AN I/O BUILDING BLOCK FOR THE IMA SPACE REFERENCE ARCHITECTURE

Cláudio Silva, João Cristóvão, and Tobias Schoofs

GMV, Portugal, Av. D. João II, Lote 1.17.02, Torre Fernão Magalhães – 7º, 1998-025 Lisboa, Portugal – Email: claudio.silva@gmv.com, joao.cristovao@gmv.com, tobias.schoofs@gmv.com

ABSTRACT

The IMA for Space project (IMA-SP), led by Astrium, defines a reference architecture for space on-board software based on the concept of Integrated Modular Avionics (IMA). This platform encompasses not only low-level software like operating systems and communication middleware, but also domain-specific services, such as FDIR, mode control or on-board software maintenance. Part of the platform is a generic I/O solution that provides services based on network interfaces and other devices. In compliance with the overall objectives of the platform, the I/O component should not only integrate device control into a partitioned environment, but shall also relieve applications from the burden of addressing I/O directly. Applications built for the IMA platform are in consequence reusable across different hardware platforms and different missions.

Key words: IMA; TSP; ARINC 653; AIR; Operating Systems; I/O; IMA FOR SPACE; MIL-STD-1553B.

1. INTRODUCTION

Modern computer platforms used in desktop computing, mobile phones and large server systems encompass sets of standard I/O devices and the driver software that hides the individual characteristics of these devices from applications. Applications for this kind of platforms do not need to deal with peculiarities of I/O, but instead use standard APIs, such as POSIX, or APIs that are part of the language environment (*e.g.* Java). Applications also accept and rely on the policies implemented by the OS used on the platform, which may be optimised for resource usage, parallel computing or real-time.

For embedded systems and, in particular, for embedded real-time systems, such as on-board systems in space, the situation is quite different. There are, on one hand, devices for very specific activities, *e.g.* star trackers, gyroscopes or thermal and battery control systems, and on the other hand, the computing platforms are less standardised than in general purpose computing. I/O is therefore an aspect that has to be taken into account in the application code. Since applications in this domain, typically, have to obey hard real-time requirements, the use of devices has to fit into frequencies and time budgets defined on the level of individual tasks, but orchestrated on system level. I/O related code, in consequence, is often rewritten for individual missions even if the purpose of the applications is very similar to previous use cases.

Much effort is being spent by the space community to overcome this kind of issues and to standardise the use of system resources. The industry, today, is on its way to define a standard architecture based on reusable components [7, 12].

Part of this effort is the definition of a platform based on time and space partitioning, inspired by Integrated Modular Avionics (IMA) as it was introduced in the aeronautics industry during the last twenty years [13]. The IMA-SP project defines a standard IMA platform specific for the space domain. The platform does not only provide services on operating system level, but includes domain-specific services, such as FDIR, mode control, on-board software maintenance and a generic I/O solution.

In this paper, we will present the I/O component, which is being developed by GMV in the scope of IMA-SP. The paper is organised as follows: In section 2, we will describe the challenges to address in more detail and will discuss the IMA architecture which is the environment in which the issues have to be solved. In section 3, we will discuss the architecture of the I/O solution, including its integration into the system, its internal design and its configuration tool chain. In section 4, we will present the current state, some benchmark results, lessons learnt from demonstration activities as well as issues to be addressed by future work. Section 5 concludes the paper.

2. I/O, IMA AND THE UNIVERSE

The problems encountered with I/O in embedded real-time systems development are related to the diversity of devices for very specific purposes that need to be integrated, the limited standardisation of such devices and the need to obey hard real-time requirements on system level. In consequence, I/O-related aspects are usually still present in application code. Applications, in consequence, depend on specific hardware and cannot be

reused across different systems and missions.

In the aeronautics community, much has been achieved by the introduction of IMA [13]. With IMA, collections of equipment plugged together to a federated system have been replaced by a platform consisting of standardised components, namely central processing modules and an avionics data network (AFDX). Recent research projects have started to take this standardisation one step further [3, 5]. In addition to central processing modules, remote components like data concentrators and embedded control units, to which special purpose devices are connected, are integrated with the avionics network. This way, the avionics platform becomes a highly integrated system that can be designed, configured and programmed with a set of canonical tools. The final goal is that, in the future, avionics applications will be reusable among different configurations, including different types of aircrafts.

IMA, for these reasons, has been compared to the iPhone® [14]. Like the iPhone, IMA constitutes a platform for applications from multiple vendors that can be *easily* added or removed. Avionics systems, of course, are not as *easy* to use, not to mention to integrate, as the iPhone. The platform, however, hides details of specific hardware, such that applications can address standard services instead of proprietary component interfaces.

In the space community, these issues are addressed by efforts to define a reference architecture that strictly distinguishes mission-specific application building blocks (AOCS, thermal and power control, FDIR, *etc.*) and generic execution platform building blocks (PUS services, RTOS, OBCP interpreters *etc.*) [12]. The interfaces between these building blocks are well-defined and, hence, can be, to some extent, combined with each other to construct a system.

The approach of CCSDS recommendations on the Spacecraft Onboard Interface System (SOIS) [4] is to standardise the interfaces between items of spacecraft equipment by specifying standard service interfaces and protocols. This way, the access to sensors, actuators and other specific devices allows for spacecraft applications to be developed independently of the mechanisms that provide these services. Applications are thus insulated from the specifics of a particular spacecraft implementation and may be reused across missions.

SOIS presents a layered architecture to hide details of I/O devices from different levels and domains of processing. The lowest layer, the sub-networking layer, is aware of the underlying network architecture and provides services like memory access, time distribution, *etc.* The transfer layer, defines services for packet transfer such as TCP or UDP. The application support layer at the top of the stack provides higher level services directly to applications, *e.g.* command services, data acquisition services, time services, file access services and so on.

A reference architecture based on IMA should not fall back behind the reusability standards achieved – or promised – by concepts like SOIS. IMA, however, is not a layered architecture, but consists of horizontally integrated components, *i.e.* partitions. In the application support libraries, a layer is needed that bridges component interfaces from a vertical call hierarchy to a horizontal message passing mechanism. Additionally, the IMA architecture has to provide policies to assign time resources to the I/O processing. In a call chain, time consumed by I/O is naturally charged to the caller – calls are always synchronous. With I/O processing potentially sourced out to another partition, as described, for instance in [9], the communication may be asynchronous. This is not necessarily a disadvantage: the time the caller would otherwise spend on waiting for data can be used productively. In fact, it is not obvious in the SOIS approach, how resource usage can be optimised with standardised layers that treat all devices the same.

3. ARCHITECTURE

3.1. External Interfaces

As a reusable building block, the I/O component must provide a homogeneous interface to applications and an interface to communicate with the underlying operating system. The IMA-SP platform defines a set of standard interfaces between operating system kernel and applications; one of these interfaces is the space-tailored and qualified version of RTEMS [10], another is a subset of the ARINC 653 APEX [1] tailored to the needs of space applications. The I/O component uses RTEMS primitives for internal task management and ARINC 653 queuing ports to communicate with applications. There is additionally the option to use shared memory to optimise the handling of huge amounts of data when the underlying operating system supports such a feature.

From an application point of view, there are three possible interfaces to the I/O component; two of them rely in ARINC 653 queuing ports and the last on shared memory.

The first is a classical API, used to request services from the I/O partition. There is a set of basic calls modelled after the POSIX standard: READ[1], WRITE and IOCTL for reading and writing data to a device and, if this is allowed and appropriate in the given context, for performing device specific tasks like reconfigurations. Additionally, there is a SUBSCRIBE option that will cause the I/O partition to send all data, for instance coming from a given APID, to the subscriber application.

The second interface consists on 'simpler' remote ports, defined through configuration. This, in compliance with ARINC 653, is transparent to applications. A remote ARINC 653 port is not linked with another (regular) application partition, but instead with a local I/O partition; data written to this port by the application is automatically

[1]The READ service, in fact, is a bit more complex. It is split into two parts, the first one, requesting data, the second one, polling for data.

sent, by the I/O partition, to a network interface defined by configuration means. The data is read on the other side by a counter I/O component and written to an ARINC 653 port that, eventually, links with the final target application.

Finally, the shared memory interface is used to expose the device data buffers directly to a given partition, therefore reducing the number of memory copy operations required. This, however, comes at a cost of reduced interface independence.

All these interfaces use the logical device concept, where the application is allowed to address each distinct device in a transparent way, just through the logical device id or, in the case of the remote port option, the remote port id. The mapping between the logical identifier and a given address on a given bus is part of the I/O module configuration and therefore transparent to the application.

Beyond these low level communication mechanisms, the I/O component may also provide domain-specific services like access to on-board parameters and events; access to data stores (either via mapping of files to memory, dedicated hardware devices like packet terminals or through a full-fledged file system modelled after the ARINC 653 standard [2]) as well as time services and PUS services.

For dedicated devices, used by only one application in the system, the I/O component can be linked with this application; the drivers for the specific device are then local to the partition. The interface to the application, however, is still based on asynchronous message communication through ports. The I/O is, hence, flexible in terms of its location in the overall architecture and easily scalable to the number of client applications using its services.

For devices used by many applications, typically interfaces to MIL-STD-1553B sub-systems or SpaceWire networks, it is mandatory to isolate the I/O component in a partition. This way, devices are protected from direct – and potentially destructive – access by applications. The I/O partition, in this case, is responsible for sequentialising access to the device [9]. This is achieved by a set of sub-components to dispatch service calls and to route data. Figure 1 illustrates this design:

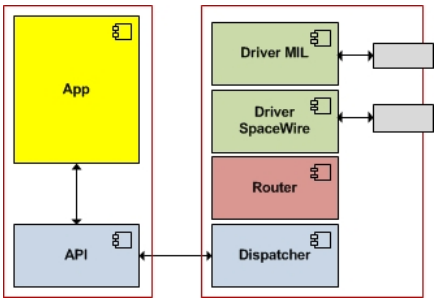

Figure 1: I/O Building Block

Different interfaces, of course, may be controlled by different I/O partitions; there may be, hence, more than one I/O partition in a system, each one potentially taking care of different aspects, *e.g.* serving different applications, providing different kinds of interfaces or acting at different assurance levels.

Isolated in a partition, the I/O component underlies temporal partitioning. This means that the component will be scheduled with a predefined frequency and for a predefined amount of time. It is the responsibility of the system integrator to provide a schedule that is able to guarantee the required data troughput per client application. Requirements for the I/O component are expressed in terms of amount of data to be read or written within a given period and a limited time budget. Note that the amount of data is directly mapped to time requirements: the data that can be exchanged with a network or other device is limited by the time available to the drivers. Tool support for configuring the component, in consequence, is based on assumptions about execution time including bus wait states, competing direct memory access (DMA) and execution time of drivers, APIs and OS-internal message transportation mechanisms. This is further discussed in subsection 3.5.

Although more efficient alternatives exist to an I/O partition, like co-partitions [8], that allow shorter latencies, this is not without a cost. A system that allows a partition to be preempted before the end of its execution window to perform I/O operations is more difficult to analyse, qualify, and in perspective, to fully guarantee as predictable.

Integrating the I/O module in the underlying operating system seems the most obvious alternative to relegating I/O to a dedicated partition. This integration would increase the size and complexity of the operating system kernel making it more difficult to qualify. It also decreases the portability of the solution, because the I/O module becomes dependent on a specific kernel.

3.2. Internal Design

Internally, the I/O component consists of a set of tasks that are periodically executed according to the partition frequency and execution time. Its internal schedule, *i.e.*, the periods and priorities assigned to the tasks, is devised in order to always achieve a full write and read cycle per period:

$$\text{DISPATCHER} \rightarrow \text{ROUTER} \rightarrow \text{DRIVER} \rightarrow \text{ROUTER} \rightarrow \text{DISPATCHER}$$

The dispatcher receives requests from applications and passes them to the router; the router determines the hardware interface and passes the request plus additional routing parameters to the corresponding driver which writes pending data to the device memory. This terminates the

WRITE cycle and corresponds to a half complete I/O cycle. The driver then starts the second half of the cycle by reading pending data from the device memory; the destination of arrived data is determined by the router, which passes the response to the dispatcher; the dispatcher, finally, delivers the data to the application. Note that passing of data, here, does not necessarily mean that data is physically copied in memory. On the contrary, data is left untouched whenever possible and copied once between the device memory and the application I/O data segment when shared memory is used, or to the port memory area, when ports are used.

The I/O component is optimised for periodic polling of memory devices. This way, it is fully deterministic in terms of data throughput and CPU usage per period. When serving many different interfaces for different applications, however, configuration can become very complex. In such cases, tool support will be needed to schedule the different drivers and the data to be delivered to different applications. This is further discussed in subsection 3.5.

Currently, the I/O component supports RS-232, UDP on Ethernet (the GRETH implementation on top of the AMBA bus), SpaceWire(GRSPW2) and MIL-STD-1553B. When the driver is based on DMA, which is the case for the most of the aforementioned devices, it is mainly busy with reading/writing data from and to the DMA memory area assigned to the particular device as depicted in Figure 2:

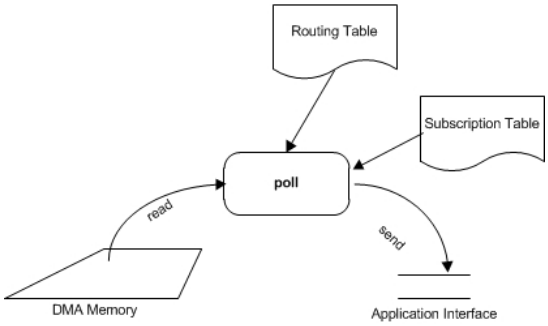

Figure 2: Typical Driver

3.3. The MIL-STD-1553B special case

A particular problem is presented by the MIL-STD-1553B-interface. Different implementations of MIL-STD-1553B for the LEON processor are available with significant differences in their configuration; in fact, the I/O component, currently, implements two distinct MIL-STD-1553B drivers targeting different devices.

It does not seem to make much sense to impose restrictions on the use of the MIL-STD-1553B interface by applications and, this way, reduce its flexibility. Instead it appears to be more promising to expose MIL-STD-1553B commands during design time. Device vendors, application developers and system integrators would then construct MIL-STD-1553B command chains as part of system development and integrate them into application configurations.

These command chains can vary considerably from one MIL-STD-1553B core to the other. In order to facilitate the adaptation of additional cores to the I/O component, an intermediate MIL-STD-1553B command list idiom was created, which is then translated into the specific core commands.

A MIL-STD-1553B bus controller operates by executing a predefined set of bus commands (command lists) during strict time windows. These commands are usually grouped in structures called minor frames, which are in their turn grouped into a major frame. A major frame acts as a fixed duration schedule of MIL-STD-1553B commands that is repeated in time with a given frequency. The MIL-STD-1553B command schedule is synchronized with the application scheduling so that data is timely available to the commands and applications being executed.

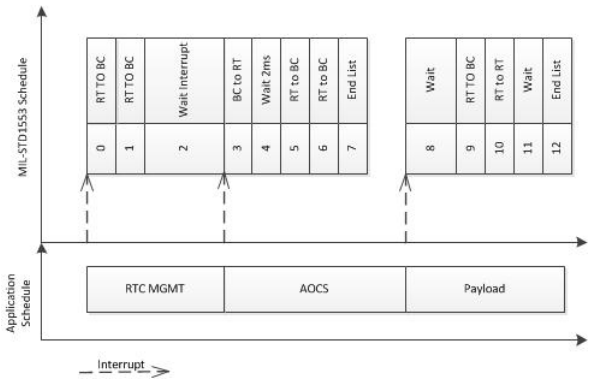

Figure 3: Example MIL-STD-1553B and application schedule synchronization

The synchronization between application and bus controller schedules poses a bigger challenge in a time partitioned operating system because the applications are bound to a two level schedule. This problem can become complex because both the partition schedule and the process schedule inside each partition must be adjusted to ensure synchronization with MIL-STD-1553B.

Even if the MIL-STD-1553B and application schedules are synchronized by master frames scheduling, a drift between the two can accumulate over time. To avoid this drift one possibility is to create synchronization points using a real time interrupt. Both schedules start simultaneously at each synchronization point, therefore ensuring that no drift persists. In satellite on board systems these interrupts are provided by a real time on board timer that ticks every MTF.

This synchronization with an external interrupt introduces a problem to time and space partitioned operating

systems since the partition scheduling is, according to the ARINC 653 standard, static.

3.4. Approaching the MIL-STD-1553B specificities

The MIL-STD-1553B bus controller command lists are statically defined as part of the I/O module configuration process. Multiple command lists can be configured; these lists can be switched during run time to provide support to different system operating modes. Depending on the underlying target device these commands can be not only MIL-STD-1553B bus commands but also (bus controller) control flow commands like branch or terminate list.

Each command on any given list features an unique *slot id* number. This *slot id* is used across the configuration and API to refer to this concrete command. The configuration also allows mapping a given *slot id*, and therefore its respective command, with a local inter-partition port. Data written to this port by the application will be written to the correspondent MIL-STD-1553B command. The *slot id* is also used to identify data buffers when the underlying MIL-STD-1553B device has shared memory support.

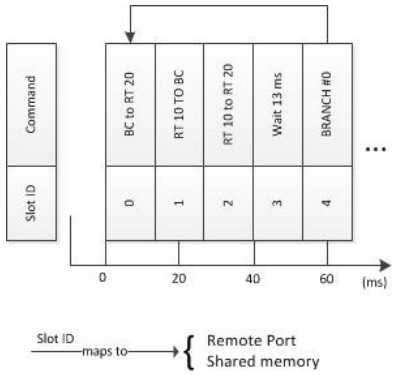

Figure 4: MIL-STD-1553B command list

If the used MIL-STD-1553B core has no specific memory constrains, thus being able to access the full memory address range, it is possible to use shared memory exposing the data buffers used in the MIL-STD-1553B commands to the applications.

In this case the I/O partition assumes only the role of managing the device state and not the data transfers. The mapping between a command and the partition that has access to it must be done at the configuration level.

All buffers belonging to a partition are bundled together and mapped to the partition address space. Since the data buffers are accessible by the application there is no requirement to run the I/O partition to feed data to the command list; this role is done directly by the application. This descreases the complexity of synchronizing the application and MIL-STD-1553B schedules.

A special user side API is used to manage these data buffers, ensuring that the correct data buffer for a given *slot id* is written or read. For MIL-STD-1553B devices that do not allow mapping the data buffers to the application address space, the I/O partition has to be responsible for handling data input/output on the command list.

Instead of writing directly to the data buffers, the application will send data to the I/O component (running in a dedicated partition), which writes it to the command list during its own execution windows. The synchronization between the bus and application is more complex because the I/O component must run each time a bus data access is required.

In this situation the I/O component must also track the completed commands and forward any incoming data to the respective partition. Each command may be accessed through the request/reply API, or by direct mapping, by configuration, to a sampling port.

Support for schedule synchronization using an external interrupt must be provided by the underlying OS. There are several ways to address this synchronization problem with support from a hypervisor.

As an example, when such feature is enabled in the AIR operating system, a special partition is configured in all partitioning schedules, always occupying the last execution window. This last partition execution window has a dynamic, but bounded, duration. The external synchronization interrupt defines the real execution time of the window. When raised, the interrupt handler associated with the external interrupt, is responsible for starting the next partitioning schedule MTF, and implicitly the MIL-STD-1553B command schedule. Hence the synchronization between these two schedules is ensured.

3.5. Configuration

The I/O component configuration can be split into two major aspects: routing and scheduling.

The first concerns the mapping between each logical device, used by the high level API or the remote port interface, and the actual physical device address (and subaddress, when relevant) in the appropriate bus.

This set of routing tables should be specified by the system integrator, thus freeing the application developer from the implementation details. The application developer needs only to be concerned with the logical device id or remote port, the data profile and scheduling given the partitioned environment.

The scheduling configuration is based on estimations of the worst case execution time for a whole cycle of reading/writing and delivering data. The input provided by the system integrator is the amount of data to be processed within a given time frame, most probably in the

form of a schedule of required transmissions on each relevant bus. The I/O component schedule configuration tool returns the amount of time that has to be assigned to the I/O component during this time frame. And this information can then be provided to the schedule design tools or schedulability analysers.

Currently, there is no modelling framework to support users in estimating the amount of data that needs to be moved arround in the system. There is also no general model as part of the tool chain that could provide a precise execution time prediction, even if such models exist for the LEON processor [6]. Integrated solutions may provide such tools in the future.

There are several other detailed options that need to be defined by the system integrator mainly related to the hardware devices used. Each one of the hardware cores supported has its own specific features that are exposed to the developer as configuration options. These options impact the data bus behavior, the memory consumption of the driver and data buffering capabilities among others.

4. CURRENT STATUS AND FUTURE WORK

4.1. Status

In the scope of the IMA-SP project, the following I/O subcomponents have been finalised:

- WRITE, READ, IOCTL and SUBSCRIBE interfaces;
- Remote ARINC 653 ports;
- The dispatcher/router and subscription environment;
- A RS-232 driver;
- A deterministic IP stack providing UDP on top of an Ethernet driver;
- SpaceWire driver;
- MIL-STD-1553B drivers for Bus Controller and Remote Terminal modes.
- MIL-STD-1553B core independent microcode to define MIL-STD-1553B commands.

These components have been extensively tested and are considered in *Beta Version*. More advanced features like high-level interfaces will be available as *Prototype* at the end of the project.

4.2. Use Case

The MIL-STD-1553B bus is ubiquitous in space embedded systems, but it represents a particular configuration problem, since it is not just a channel for data transportation, but a configurable system to control remote components.

The MIL-STD-1553B commands list master frames in space avionics systems generally have frequencies in the order of dozens of Hz, with several synchronization interrupts occuring in a master frame. Any realistic use case for the I/O component should be able to cope with timing requirements in this order of magnitude. The major issue with this data bus lies in schedule synchronizaion and not in data throughtput.

Whereas the MIL-STD-1553B bus is used for communicating with low bandwidth devices with strict timing constrains, the spacewire bus is generally used for data transfers between high bandwidth applications and devices with soft time requirements. A suitable use case for this data bus must test the data throughput of the IMA-SP module under stress conditions.

4.3. Benchmark Results

A example configuration was devised with two partitions, each communicating with a remote terminal on the MIL-STD-1553B bus. The I/O partition is executed during 2 ms at each 6 ms, thus with a frequency of 166 Hz. The first two miliseconds of the frame are occupied by an application that sends data to a remote terminal, while the last two miliseconds are used by an application that receives data from the same remote terminal.

The MIL-STD-1553B command master frame is composed by a RT to BC transfer occupying the first 4 ms and by a BC to RT transfer using the remaining 2ms. The synchronization between the application an MIL-STD-1553B schedules is depicted in the following figure:

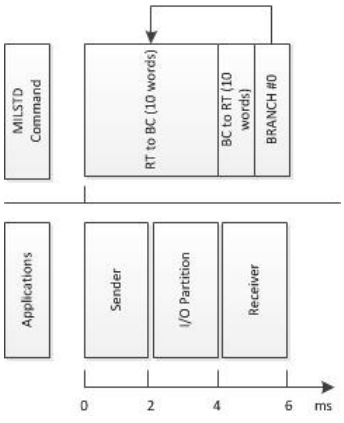

Figure 5: MIL-STD-1553B demonstrator

The MIL-STD-1553B schedule was built to ensure that the data is timely available to the applications and to the MIL-STD-1553B command. No external synchronization methods were used, therefore syncronization only relies

on the matching Master Frames. We were able to send and receive the MIL-STD-1553B messages on each task, with apparent synchronization.

To further test the command definition interface, the underlying MIL-STD-1553B core was switched and the example was re-ran successfully; Therefore demonstrating that the I/O partition's MIL-STD-1553B microcode is really core agnostic.

The Spacewire benchmark consists of a system composed by one application partition and one I/O partition. Each partition is allowed to run for 5 ms with a frequency of 100 Hz. The application partition tries to send 10 megabytes of data in loopback throught the Spacewire bus using messages of 1 kilobyte.

This test yielded a system data output of about 2 megabits per second. This value can be explained by the bottleneck generated by the memory copy operations in the interface between the two partitions. The usage of shared memory, bigger messages and a I/O partition optimized for Spacewire could easily increase this value.

During these benchmark and demonstration activities it became clear that the major problem related with the I/O module lies in the configuration. It is very hard for a developer to estimate the time and frequency required by the I/O partition to manage a given data profile. At this point and given the lack of model support, this adjustment has to be made in a trial and error process.

4.4. Future Work

Open issues are mainly related to matters of configuration and tool chain. A framework is needed to base estimations of data throughput generated by complex systems and worst case execution time on solid models. Even if models exist for the LEON processor [6], they have not yet been integrated with the tool chain. A promising way to go is a model-based tool chain integrating different desing aspects such as model of the processor and other hardware devices and system design models (based on *e.g.* AADL).

Another future development would be a framework that allowed building composite services based on simpler services provided by the I/O APIs. A Star Tracker device, for instance, would use the SUBSCRIBE and IOCTL primitives to build an API that provides services on a higher level, *e.g.* publish/subscribe of current state, request/reply of specific information and specific control requests. This Star Tracker service component is then in itself hosted on a partition (potentially together with other components) and provides domain-specific services as a specialised I/O component.

Alternative hardware architectures may even substitute this software-implemented I/O module completely or partly in the future. A smart DMA device, similar to an AFDX subsystem, could use discontinuous control and data areas directly assigned to partition address spaces such that each application has direct access to its own share of device memory. No additional CPU resources would be needed for I/O in such a configuration. Also, bus contention caused by concurrent access to memory by applications and the DMA controller could be reduced or completely avoided by introducing bus lines exclusive for the I/O susbsystems. For the time being, however, a software device is still necessary to get the job done.

5. CONCLUSIONS

The designed I/O component has proven a versatile solution to the I/O problem in a partitioned system. It proves that it is possible to design a higher level abstraction to access space domain communication protocols whilst respecting real time requirements of typical space applications.

However, the schedulability of a partitioned system featuring an I/O component shared among several applications is an open problem, especially if we consider synchronous buses like the MIL-STD-1553B. Further work is needed in proper configuration and analysis tools.

The I/O component will reach a prototype status soon, when it should be ready to be tested in effective use cases.

REFERENCES

[1] Airlines Electronic Engineering Committee (AEEC), ARINC Specification 653 Part 1 (Supplement 2), Required Services, Aeronautical Radio Inc., November 2010.

[2] Airlines Electronic Engineering Committee (AEEC), ARINC Specification 653 Part 2 (Supplement 1), Extended Services, Aeronautical Radio Inc., December, 2008

[3] B. Andrillon. Contribution of Integrated Modular Avionics of Second Generation for Business Aviation. SCARLETT Consortium, available at `http://www.scarlettproject.eu`

[4] The Consultive Committee for Space Data Systems. Spacecraft Onboard Interface Services. Informational Report CCSDS 850.0-G-1, June, 2007.

[5] R. Coutinho. Aspects of the *Architecture of Independent Distributed Avionics* (AIDA). In Proc. of the 27th AIAA/IEEE Digital Avionics Systems Conference (DASC 2008), St. Paul, Minnesota, USA, October, 2008.

[6] G. Gebhard, C. Cullmann, R. Heckmann. Software Structure and WCET Predictability. Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011), Dagstuhl, Germany, 2011.

[7] SAVOIR-FAIRE Working Group. Space On-Board Reference Architecture. In Proc. of the DASIA 2010 – DAta Systems In Aerospace Conference, EUROSPACE, Budapest, Hungary, May 2010.

[8] T. Schoofs, J. Cristóvão. Sharing the Costs of Common Tasks in the AIR Partitioning OS. In Proc. of the DASIA 2011 – DAta Systems In Aerospace Conference, EUROSPACE, San Anton, Malta, May 2011.

[9] R. Shah, Y. Lee, D. Kim: Sharing I/O in Strongly Partitioned Real-Time Systems. In: *Proceedings of the 2nd International Conference on Embedded Systems and Software (ICSS)*, pp. 502-507, Xi'an, China, September, 2005.

[10] H. Silva, J. Sousa, D. Freitas, S. Faustino. RTEMS Improved – Space Qualification of RTEMS Executive. In INForum 2009 - I Simpósio de Informática, Lisbon, Portugal, September 2010.

[11] A. S. Tanenbaum. Modern Operating Systems. Prentice Hall, December 2007.

[12] F. Torelli. SOIS and the Reference Architecture. In Proc. of the DASIA 2011 – DAta Systems in Aerospace Conference, EUROSPACE, San Anton, Malta, May 2011.

[13] C. B. Watkins, R. Walter. Transitioning from federated Avionics Architectures to Integrated Modular Avionics, in Proc. of the 26th AIAA/IEEE Digital Avionics Systems Conference, DASC 07, Columbia, Maryland, October, 2007.

[14] C. B. Watkins, R. Walter. Comparing two Industry Game Changers: Integrated Modular Avionics and the iPhone, in Proc. of the 28th AIAA/IEEE Digital Avionics Systems Conference (DASC), Orlando, Florida, October, 2009.