Time and Space Partitioning the EagleEye Reference Mission

Victor Bos[(1)], Peter Mendham[(2)], Panu Kauppinen[(1)], Niklas Holsti[(1)], Alfons Crespo[(5)], Miguel Masmano[(3)], Juan A. de la Puente[(4)], Juan Zamorano[(4)]

[(1)]Space Systems Finland, Kappelitie 6B, 02200, Espoo, Finland, Email: {victor.bos,panu.kauppinen,niklas.holsti}@ssf.fi

[(2)]Bright Ascension, 42 Queen Street, Newport-on-Tay, Fife, DD6 8BD Scotland, UK, Email: peter@brightascension.com

[(3)]fentISS, Ciudad Politécnica de la Innovación, Edificio 9B Despacho 3, 46022 Valencia, Spain, Email: mmasmano@fentiss.com

[(4)]Universidad Politécnica de Madrid, 28040 Madrid, Spain, Email: {jpuente,jzamorano}@dit.upm.es

[(5)]Universidad Politécnica de Valencia, 46022 Valencia, Spain, Email: alfons@disca.upv.es

ABSTRACT

We discuss experiences gained by porting a Software Validation Facility (SVF) and a satellite Central Software (CSW) to a platform with support for Time and Space Partitioning (TSP). The SVF and CSW are part of the EagleEye Reference mission of the European Space Agency (ESA). As a reference mission, EagleEye is a perfect candidate to evaluate practical aspects of developing satellite CSW for and on TSP platforms. The specific TSP platform we used consists of a simulated LEON3 CPU controlled by the XtratuM separation micro-kernel. On top of this, we run five separate partitions. Each partition runs its own real-time operating system or Ada run-time kernel, which in turn are running the application software of the CSW. We describe issues related to partitioning; inter-partition communication; scheduling; I/O; and fault-detection, isolation, and recovery (FDIR).

1. INTRODUCTION

EagleEye is a *reference mission* of ESA (European Space Agency) to try out and evaluate methods, technologies, and tools for space mission development [14]. The EagleEye consists of a *Software Validation Facility* (SVF) and the *Central Software* (CSW) of a satellite. The SVF and CSW are both modularized software systems which can be run on stand-alone workstations. All validation (testing) is done by software simulation of the environment. The SVF modules simulate spacecraft environment and spacecraft hardware. As such the SVF and CSW form a flexible setup for studies of new technologies. The EagleEye CSW is representative for on-board software of actual Earth Observation satellites and contains control software for satellite subsystems, including a simple (scientific) payload.

The experiences we discuss in this paper are about porting the EagleEye CSW to a processor module with support for *Time and Space Partitioning* (TSP). The activity was carried out under an ESA contract. We hope our experiences provide valuable "lessons learned" for future TSP activities for satellite on-board software.

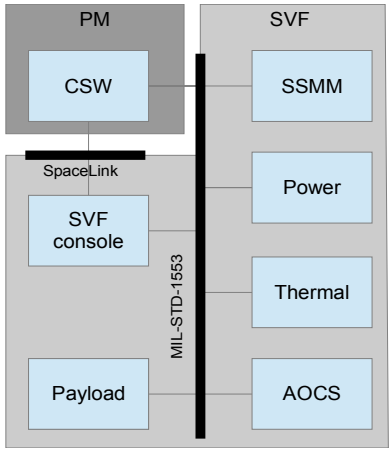

Figure 1. EagleEye SVF and CSW architecture

2. PARTITIONING EAGLE-EYE CSW

TSP for space applications is relatively new, see [7]. TSP allows an integrated system containing applications of different criticality to run on the same processor module with the guarantee that lower criticality applications cannot cause failures in higher criticality applications. The main advantage of this, when compared to an integrated system running on a conventional (non-TSP) processor module, is that the efforts needed for developing high criticality systems are independent of the size and number of lower criticality systems running on the same system.

The architecture of the EagleEye SVF and CSW is displayed in Figure 1. The processor module (*PM*) running the *CSW* is connected via a *MIL-STD-1553* bus and a *SpaceLink* to the EagleEye on-board devices which are simulated by SVF modules. These simulated devices include: Solid State Mass Memory (*SSMM*), *Power* subsystem, *Thermal* subsystem, the *AOCS*

subsystem, and the *Payload* subsystem. The *SVF console* provides the user interface with functionality for configuration, starting/stopping, test scenario execution, etc. The architecture shows that MIL-STD-1553 connects the CSW to the EagleEye on-board devices (SSMM, Power, Thermal, AOCS, and Payload). The SpaceLink provides a TM/TC link to ground.

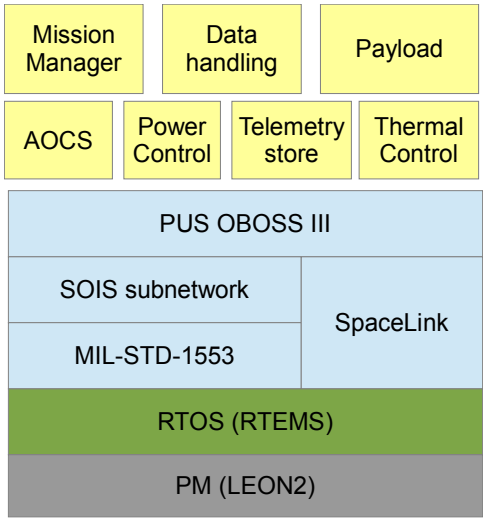

Figure 2. Initial CSW architecture

Our intention has been to change the SVF as little as possible. The PM changed from a conventional processor (LEON2) to a TSP-enabled processor (single-core LEON3 with MMU).

2.1 Initial EagleEye CSW Architecture

The CSW is, as mentioned above, a modularized software system. Its architecture at the start of the project is displayed in Figure 2. The application processes are Mission Manager, Data Handling, Payload AOCS, Power Control, Telemetry Store, and Thermal control. Below the application processes is the PUS implementation OBOSS III [4] and [5]. It provides standardized TM/TC communication between application processes and ground. Hardware access to (simulated) EagleEye equipment is provided via the SOIS subnetwork [8] and MIL-STD-1553 stack. Finally, SpaceLink provides access to the SpaceLink bus which realizes the TM/TC channel to ground. The real-time operating system used is RTEMS. The processor module is based on a (simulated) LEON2 CPU.

One characteristic of the EagleEye CSW is that it is implemented partly in Ada and partly in C. Supporting a multi-language system puts cooperability requirements on compilers, build systems, linkers, and run-time systems. Previous EagleEye activities have shown that meeting these requirements takes considerable efforts. One of the hopes of the EagleEye TSP activity is to make the cooperability requirements due to multiple implementation languages more manageable.

2.2 Final EagleEye CSW Architecture

The final CSW architecture is given in Figure 3. The partitions, displayed as five columns, are DMS, Payload, IO, AOCS, and FDIR. Five of the original application processes are put in the DMS partition. The Payload application process runs, naturally, in the Payload partition. The IO partition does not have application processes. The AOCS application process runs in the AOCS partition. Finally, the FDIR partition contain a new application process called TSP FDIR. This process monitors and controls the partitions of the TSP platform.

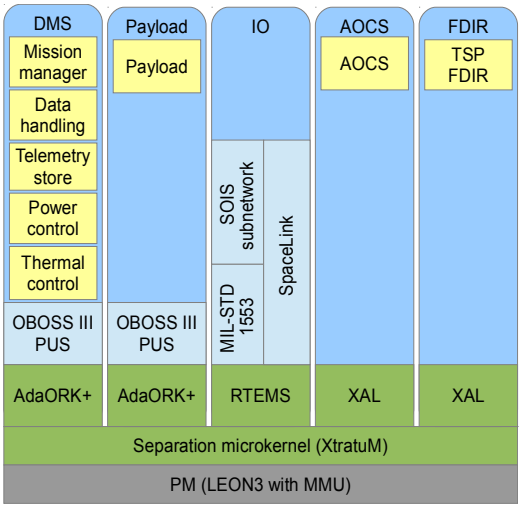

Figure 3. Final CSW architecture

Each partition has its own operating system and/or runtime kernel. The DMS and Payload partitions are implemented in Ada and run on an AdaORK para-virtualized for XtratuM. The IO partition is implemented in C and runs on RTEMS. Finally, the AOCS and FDIR partitions run on XAL, a simple operating system to run single-threaded C programs. XAL is provided with XtratuM, see next paragraph.

The separation microkernel is XtratuM [1], [2]. This is an open source software system developed and maintained by fentISS (www.fentiss.com). XtratuM was adapted specifically for this project to support the processor module for EagleEye TSP. The processor module is based on a (simulated) LEON3 CPU with MMU.

As can be seen by comparing Figure 2 and Figure 3, the OBOSS III component is duplicated in the final architecture. The reason for this is that communication

between application processes in the DMS and Payload partitions was based on OBOSS III (TM/TC PUS). As both DMS and Payload partitions run on the AdaORK runtime kernel, only minimal changes were needed to make this TM/TC based communication between Payload and DMS application processes cross partition boundaries.

3. BEFORE-AFTER DISCUSSION

The final architecture in Figure 3 shows new components in the green layer (AdaORK+, XAL, and XtratuM). However, since application processes of different partitions communicate with each other, additional software components were needed to implement this *Inter-partition communication* (IPC). XtratuM provides low-level IPC primitives with which bytes can be transfered from one partition to another. The primitives are made available by the partition operating systems or runtime kernel. The additional code we talk about here encapsulates the XtratuM IPC primitives such that changes to "legacy" EagleEye CSW code to support IPC are minimal. One TSP-specific design issue is to decide the best place to implement this encapsulating code.

During the EagleEye TSP activity, many decisions had to be made. Some of the decisions are EagleEye specific, whereas others can be generalized to broader contexts. To limit the scope of this paper, we discuss some of these broader issues. We focus the discussion by questions that can be raised by comparing the initial and final CSW architectures:

1. Why is OBOSS III in (only) two partitions?
2. How does IO work?
3. What is TSP FDIR (and why not just FDIR)?

3.1 Why is OBOSS III in (only) two partitions?

The original EagleEye CSW application processes communicate mostly via TM/TCs according to the PUS standard. The move to a TSP enabled platform placed some of the application processes in different partitions. As a result, decisions had to be made about how to implement cross-partition PUS communication. It is possible to adapt the PUS code, i.e., OBOSS III code, to run PUS over IPC. That is, PUS TC/TM packets, in their external form as octet strings, are sent over IPC from one partition to another, and there is no intimate interaction between actual PUS Services (such as service 3, HK) in different partitions. For example, PUS Service 3 in one partition cannot read and report HK parameters which exist in a different partition.

An advantage of this approach is that clients of PUS services (i.e., applications) do not need to be changed. Another advantage of this approach is that the IPC handling code is designed to implement the protocol and, therefore, likely to be application independent. A disadvantage of this solution is that the PUS code has to be available in each partition that uses the PUS protocol. Furthermore, since single-language partitions is a goal of the EagleEye partitioning activity, a PUS implementation has to be available in Ada and in C. Consequently, development and maintenance efforts of PUS code for EagleEye CSW increase substantially.

Another option is to hook-in IPC code between the PUS code and the application code. In the extreme case, this leads to a situation in which the PUS code can be replaced completely by IPC code. However, often not all PUS communication has to cross partition boundaries, e.g., PUS communication between application processes in the same partition. It would not make sense to replace such PUS communication by IPC-based communication *inside* a single partition.

Note that the discussion so far is not specific for PUS. Any communication protocol between application processes can either be implemented on top of IPC or be partly replaced by IPC.

For EagleEye CSW, we used both options. For OBOSS III code, which implements the PUS protocol, we decided that it was best to keep it as the communication protocol between the application processes running on the DMS and Payload partitions. This OBOSS III code now uses IPC to realize this DMS-Payload communication. Since both the DMS and Payload partitions are Ada partitions, the IPC code used by the OBOSS III code is exactly the same.

On the other hand, we decided that for communication between the DMS partition and any of the C-partitions (AOCS, IO, and FDIR) it would be too expensive to implement OBOSS III in C. Therefore, this communication completely bypasses PUS and is directly expressed in terms of IPC.

There are more reasons for the actual distribution of EagleEye code over partitions. For instance, the AOCS algorithmic code was (and had to be) treated as a black box developed by a third party. We assume the underlying reason for this is to simulate a business situation in which development of a (software) sub-system is sub-contracted. By aligning the partition boundaries to contractual boundaries for sub-systems, independent development and validation (of partitions) is supported by the TSP platform. We have not investigated potential advantages of this approach further, because it was not in the scope of the project. However, currently the partitioning of the AOCS feels a bit awkward, because the AOCS algorithmic code is located in the AOCS partition (written in C) and other

AOCS code, handling, e.g., pre- and post-processing is located in the DMS partition (written in Ada).

3.2 How does IO work?

The IO partition is solely responsible for handling all IO with the CSW. That is, the IO partition has exclusive access to the EagleEye's MIL-STD-1553 and SpaceLink buses.

The SpaceLink implements EagleEye's link to ground. It is used exclusively by the Mission Manager application process. The effect of moving SpaceLink access to the IO partition is that EagleEye-to-ground communication includes an extra IPC-hop. As there were no strict latency requirements for this communication link, the effect can be ignored.

The effect of moving MIL-STD-1553 access to the IO partition has more consequences. The reason is that the MIL-STD-1553 bus is shared by several application processes, e.g., Data handling, Payload, and AOCS. Several aspect are now important:

- Functional coupling of application processes on different partitions;
- Latency and/or bandwidth requirements;
- Synchronisation between MIL-STD-1553 schedule and partition schedule.

To ensure the MIL-STD-1553 bus is used correctly by the EagleEye CSW, careful analysis of these aspects and their inter-dependencies was needed. Based on this, decisions were made regarding the partition schedule and the type of IPC channels (queueing or sampling) to be used.

3.3 Redesign of MIL-bus code

During the process of porting the MIL-bus code from Ada to C and its subsequent validation, it became clear that some assumptions of the MIL-bus code intefered with the partition schedule we had defined. In particular, the MIL-bus code assumes that its clients, i.e., the application software, request a MIL-bus transfer two or more MIL-bus minor frames before the transfer has to take place. For instance, the MIL-bus schedule has a slot for AOCS actuator commands transfers in minor frame 3 (minor frame numbering starts at 0). The assumption in the MIL-bus code implies that the AOCS requests actuator commands transfers not later than minor frame 1.

Figure 4 illustrates this specific situation as it was for the unpartitioned CSW as displayed in Figure 2. Each minor frame takes 50ms (milliseconds). During minor frame 0 (0—50ms), the MIL-bus code (here indicated with I/O) acquires inputs from AOCS sensors and actuators. At the end of minor frame 0, these inputs are forwarded to the AOCS code located in the DMS (i.e., AOCS code written in Ada). In minor frame 1 (50—100ms), the inputs are forwarded from the DMS to the AOCS. The AOCS computes actuator commands from the inputs and returns the actuator commands to the DMS. Finally, at the end of minor frame 1, the requests for actuator commands transfers are sent to the MIL-bus code. At the start of minor frame 2, the MIL-bus code collects all requests issued until the start of minor frame 2 and allocates them to appropriate MIL-bus time slots. The final step of the MIL-bus is to perform the actual transfer in the bus. However, the process of collecting requests, allocating transfers, and performing transfers takes more than a complete minor frame, as indicated in the diagram. Consequently, the AOCS actuator commands transfers are not executed until minor frame 3 (150—200ms).

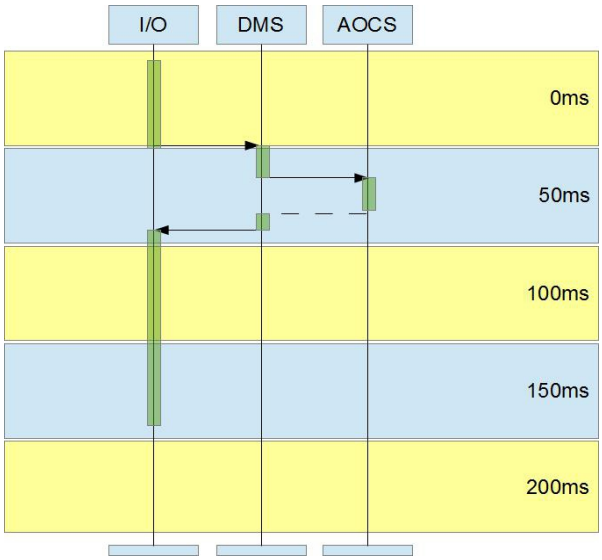

Figure 4. Illustration of original MIL-bus assumptions

Note that it is crucial that the DMS and AOCS can perform their processing completely during minor frame 1, otherwise the requests for the actuator commands will arrive late at the MIL-bus code. In the unpartitioned system, this was not a problem, because the DMS and AOCS tasks were given sufficiently high priorities to ensure were run as soon as their inputs became available.

Unfortunately, in the partitioned system of Figure 3, task priorities are not the main criteria to decide which task runs at a given time. Instead, it is the *partition schedule* which effectively groups tasks into fixed, statically defined, time slots. In each time slot, at most one partition executes. Consequently, only tasks of the

currently executing partition are able to run. Therefore, task priorities have influence only on tasks within the same partition. The partition schedule we designed is illustrated in Figure 5 together with the AOCS processing chain. As can be seen, the partition schedule allocates time slots to the DMS and AOCS such that processing of the AOCS inputs lasts until the end of minor frame 2 (100—150ms). Therefore, the MIL-bus code in the I/O partition does not receive the actuator commands transfer requests until minor frame 3.

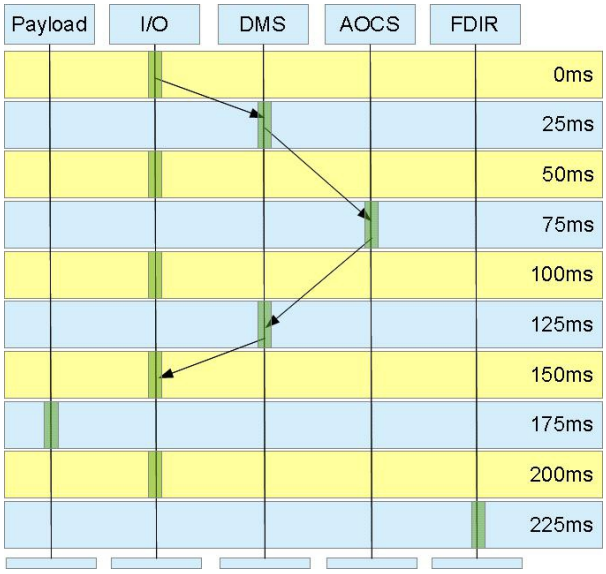

Figure 5. MIL-bus processing in the partitioned system

From this diagram, it is clear that the MIL-bus code starts processing AOCS actuator command requests only in minor frame 3 (150—200ms). This is the same minor frame in which the actuator commands have to be transferred according to the MIL-bus schedule.

Consequently, we either had to update the partition schedule such that the DMS and AOCS partitions would be scheduled often enough during MIL-bus minor frame 1 or we had to redesign the MIL-bus code such that transfer requests can be processed in the same minor frame as in which they are received by the MIL-bus code. We chose for the latter option and updated the MIL-bus code accordingly. This means the Figure 5 shows the actual AOCS processing chain of the partitioned CSW.

3.4 What is TSP FDIR (and why not just FDIR)?

One issue was the FDIR partition. The required behaviour of the FDIR partition was health monitoring and control of partitions. The original EagleEye CSW has some FDIR functionality to handle redundancy in AOCS equipment. Moving this functionality to the FDIR partition would be a valid design decision. However, this would require identifying the existing FDIR code and porting it to the FDIR partition. Since this was not explicitly required, we decided it was better to focus on providing TSP-related FDIR functionality without changing existing FDIR functionality. Future EagleEye activities can focus on redesigning the EagleEye FDIR system, including the TSP FDIR, and restructuring the EagleEye TSP architecture so that the FDIR partition contains a general and complete FDIR application process.

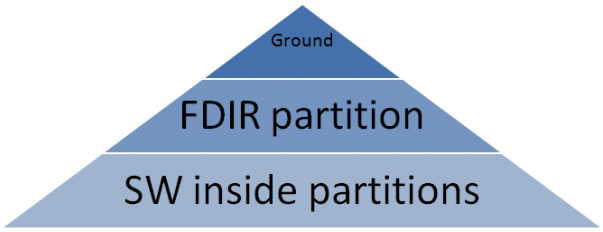

Figure 6.Hierarchical FDIR

As illustrated in Figure 6, we adopted a hierarchical FDIR system for EagleEye CSW. At the lowest level, (some) partitions perform their own FDIR. If a failure cannot be resolved at partition level, it is propagated to the FDIR partition. Finally, if the FDIR partition cannot handle it, the failure is propagated to ground.

3.5 TSP Development Environment

In addition to architectural decisions, as described above, decisions about changes to the development environment have to be taken. For EagleEye, the development environment consists of Makefiles, scripts, configuration files, and tool chains including compilers, linkers, etc.

When moving to a TSP platform, new functionality is needed to configure and use the separation microkernel. For EagleEye TSP, we approached this problem by creating and/or adapting scripts to use the functionality that come with the TSP platform (i.e., using tools and scripts from XtratuM, AdaORK+, RTEMS). This resulted in an ad-hoc collection of tools, scripts, and configuration files. Some of the main problems we had with this approach are:

1. Certain TSP configuration parameters are defined in multiple places;
2. Debuggers cannot easily be used;
3. No tool support for partition schedule analysis

We expect that future work is needed to develop a more consistent and complete development environment. If

done properly, this such work will greatly benefit other TSP-related activities for on-board software.

4. RESULTS

During the EagleEye TSP activity, several concrete software products were adapted and/or created. Firstly, the open source XtratuM separation microkernel was ported to the LEON3 processor. fentISS, the company behind XtratuM, joined the EagleEye TSP consortium to realize the XtratuM port for LEON3 and to provide support to the other parties of the consortium. In addition to LEON3, XtratuM runs on LEON2, LEON4, x86, and PowerPC. EagleEye TSP was developed mostly on XtratuM 3.3.3. The final EagleEye TSP runs on XtratuM 3.4. The XAL operating system, which we used as partition OS in the FDIR and AOCS partitions, is part of XtratuM.

The Ada runtime we used, AdaORK+, existed already for the XtratuM/LEON2 platform. AdaORK+ is an open source software product developed by the OpenRavenscar Project at the Universidad Politécnica de Madrid (http://web.dit.upm.es/~ork/index.html/) 13. Members of the OpenRavescar project joined the EagleEye TSP consortium to port the AdaORK+ runtime kernel to the XtratuM/LEON3 platform and to provide other parties of the consortium support.

The EagleEye SVF was updated to support EagleEye CSW running on a LEON3 processor module. The differences between the original SVF and the updated SVF are minimal. In fact, the the updated SVF can run both the unpartitioned and the partitioned CSW (both on a LEON3 processor module). The main additions to the SVF consist of a unit test framework to develop and execute tests in which IPC (Inter Partition Communication) is the main interface to the system under test. We have used this unit test framework to validate the five EagleEye CSW partitions independently from each other.

The EagleEye CSW architecture and software is now divided over single-language partitions. As a consequence, cooperability requirements, as discussed above, have become less limiting. Each partition can be developed and maintained with tools for one implementation language. Integration is based on IPC, which means the integration tools need not be aware of the implementation languages at all.

The MIL-STD-1553 and SpaceLink code has been ported to C and runs on RTEMS. These software products are relatively independent of EagleEye CSW and can therefore be reused in other projects.

The (new) TSP FDIR code demonstrates clearly the type of monitoring and control necessary/desirable in a TSP-based system. The TSP FDIR code can be adapted to future EagleEye CSW changes with more partitions.

Finally, the overall result is, of course, the EagleEye CSW for a TSP enabled system. It is our belief that the resulting EagleEye CSW has become more modular and, therefore, simpler to adapt or extend in future projects. Most of the TSP benefits will have impact on EagleEye development and maintenance efforts. However, some TSP benefits can be visualized directly by running the EagleEye CSW. The TSP demonstration scenarios show, e.g., how the TSP-based EagleEye system is capable of detecting and recovering from failing partitions.

5. REFERENCES

1. XtratuM, http://www.fentiss.com/, Accessed 27th January 2013.
2. XtratuM, http://www.xtratum.org/, Accessed 27th January 2013
3. RTEMS Centre, http://rtemscentre.edisoft.pt, Accessed 6th June 2012
4. OBOSS System Integration - Manual for Reuse, Terma/SPD/OBOSS-III/013, Issue 1, 9th February 2004
5. OBOSS-III Operations Manual, TERMA/SPD/OBOSS-III/012, Issue 1, 5th February 2004
6. Securely Partitioning Spacecraft Computing Resources - FINAL REPORT, SSL/08467/DOC/012, Issue 1.1, 25th July 2011
7. Time and Space Partitioning in Spacecraft Avionics, Windsor, James, Kjeld Hjortnaes, 2009, Third IEEE International Conference on Space Mission Challenges for Information Technology
8. CCSDS 850.0-G-R1.1 May 2010, Green Book – Spacecraft Onboard Interface Services
9. IMA-SP - I/O Handling Strategies, IMA-SP/D06 Issue: 1.0, 15th April 2011
10. IMA-SP - IMA-SP Onboard Software Maintenance Strategy, IMA-SP/D09 Issue: 1.4, 5th October 2011
11. IMA-SP - IMA-SP System Level FDIR Approach, IMA-SP/D07 Issue: 1.3, 5th October 2011
12. S. D. Fowell, P. Mendham, A View on the use of SOIS in a TSP-Based Architecture, Proceedings of Data Systems in Aerospace (DASIA), Budapest, Hungary, 2010.
13. GNAT/ORK: An Open Cross-Development Environment for Embedded Ravenscar–Ada

Software, Juan Zamorano and José Ruiz, 15th triennial World Congress, Barcelona, Spain, 2002.

14. EagleEye Virtual Spacecraft System Architecture, Michael Schön (ESA) and Gert Caspersen (TERMA), TOS-EMS-VSRF-TN-0002, Issue 1, revision 2, 23.03.2004.