Mission Critical Software in LISA Pathfinder

J. A. Ortega-Ruiz, A. Conchillo, X. Xirgu and C. Boatella

Institut d'Estudis Espacials de Catalunya (IEEC), Barcelona, Spain

Abstract. We provide a quick overview of the challenges faced during the engineering of LISA Pathfinder's Data Management Unit from the perspective of software development. Data collection and system monitoring is coordinated by the DMU via a software system that faces many a design and implementation challenge in the form of performance and reliability requirements. This paper introduces these challenges, their origin, and the software techniques used to overcome them.

Keywords: Gravitational Waves, LISA, LISA Pathfinder, Software Engineering
PACS: 04.80.Nn, 95.55.Ym, 04.30.Nk

OVERVIEW

From a technological point of view, LISA Pathfinder (LPF) is a major engineering undertaking, which involves a variety of (sometimes brand new) hardware and software developments. One of the components of LPF is the Data Management Unit (DMU), in charge of coordinating the acquisition of data from the laser interferometer as well as the different diagnostic subsystems that conform the mission. Thus, the DMU will play the part of a communications gateway between the central on-board computer in the spacecraft and several subsystems, with a considerable amount of autonomous operative and control responsibilities which grant a dedicated on-board computer and associated software. This presentation describes the architecture and main design traits of the software in charge of controlling the DMU[1]. Functionally, it must meet a series of non-trivial requirements, including real-time performance and the adequate use of redundant communication channels. Moreover, the architecture must be flexible enough to accomodate in-flight modifications of the executable software (in the form of telemetry patches), while, at the same time, ensuring the robustness of this mission-critical application. In addition to describing how and why we combine off-the-self components (like real-time operating systems) with special purpose software in order to meet our ends, we intend to provide an overview of the engineering processes and development standards in place to ensure the accomplishment of the stringent requirements we face.

THE DATA MANAGEMENT UNIT

Figure 1 gives a simplified, yet accurate, description of the interaction context where the DMU's activities take place. As we can see, its main responsibility is acting as an

[1] We refer the reader to Alberto Lobo *et al* in this volume for a description of the diagnostic subsystems coordinated by the DMU software.

interface between the on-board computer (OBC) and the measuring instruments that conform the scientific payload of the LPF spacecraft, namely, the phasemeter and inertial sensors on the one hand and the diagnostic subsystems on the other. The DMU receives commands from the OBC, forwarding them as necessary to allow remote control of the scientific instruments. It also collects and delivers the science data produced by those instruments and monitors the overall status of all subsystems.

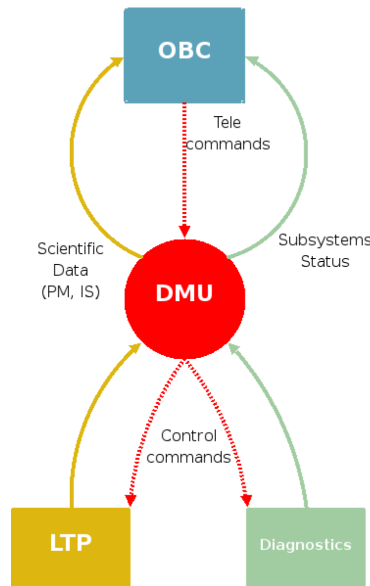

FIGURE 1. A simplified block description of the DMU's interaction context.

The DMU software must meet not only the above functionality, but also a set of non-functional requirements that actually constitute the main challenges faced during its implementation. To begin with, we need to ensure the robustness and reliability of the system, providing adequate recovery pathways in case of error or malfunction. The software must also meet hard real-time constraints when it comes to timely delivery of scientific data, since phasemeter readouts must be collected at a very precise frequency of 100Hz. Finally, these goals are to be attained with a small footprint, both in space and processor resources, imposed (as we show below) by a software architecture designed to fulfill rather stringent redundancy and safety requirements.

Of course, no software system is an island. The DMU software runs on a specially designed hardware platform, which often simplifies and improves (but sometimes hampers) the implementation of the required features. The main traits of the DMU hardware platform are the following:

- An ERC32, RISC-based processor running at a clock frequency of 12 MHz, which imposes a hard limit on the software's operation velocity.
- 1 Mb RAM memory, with automatic 2-bits error detection and 1-bit correction, which greatly simplifies memory error management. In addition we have at our disposal 2 Mb of EEPROM memory, which is used for modifiable, persistent code.
- 64 Kb PROM (non-modifiable) persistent storage, for bootstrapping the whole system. The reduced amount of PROM available constitutes a major challenge.

- Robust communication with external systems via MIL-STD-1553B buses. These communication devices use 20-bit buses which include a 4 bit Hamming code for error detection. In addition, most of the data-link communication layer is independently handled by the available bus controller chips.
- Two FPGAs providing memory-mapped access to all external devices. Another big win from the software's point of view, since these specially programmed FPGAs hide all the complexity associated with the diverse hardware interfaces of the DMU.

Finally, every hardware component is redundant. This fact not only boosts the DMU's overall reliability, but simplifies our software by removing complex execution paths usually associated with recovery actions in the face of hardware failure. As a generic architectural design decision, we avoid all but the simplest hardware recovery actions, relying instead on a switch to redundant components. Our main worry becomes therefore ensuring the reliability of software operations *per se*, with an architecture, reviewed in the next section, that favours flexibility and remote intervention.

THE ARCHITECTURAL DIVIDE

As already mentioned, the DMU's software architecture must maximize flexibility and remote recovery actions in order to prevent unrecoverable lock-outs during the mission's operational life. As much as a proper quality assurance process (as described below) provides guarantees of the software's correctness, the possibility of undetected flaws in the flying software cannot be discounted. Thus, the design leaves open the possibility of bug fixing (or even functional modifications) during the DMU operations. Obviously, the smaller a source code base is, the larger will be our chances of making it bug-free. Therefore, we have extracted a small functional core, called the Boot Software (BSW), which fits in the 64 Kb PROM and is in charge of bootstrapping the system. The rest of the application software, stored in the EEPROM, is remotely modifiable, allowing the correction of unexpected flaws during the mission lifetime.

Turning our attention to the BSW, the goal of producing a 64 Kb ($\approx 10^4$ LOC in C with assembler snippets) application is attainable, although this reduced size poses certain implementation challenges. Admittedly, 64 Kb is usually not a small size by embedded systems standards, but in our case, the BSW is not only in charge of booting the system, but also of setting up the communications link with the OBC (where we find non-trivial data-link and transport protocol layers), keep an eye and report on the overall hardware health status and managing the ASW code (stored in EEPROM), allowing its patching by incoming telecommands. This actually amounts to implement a small real-time kernel, as shown in figure 2. Here we find a hardware abstraction layer (greatly simplified by the FPGAs, but needing nonetheless a driver for the MIL-STD controller), a communications stack implementing the OBC-DMU transport protocol (PUS), and a task scheduler to manage concurrent activities. Those modules are among the typical ones to be found in regular operating systems, although implemented in a much simplified fashion (for instance, the scheduler is based on polling). Over them, the key high-level task is the ASW manager, which attends incoming patching requests and finally loads and excutes the ASW.

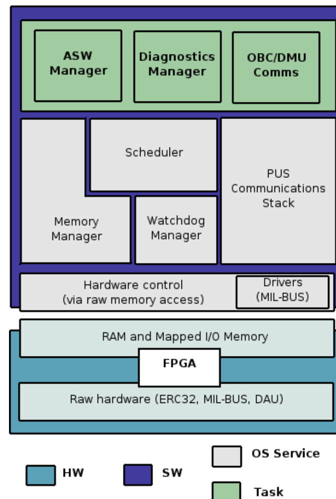

FIGURE 2. The BSW actually implements a small real-time operating system.

Although the ASW takes the lion's share of the DMU's operational time and implements a much wider range of functional requirements, its design and implementation is simplified by the use of a COTS real-time kernel (RTEMS[2]), and the reuse of the communications driver and protocol stack already present in the BSW. The main challenge faced during the ASW development is meeting the strict deadlines assigned to processing and delivery of phasemeter read-outs, which demands a precise time slicing for concurrent tasks. The hard real-time capabilities offered by RTEMS have been a key element in our accomplishing these goals, by providing higher-level abstractions and patterns. It is also worth noting that the maturity of the kernel's code base contributes to our confidence in the solidity of the resulting application, and simplifies the quality assurance process for the ASW.

QUALITY MATTERS

Quality Assurance is an integral process encompassing all stages of our software development life-cycle. It relies on a series of engineering standards, taylored from ESA templates, and covering all key aspects of the development.

Testing is central. Defined in the verification standards, it includes thorough unit testing (with processor emulators support) during the coding phase, followed by (module) integration tests (using simulators for all external subsystems communicating with the DMU) and, finally, validation tests against the mission requirements.

A well defined engineering process is in place, relying on key milestones (PDR, CDR, TRR) with the participation of all stakeholders. The whole process is captured in a solid, peer-reviewed documentation set, mimicking the waterfall development model: System Requirements, Software Requirements, Architectural and Detailed Design, Validation

[2] See `http://www.rtems.org`.

and Verification plans. The ideal waterfall process is the expected outcome of ESA's Engineering Standards. It works as a reporting template for the final product, whose actual development is much closer to the well-known *spiral model*. Especially noteworthy is (unit) testing, which actually drives code writing, in the test-driven development (TDD) spirit[3].

As for the actual methodology driving the architectural and detailed design (iterated) phases, we are using Ward-Mellor Structured Design[4], a Yourdon derivative explicitly taylored to real-time constraints. In this regard, we have consciously avoided heavy-weight methodologies and notations like UML/RUP (or languages like C++, for that matter) in order to keep our processes as lightweight as possible, making quality assurance far easier. In addition, our C-based implementation (even with heavy use of data and functional abstractions) is more amenable to a structured design description than an (exclusively) object-oriented one.

TOOLS OF THE TRADE

Let us close our overview with an enumeration of our development toolchain, which is, in our view, a key ingredient in the eventual attainment of our goals.

- Automatic tools
  - Code coverage (*Cantata*). 100% execution and branch.
  - Code linting (*splint*).
  - Metrics extraction: MacCabee, cyclomatic (*Cantata*).
- Semi-automatic tools
  - In-code documentation (*doxygen*).
  - Test libraries, stubbing and drivers (*Cantata*).
  - Software and documentation distributed version control system (*GNU Arch*).
  - Emacs, a programmable editor.
- Plain tools
  - GNU cross-compilation tool chain.
  - ERC32 emulator.
  - LaTeX, a programmable documentation system.
  - *MoinMoin WikiWiki*, a centralised information repository.
  - *Python*: gluing it all together.

Acknowledgements: We thank Ministerio de Educación y Ciencia for support under contract ESP2004-01647.

[3] See, for instance, `http://www.agiledata.org/essays/tdd.html`.
[4] See *Structured Development for Real-Time Systems* by P. T. Ward, S. J. Mellor, Prentice Hall 1987.