *Setup* function of the *SSLA App* associated to the *SSLA Core* instance in question. In this manner, the *SSLA Core* structure guarantees that the reception of a specific command message triggers the immediate execution of a specific function or piece of code from the application domain.
 - b. For data messages, the reception does not affect the execution flow. Whenever a new data message arrives, the SB module itself just stores the incoming message in the Data pipe of the *Core* instance (as long as the specific data message ID was registered in step 1.c). Thus, the arrival of data messages does not trigger any synchronous execution, and *SSLA Apps* have to read them asynchronously from the data pipe at any moment they consider appropriate.

- 3) If the *Setup* step has registered any child task for the application, it is mapped to its own operating system thread and started at the end of the *Initialize* function. Thus, for *SSLA Apps* that use child tasks, the functionality is divided into the main thread and as many child threads as created. The execution flow of child tasks is not *SSLA Core* mandated, and each developer may code it in the way that his or her desired behavior requires. For instance, a child task may be started to periodically poll for the value of a certain sensor.

This dynamic behavior is heavily dependent on the structure and functions provided by the NASA CFE Executive Services module. Thus, when porting *SSLA* to a different service layer framework, the *SSLA Core* would be subjected to most of the modifications.

3.1.1.2 *SSLA Apps*

SSLA Apps define the specific mission functionality, split between them to maximize modularity and have a robust and deterministic dynamic behavior in execution time. As explained, they are tailored to the execution flow of their *SSLA Core* instance, so they all have a *Setup* function where the application-related data structures of the *Core* are filled. This *Setup* function allows each application to:

- Register RX Ground Commands to subscribe to. In the context of *SSLA* and cFE, ground commands are an abstraction for any message that arrives from outside the platform (e.g. the actual ground segment in a space mission or a different unit or subsystem in the spacecraft). The physical particularities of how this message actually arrives to the avionics platform is out of the scope for the *SSLA App*. The subscription process determines an ID and payload content for the message, as well as a function callback to be executed when the command arrives.
- Register RX Inter-application Command messages to subscribe to. These are the commands exchanged from one *SSLA App* to another. As with ground commands, this subscription maps a specific function to be executed when the subscribed message arrives.
- Register RX Inter-application Data messages to subscribe to. Similarly to the above bullet, but for data messages instead of commands. This subscription functionality does not map any function to each registered message.
- Register TX Inter-application Command messages to publish. Each app is allowed to declare a configurable number of transmitted command-type messages, determining an ID and custom payload structure for each one.
- Register TX Inter-application Data messages to publish. Analogously, each app is allowed to declare a configurable number of transmitted data-type messages, determining an ID and custom payload structure for each one.
- Register child tasks to be created upon the initialization of the application. As explained in the *Core* section, specific mission functionality may be allocated to any number of child tasks in the context of each *SSLA App*. Their usage is up to the mission planner or developer, but in general they may be used when an application needs to implement an internal behavior not easily translated to the reception of an external command (e.g. a periodic monitoring task).

3.1.1.3 *SSLA Lib*

The *SSLA Lib* (library) defines all the services and functions that are potentially necessary for an *SSLA App*, exposed in a controlled way. This is the component that generates the mentioned two-way abstraction interface for applications: first a simplified API to the service layer below, and secondly a way to access to the FPGA cores and other hardware peripherals through standard drivers.

3.1.1.3.1 *FPGA (IP) drivers*

As mentioned in the introduction of section 3, native access to the FPGA is a fundamental feature of the MIA platform and *SSLA* layer, given its increasing usage in flight software and time-critical applications. This way, the

mission developer or flight software architect can easily allocate functionalities either to the FPGA or to the embedded C software applications. In general, repeatable mechanical operations with a strict time deadline are better handled in the FPGA. On the other hand, complex sequential processes or functions that depend on changeable data from external equipment are more easily coded in C. Every functionality that is allocated in the FPGA leads to a lighter less-constrained software with less CPU and memory load. FPGA-implemented functions are also faster than their software equivalents.

Two types of drivers are currently supported within the *SSLA Lib* to access and exchange data with the FPGA cores implemented in the programmable logic part of the hardware MPSoC. The first one is dedicated to exposing individual AXI-bus-addressed registers from every FPGA core to the software executing in the CPU processors, so that their values can be read or written. The second driver type is dedicated to configuring a specific DMA engine core, implemented in the FPGA, so that high-rate data sources from the programmable logic can directly write their data to the shared volatile memory of the MPSoC without CPU or software intervention.

[Access to individual registers addressed in the AXI bus](#)

This native driver framework is dedicated to FPGA designs in which cores are connected and accessed through the Advanced eXtensible Interface (AXI) bus. This is an on-chip communication protocol part of the Advanced Microcontroller Bus Architecture (AMBA) specification. In MPSoC systems that use this protocol (e.g. ARM architectures), the AXI bus is the common data bus connecting the programmable logic (FPGA) and the processing system (CPUs) where software is executed.

For AXI-compliant FPGA designs, each core may declare any number of registers whose value can be accessed at a specific AXI bus address. This way, reading or writing the value of an FPGA register from the software consists of an AXI bus transaction controlled with a specific handshaking using some control signals. In the transaction the accessing software driver first has to specify the address of the register in a dedicated address channel. Then for write accesses from software to FPGA, the software specifies the value to be written in a data channel. Oppositely the data channel is populated by the FPGA core with the requested value, in read accesses from software to FPGA.

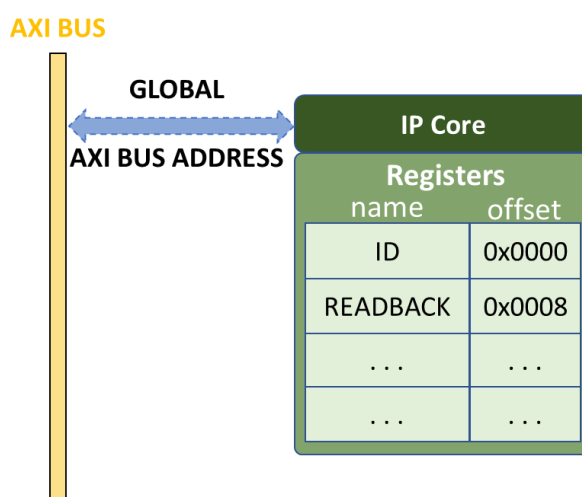


Figure 3-3: *SSLA native access to FPGA core registers*

For each FPGA core, *SSLA Lib* assumes that at the 'base address' of the core an ID register exists. Then at a subsequent address a readback register is set where the driver can be used to test the correct functioning of the AXI access (a value written to that register is then read to check the proper working of both the hardware and the driver). Then individual registers for mission-specific functionality exist, up to the FPGA developer and flight software architect decisions. This access is represented in Figure 3-3. Since the AXI bus register access involves a

complex transaction and specially to decouple FPGA workflow from software development, the actual coding of this read/write AXI register drivers is done automatically by means of a script as explained in section 3.2.1.

Management of Direct Access Memory (DMA) transfers between the FPGA and the volatile memory

Very frequently in space and avionics applications, it is necessary to capture data from external sources (e.g. sensors or secondary processors) at a very high rate. As described above, having the FPGA handle the low-level acquisition of these data leads to a lighter software which is also less coupled to external equipment and manufacturers. When these data sources have to be read at a high rate, the previously mentioned individual register access becomes inefficient and, in some cases, inadequate. To solve this, the FPGA design can make use of the DMA engine provided by most processors, that allows unattended access to the volatile memory without CPU intervention. Manufacturers of MPSoC and FPGAs provide specific IP cores dedicated to handling this DMA access.

The current version of SSLA supports the configuration and management of a specific AXI-DMA IP core provided by an FPGA manufacturer through integrated drivers in the *SSLA Lib*. These drivers provide a method for setting up the FPGA DMA core for unattended transfers in both directions: from the FPGA to the volatile memory or from the volatile memory to the DMA. More specifically, the drivers support the configuration of DMA transfers in two modalities:

- Direct register mode: this method requires the software application to ask the FPGA DMA core to initiate each individual data transaction. Even if the CPU has to actively request each transaction, the data transfer itself then occurs unattended, so this scheme still reduces the CPU load of the processor for external data acquisition/sending.
- Scatter/gather support: this advanced DMA mode further offloads the CPU of DMA management. In this case, the driver just configures the DMA core at initialization for the expected frequency and length of the transactions as well as the destination/source address of the data in memory. Once this is set up, all the DMA transactions (data and control handshaking) occur unattended without CPU intervention.

3.1.1.3.2 Ground command driver

A driver component exists in *SSLA Lib* that provides an abstracted use of ground command reception and sending to applications. In the context of SSLA, ground commands refer to any TC message that arrives to the MIA platform from any external agent and is ultimately made available to any of its running applications. It does not have to actually come from a ground segment in the specific sense, it can come from any other equipment or unit in the spacecraft. This terminology has been inspired by the cFE framework, but it has been extended in the case of SSLA to also include any TM message sent from MIA to the outside of the unit (which would be called ground telemetry).

The incoming ground command support provided by cFE is very debug/test oriented, and it is directly coupled to reception through a networked UDP/IP connection. In cFE ground commands are received by a specific application, that implements the UDP reception and then forwards incoming messages to the internal software bus. This configuration is rigid and requires a physical processing platform with traditional networking capabilities, which is not always available in every space/avionics scenario. In resource-constrained devices and equipment, lighter communication protocols such as traditional serial buses (CAN, SPI, I2C, UART) or avionics-oriented protocols (SpaceWire, 1553) are much more common in our experience.

The *SSLA Lib* includes its own ground command support component to overcome these limitations, providing a layered physical/logical set of driver components. This way, the actual physical bus or communication device through which the data is sent/received in the platform is not known at the upper driver layers. This results in enhanced portability and support for a greater variety of communication methods. E.g., a ground command may be received at the platform from a UART port or from a dedicated RF transceiver and this would be unknown at the logical level of the ground command driver, and ultimately will be seen by the individual application(s) without any distinguishable difference.

For explanation purposes, Figure 3-4 shows the particular configurations of the ground command driver set when the physical layer consists of a UART port (left), and when it consists of a CAN bus connection (right).

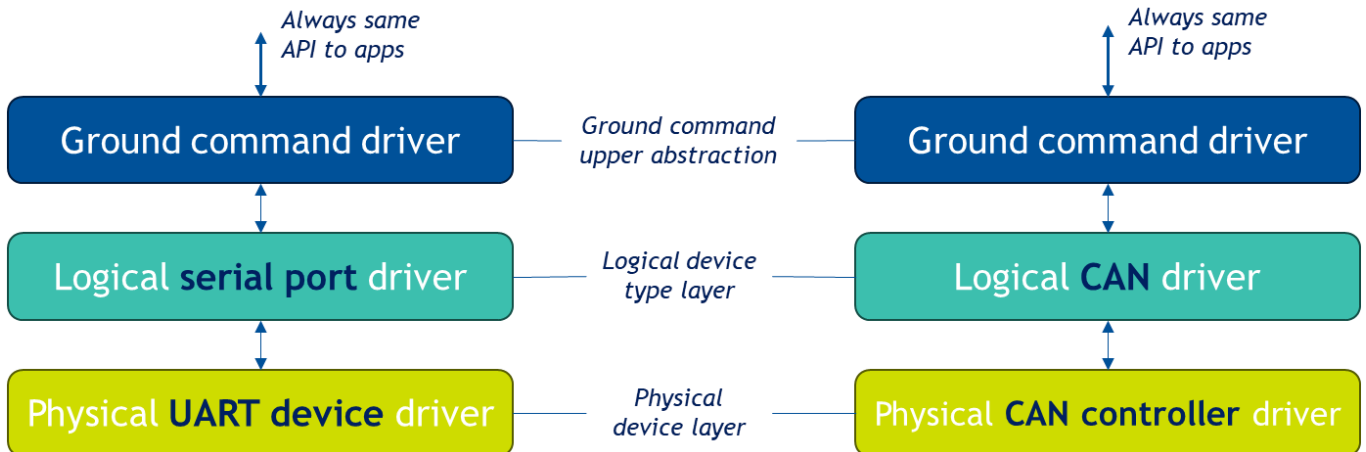


Figure 3-4: Layered structure of the ground command driver set

From the bottom-up, each of the layers provides abstraction to the one on top, and has the following purpose:

- Physical device layer: this driver implements the specific characteristics of the hardware peripheral in charge of communications. For instance, in case of an on-chip UART controller peripheral, this is the driver that would set the relevant values in the peripheral configuration registers to: enable the port, set its clock line, configure its baudrate, set the format of the data, determine which in/out pins it uses, setup interrupts, etc. As expected, this layer is completely coupled with the processor and manufacturer.
- Logical device type layer: this driver provides an enhanced abstraction on top of the previous one, absorbing the manufacturer-specific characteristics and presenting them in a generic way to the upper layer. Following with the UART example, in this case the logical device-type would be a serial port, which would have generic functions to configure settings present in all serial ports, e.g. data character length, number of stop bits, parity and baud rate, and then read and write functions.
- Ground command upper abstraction: finally, the ground command driver provides the high-level receive/send functions for TM/TC that applications can directly use. At *SSLA Lib* initialization, the three layers are initialized with their proper values and configurations. From this point on, any reception/sending of a ground command/telemetry is done using the complete driver set.

In addition, to provide a seamless integration with the software bus and the ground commanding scheme present in cFE, a couple of SSLA applications are optionally provided: a command ingest app (*ssla_gnd_ci_app*) and a telemetry output app (*ssla_gnd_to_app*). The first of these applications is in charge of polling the external ground command driver interface for incoming TCs, and then forwarding them to the software bus. This way, subscribed applications (which have registered this specific ground command ID in their Setup function) can receive them in a straightforward way from the software bus without having to even interact with the ground command driver. The second application provides the opposite functionality by polling the software bus for outgoing messages that are directed to the outside of the unit, and then relaying them using the ground command driver set. As can be seen, these optional applications absorb most of the external interfacing of the unit, so that the rest of the apps can be more decoupled of the characteristics of each mission scenario. However, since the *SSLA Lib* is accessible by all *SSLA Apps*, individual managing of the ground command driver by any specific app is permitted.

3.1.1.4 SSLA Software Bus Usage

SSLA makes an intensive use of the underlying cFE Software Bus module and, in its current form, is heavily coupled to its existence. The SB is probably the most important service in the cFE layer because it ensures the portability of cFE-compliant applications between different RTOS and processors. It's also the main responsible for the enhanced

application reusability and modularity achieved with SSLA and MIA. This is because it abstracts the apps from the way the actual information is shared in the hardware. It also provides a standard way for implementing inter-application information exchange which fosters the creation of hyper-specialized applications with a very clear and reduced objective.

An example of this could be a specific app for decoding incoming telecommands of a particular kind or source. In the SSLA approach, this specialized app would subscribe to the arrival of these TCs through the SB (which in turn may have been actually read from an external bus by another app). Once received and decoded, the app would publish the relevant parsed data contained in another custom message to the SB, which would be further read by any application that needs it. This approach allows SSLA apps to be very decoupled from the mission-specific behavior and, consequently, to be easily reused from mission to mission.

When using the SB, SSLA classifies exchanged messages in two different types: command or data. As explained in the above subsections, these two different messages are stored in two different pipes as depicted in Figure 3-5. The SSLA framework makes sure that depending on the message ID, this classification is done properly. The *SSLA Core* component also guarantees that the reception of a command message is immediately followed by the execution of a specific registered function, while the data pipe must be polled by the application asynchronously.

With this approach, the complete mission behavior can be translated into apps and inter-app messages in a straightforward manner. Command messages may be used to strictly determine the order of execution of different apps to achieve a desired sequential behavior, by having each app subscribed to a command from the previous app in the sequence, thus making sure that an app is not executed until its required inputs are ready and updated. Data messages are used for any other behavior that has to be done asynchronously or with a lower priority (e.g. regularly polling a sensor value which may be an input for any other function in the system, but does not need to be read at every execution cycle). Considering this scheme, the assignment and definition of message IDs and payloads has to be done in a centralized way by the mission developer when defining the complete SSLA applications architecture.

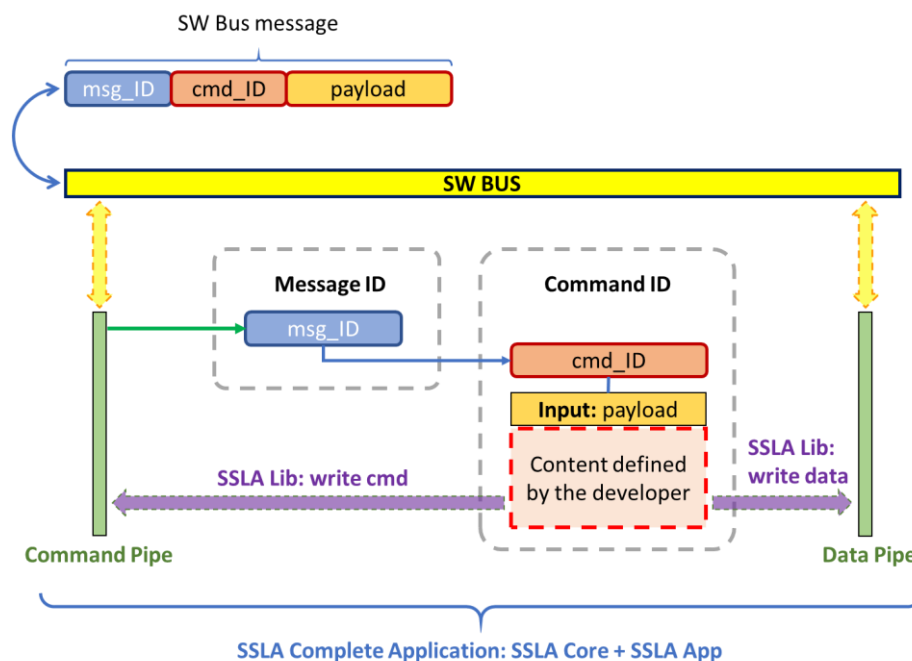


Figure 3-5: SSLA Software Bus usage

3.2 SSLA Software Development Kit

The abovementioned standardized structure allows us to provide an SDK for SSLA application developers, aimed at expediting and facilitating their workflow. Two semi-automated tools of this SDK stand out for their usefulness during development.

3.2.1 Automated FPGA driver generation

A script-based tool has been generated with the main objective of isolating the workflow of the FPGA and SW teams in a given project. The tool parses the source files of a complete FPGA project (e.g. the HDL files, TCL scripts and block descriptions) but not the actual implemented bitstream. With this, the script automatically learns what FPGA registers are exposed through the AXI bus, and in what addresses. Then, it generates the corresponding drivers into the *SSLA Lib* module, to read and write from the discovered registers, as well as a set of explanatory documents (see Figure 3-6).

This is very useful during development time for the SW team, since it avoids having to have an active and continuous interaction with the FPGA team about possible changes in their project's structure and components. It is also compatible with any kind of automated continuous integration scheme.

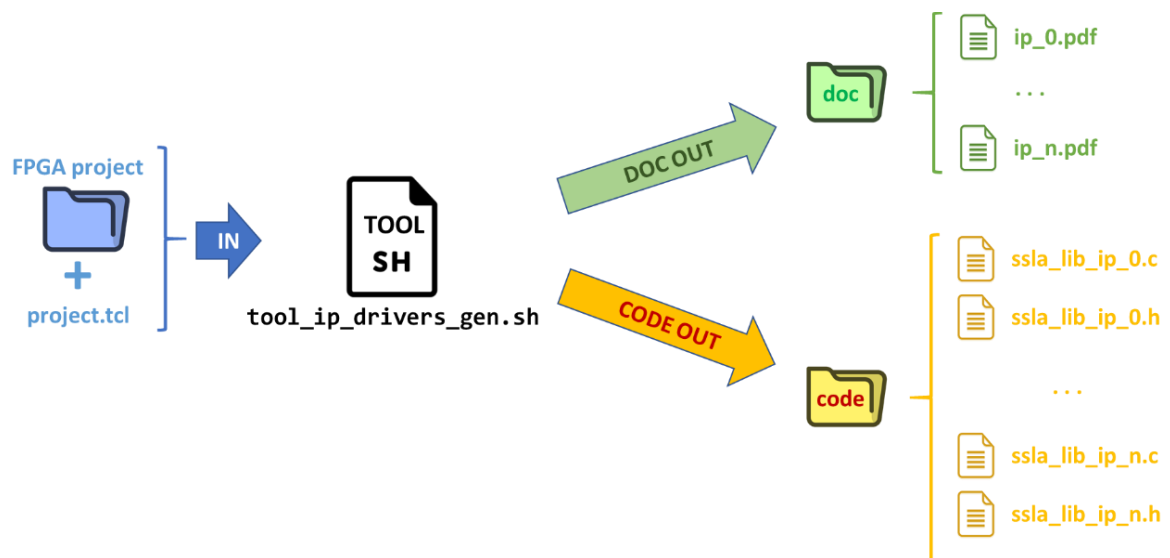


Figure 3-6: SSLA Automatic FPGA driver generation

3.2.2 Application generator

The screenshot shows the 'Create SSLA Application' tool interface. At the top, there's a title bar with several tabs: 'pre.c', 'cfe_sb_apl.c M', 'cfe_sb_lib.c', 'Extension: SENER Service Layer API', 'SSLA Create Application X', 'cfe_sb_util.c M', 'cfe_sb.h M', 'sslalib_core_msg.h', 'test_test0.sh', and 'cmdUtil'. The main window has a dark header with the title 'Create SSLA Application'. Below the header, there are several input fields: 'SSLA Application name' (containing 'Navigation_HPT'), 'Author' (containing 'SENER Aeroespacial'), 'Performance Id' (containing '110'), 'MID for GND Cmds (hex)' (containing '0x1882'), 'MID HK Req (hex)' (containing '0x1883'), and 'MID HK TIm (hex)' (containing '0x0883'). At the bottom, there is a 'Create SSLA Application' button.

Figure 3-7: SSLA application generator tool

Another useful tool contained in the SDK is related to automate the generation of SSLA applications, to take advantage of their abovementioned standard structure. The tool is based on having a template or skeleton for SSLA Apps in which the standard structures are pre-written with default values. The application developer, through a graphical menu (Figure 3-7), introduces the actual mission values (e.g. name and ID of some message for the application to subscribe to). The tool then fills these mission-related values into the app template and adds it to the project's code structure, as well as modifying the compilation flow accordingly so that the app is built and linked into the final executable.

4 QUANTITATIVE ANALYSIS AND METRICS

4.1 SSLA source files

The different services and components of SSLA presented in the previous chapter are actually coded and structured in several header and source code files, written in the C language. Figure 4-1 shows the files corresponding to each *SSLA App*, where as depicted, each different functionality is written in its specific source file. It is important to remark that the SSLA structure and SDK already provides pre-filled templates for all these source files. Thus, a mission developer only has to add and modify a very low percentage of the code to implement its desired functionalities.

For instance, if an app is required to perform a mathematical calculation upon the reception of a specific inter-application command, then its developer will have to provide the format and payload of that command as well as the function for the needed calculation in the `ssla_app_rxMidCmds.c/.h` files. The message ID for this command message will have to be declared in the `ssla_app_msgids.h` header. Finally, the Setup function of `ssla_app.c` will have to be updated with the corresponding register function for that inter-application command message. Anything apart from this is already contained either in the *SSLA Core* and *SSLA Lib* components code, or is already provided in the *SSLA App* template. This results in a drastic reduction of the time needed to deploy and set up each individual application, which can be further reduced making use of the above automated application generator.

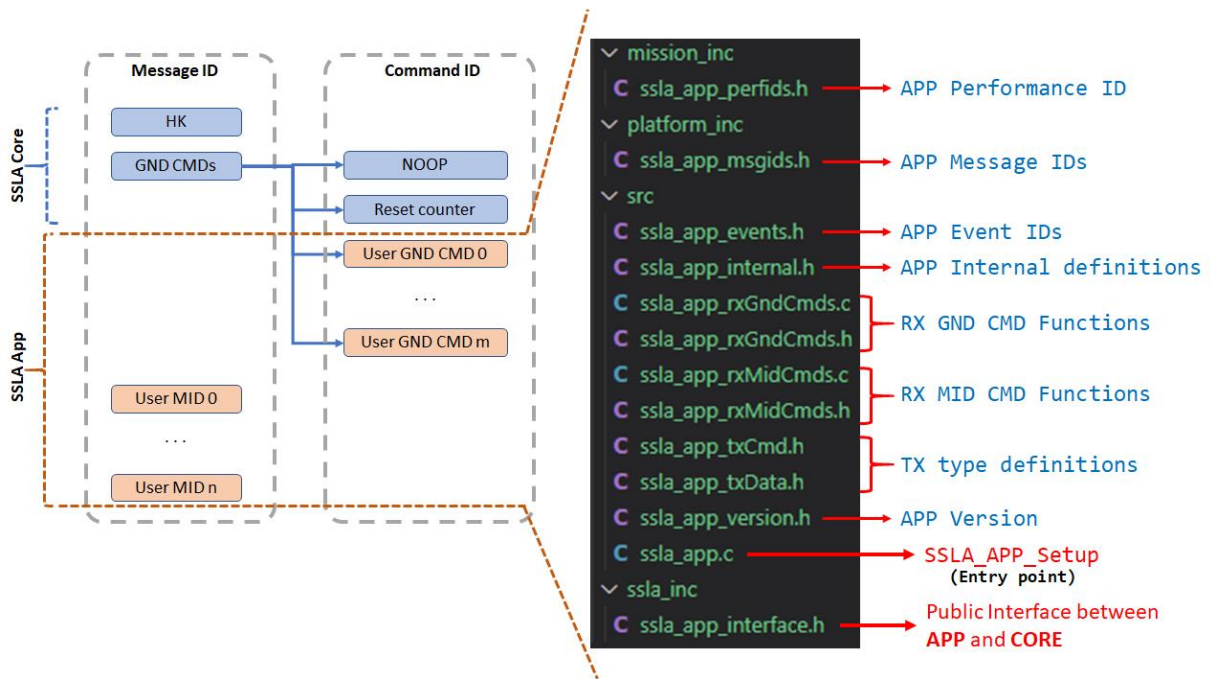


Figure 4-1: SSLA source code structure

4.1.1 SSLA code metrics

In a given project, the complete SSLA code base is composed by the *SSLA Core* and *SSLA Lib* modules, and as many *SSLA Apps* as the mission requires. To have an overview of their size and complexity, Table 4-1 shows their values in terms of lines of code, with comments or blank lines being removed.

Table 4-1: Files and lines of code of SSLA components

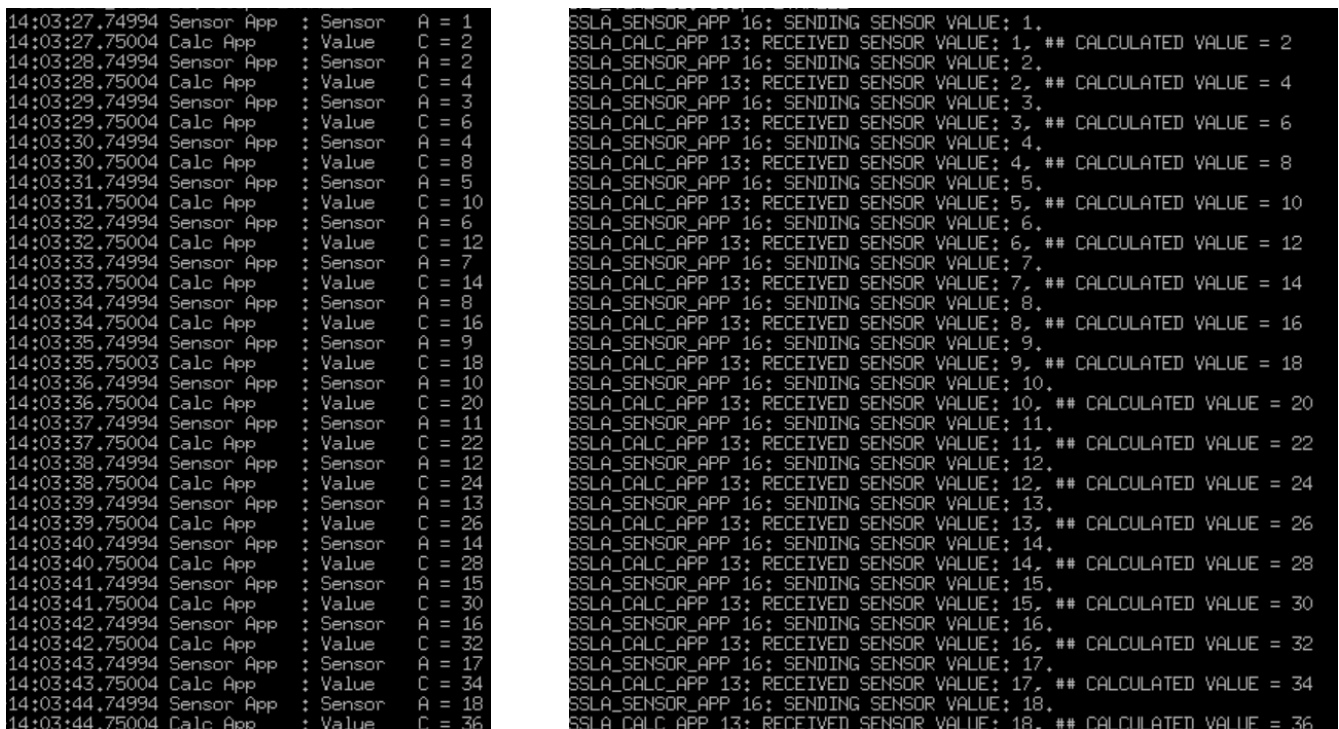
Component	Files	Lines of Code
<i>SSLA Core</i>	6	571
<i>SSLA Lib</i>	9	760
<i>SSLA App (base template with no included functionality)</i>	16	216
TOTAL	31	1547

4.2 Example mission scenario

An example mission scenario has been selected to perform a quantitative analysis of the effects and overhead introduced by the SSLA layer. This scenario is a simple but representative one in which two applications co-exist with the following behavior:

- **SENSOR Application:** this app reads data from an emulated sensor at 1 Hz. The value from the sensor is emulated to be incrementing by 1 at each reading. Once the sensor is read, the application sends its value through the software bus in a predefined message.
- **CALCULATOR Application:** this application is subscribed to the message from the sensor app. Upon receiving this message, it extracts the sensor value from the message's payload and multiplies it by two. This *calculated* value is then reported through the standard output.

This example scenario applications have been implemented in two distinct manners in the MIA execution platform. In the first way, the apps are implemented directly on top of cFE, i.e. without using SSLA. The second manner uses the SSLA layer and components, implementing the scenario apps as two *SSLA applications*. Examining the results of both implementations allows us to have a quantitative measure of the effects and overheads introduced by our proposed SSLA layer. Figure 4-2 shows the standard output of the MIA platform during the execution of both approaches, to demonstrate their equivalent behavior.



```
14:03:27.74994 Sensor App : Sensor A = 1
14:03:27.75004 Calc App : Value C = 2
14:03:28.74994 Sensor App : Sensor A = 2
14:03:28.75004 Calc App : Value C = 4
14:03:29.74994 Sensor App : Sensor A = 3
14:03:29.75004 Calc App : Value C = 6
14:03:30.74994 Sensor App : Sensor A = 4
14:03:30.75004 Calc App : Value C = 8
14:03:31.74994 Sensor App : Sensor A = 5
14:03:31.75004 Calc App : Value C = 10
14:03:32.74994 Sensor App : Sensor A = 6
14:03:32.75004 Calc App : Value C = 12
14:03:33.74994 Sensor App : Sensor A = 7
14:03:33.75004 Calc App : Value C = 14
14:03:34.74994 Sensor App : Sensor A = 8
14:03:34.75004 Calc App : Value C = 16
14:03:35.74994 Sensor App : Sensor A = 9
14:03:35.75003 Calc App : Value C = 18
14:03:36.74994 Sensor App : Sensor A = 10
14:03:36.75004 Calc App : Value C = 20
14:03:37.74994 Sensor App : Sensor A = 11
14:03:37.75004 Calc App : Value C = 22
14:03:38.74994 Sensor App : Sensor A = 12
14:03:38.75004 Calc App : Value C = 24
14:03:39.74994 Sensor App : Sensor A = 13
14:03:39.75004 Calc App : Value C = 26
14:03:40.74994 Sensor App : Sensor A = 14
14:03:40.75004 Calc App : Value C = 28
14:03:41.74994 Sensor App : Sensor A = 15
14:03:41.75004 Calc App : Value C = 30
14:03:42.74994 Sensor App : Sensor A = 16
14:03:42.75004 Calc App : Value C = 32
14:03:43.74994 Sensor App : Sensor A = 17
14:03:43.75004 Calc App : Value C = 34
14:03:44.74994 Sensor App : Sensor A = 18
14:03:44.75004 Calc App : Value C = 36

SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 1,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 1, ## CALCULATED VALUE = 2
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 2,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 2, ## CALCULATED VALUE = 4
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 3,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 3, ## CALCULATED VALUE = 6
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 4,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 4, ## CALCULATED VALUE = 8
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 5,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 5, ## CALCULATED VALUE = 10
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 6,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 6, ## CALCULATED VALUE = 12
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 7,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 7, ## CALCULATED VALUE = 14
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 8,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 8, ## CALCULATED VALUE = 16
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 9,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 9, ## CALCULATED VALUE = 18
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 10,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 10, ## CALCULATED VALUE = 20
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 11,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 11, ## CALCULATED VALUE = 22
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 12,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 12, ## CALCULATED VALUE = 24
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 13,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 13, ## CALCULATED VALUE = 26
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 14,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 14, ## CALCULATED VALUE = 28
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 15,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 15, ## CALCULATED VALUE = 30
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 16,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 16, ## CALCULATED VALUE = 32
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 17,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 17, ## CALCULATED VALUE = 34
SSLA_SENSOR_APP 16: SENDING SENSOR VALUE: 18,
SSLA_CALC_APP 13: RECEIVED SENSOR VALUE: 18, ## CALCULATED VALUE = 36
```

Figure 4-2: On target execution traces of the two implementations of the example scenario: without SSLA (left), using SSLA (right)

4.2.1 Results and discussion

The main advantage of the SSLA layer has to do with its standardized execution flow and its provided code template for applications, based on the *SSLA Core* module and the *SSLA App* interface. When using SSLA, the total number of lines of code of the example scenario implementation is higher than in the case of not using SSLA (because of the inclusion of all files described in Table 4-1). However, from this total number of lines, the ones that are actually required to fulfill the mission implementation (functionality-specific code) are 133, which is about 7% of the complete code of the mission. Since the SSLA code is already provided to mission application developers, and can be used as is without modification, this means that for the given example scenario, the developer would have to

write a much more reduced set of code to implement its required functionality compared to the case in which SSLA was not used. This result can be seen in Table 4-2 and Figure 4-3, where the orange part of the columns corresponds to the functionality-specific code under comparison.

Table 4-2: Comparison of lines of code required in both implementations of the example mission scenario

Scenario	Base code (SSLA Core + Lib + App templates)	Functionality-specific code	TOTAL
Without SSLA	-	570	570
Using SSLA	1763	133	1876

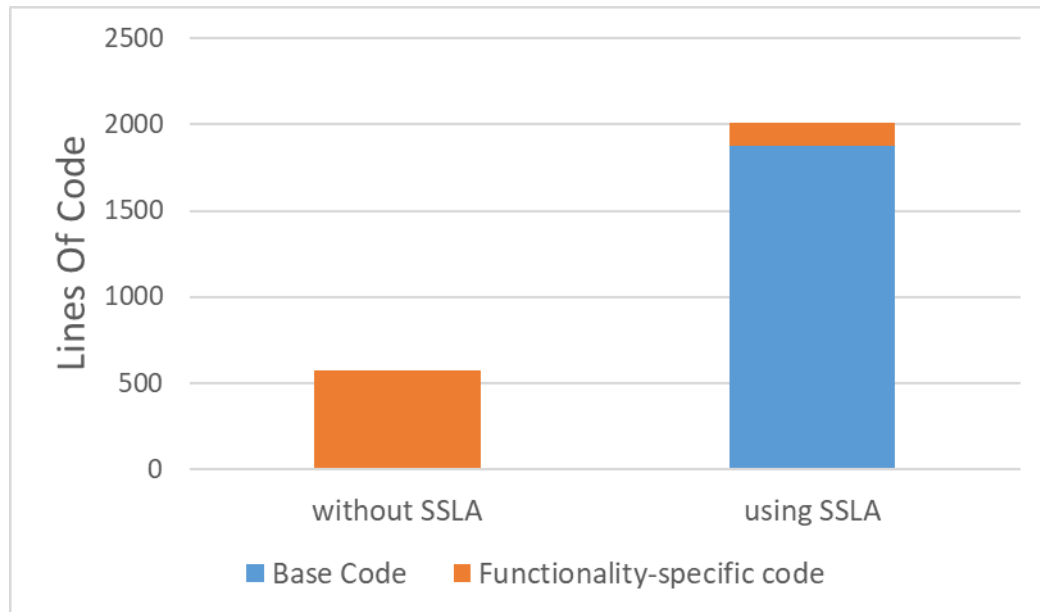


Figure 4-3: Comparison of lines of code required in both implementations of the example mission scenario

The footprint of the different program segments on the target's memory is also different in the two implementations under review. Table 4-3 shows the most relevant segments and their variation from the case of not using SSLA to the case of using it. For reference, the `.text` section refers to the compiled code in the form of instructions. The `.data` corresponds to the area where initialized global and static variables are stored. Finally, the `.bss` segment stores uninitialized global and static variables and structures.

Table 4-3: Comparison of memory footprint in bytes in both implementations of the example mission scenario

Section	Without SSLA	Using SSLA	Variation (abs)	Variation (%)
<code>.text</code>	109148 B	105188 B	-3960 B	-3.63%
<code>.data</code>	1908 B	1900 B	-8 B	-0.42%
<code>.bss</code>	1274172 B	1275964 B	+ 1792 B	+ 0.14%

As can be seen, the memory footprint of the code section is reduced in the case of using the SSLA layer. This is because even if the total lines of code of all the source files are higher when the SSLA layer is included, the code that ends up being used (and thus compiled and linked into the executable file) in this particular mission scenario implementation is much less than the total code base. This is generally true for any other mission or implementation in the sense that the complete SSLA code base is provided as a full component of which particular mission implementations just use a part. For instance, the complete set of *SSLA Lib* functions is not meant to be used in full in most missions. Additionally, even if a separate instance of the *SSLA Core* has to be associated with each

implemented *SSLA App*, their code is exactly the same, so the instructions stored in the `.text` section are not incremented with every included application.

In case of the `.data` and `.bss` sections, the difference between both approaches (with or without using SSLA) is arguably negligible. The `.bss` section is slightly increased in the case of including the SSLA layer due to the number of message and register data structures that the *SSLA Core* component introduces.

The complete size of the executable binary file produced for this example mission scenario is 2.406 MB in the case of not using SSLA and 2.399 MB in case of using the SSLA layer, an insignificant difference. Both binary files include the complete MIA software layers, containing the RTEMS kernel as well as the OS for this specific mission scenario.

In summary, the conclusion that can be extracted from this quantitative analysis is that the SSLA layer has a negligible impact in terms of the memory and performance of the resulting program, but a significant effect in reducing the development, deployment and testing time of mission software.

5 CONCLUSIONS

The presented work introduces SSLA, an enhanced abstraction and framework for the development of flight software developed at SENER Aeroespacial. In its current version, it works on top of the well-known NASA's cFS architecture, keeping its features but simplifying its exposed interface and providing a standard structure for the execution flow of applications. SSLA also provides a set of new services not present in the reviewed flight architectures, especially the native support for FPGA core access from the software applications.

In summary, the proposed SSLA architecture improves the standardization and modularity of mission applications, greatly increasing their reusability, which ultimately reduces the cost and time of software development in space and avionics projects.

ACKNOWLEDGEMENTS

This work is carried out in the frame of the SAFEST Project in which SENER Aeroespacial is the coordinator of a joint effort of many individuals and organizations. This project has received funding from the European Union's Horizon Europe research and innovation programmed under grant agreement No 101082662. We greatly thank of the great work of all the consortium members, the European Commission officers and reviews and the business development department from SENER Aeroespacial, which have believed in and firmly committed to the project.

The preliminary design and architecture of SSLA was supported by a prior R&D project called MFOC (Madrid-Flight-On-Chip). It was publicly funded by Comunidad de Madrid and the European Union, under contract No 49/520608.9/18. The authors want to thank Santiago Lozano and Carlos Rodríguez who participated in the design and implementation of this preliminary version of SSLA.

REFERENCES

- [1] D. C. McComas, "Increasing Flight Software Reuse with OpenSatKit," Nasa.gov, Mar. 03, 2018. <https://ntrs.nasa.gov/citations/20180001888> (accessed Feb. 07, 2024).
- [2] German Aerospace Center (DLR), "Open modular software Platform for Spacecraft (OUTPOST)." <https://github.com/DLR-RY/outpost-core> (accessed Feb. 07, 2024).
- [3] F. Dannemann and F. Greif, "Software Platform of the DLR Compact Satellite Series," Proceedings of 4S Symposium 2014. 4S Symposium, 26-30 May 2014, Mallorca, Spain.
- [4] J. Galizzi, P. Arberet, J. Damery, C. Guy, A. Crespo, M. Masmano, F. Roubert, "LVCUGEN – Ready for Flight?" Proc. 'DASIA 2015', DATA Systems In Aerospace, Barcelona, Spain, 19–21 May 2015 (ESA SP-732, September 2015).

- [5] S. Lozano, J. Fombellida, C. Rodríguez, C. Tato, J. Carretero, "MFOC Project: MPSoC-Based Multi-Purpose Execution Platform", III Congreso de Ingeniería Espacial: El espacio, la última frontera, Madrid, Spain, 2020, 27-29 October. ISBN: 978-84-09-31948-0. pp. 120-122.
- [6] E. Geist, "Core Flight System Training - cFS Draco," Nasa.gov, Jan. 19, 2024. <https://ntrs.nasa.gov/citations/20240000217> (accessed Feb. 07, 2024).
- [7] M. Masmano, I. Ripoll, A. Crespo, J. Metge, "Xtratum: a hypervisor for safety critical embedded systems." In 11th Real-Time Linux Workshop, vol. 9, September 2009.
- [8] "RTEMS Qualifies for the Space Domain | RTEMS Real Time Operating System (RTOS)," Rtems.org, 2022. <https://www.rtems.org/node/139> (accessed Feb. 07, 2024).