Development platform for LPF payload flying control software

Víctor Martín, Lluís Gesa, Carlos Sopuerta for the LPF Collaboration
Institut de Ciències de l'Espai (CSIC-IEEC)

Introduction

LISA Pathfinder (LPF) has placed two test masses in a nearly perfect gravitational free-fall, and has controlled and measured their relative motion with unprecedented accuracy.

To control the **L**isa **T**echnology **P**ackage **(LTP)** an embedded payload control software has been developed to be executed inside the **D**ata **M**anagement **U**nit (**DMU**). To produce it, a huge infrastructure has been built during more than 10 years.

Concept

This infrastructure allow us to develop, validate with fast and repeatable tests, and execute ours **B**oot **S**oft**W**are (**BSW**) and **A**pplication **S**oft**W**are (**ASW**) in pure 100% software emulated/simulated hardware environment, and/or in real hardware emulating all **LTP** subsystems or OBC when needed. The several components that have been built, can be classified in three big interconected blocks :

- **Implementation Technology**
- **Testing tools**
- **Simulation/Emulation Tools**

Implementation Technology

Several components written in C (C99) has been built to encapsule the needed functionalities for the flight software. This diagram shows them.

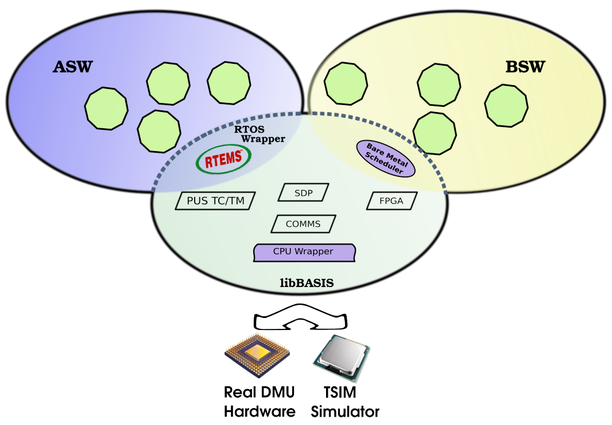

Figure: Implementation components diagram

The main modules are

- **ASW:** Main Application, containing all the simultaneous tasks over RTEMS.
- **BSW:** Manages startup, ASW load, patch and boot communicating with **OBC**.
- **libBasis** Main common functionality, as communications (MIL-STD-1553), SDP, PUS TC/TM, EEPROM, FPGA, etc.

The builder environment makes it possible to compile for several platforms:**SPARC v7** bare metal, RTEMS operating system, and for i386 under GNU/linux. This environment is a customized version of an existing python tool, Scons.
The key components of the implementation are :

- **libBasis** wraps the underlying **R**eal **T**ime **O**perating **S**ystem (**RTOS**): RTEMS for **ASW**, our own system for **BSW**, so migration to another **RTOS** would be isolated to that lib.
- **BSW** implements a small scheduler, used to be able to multiplex between all the needed tasks without an operating system
- Communications using a custom **MIL-STD-1153** bus driver to allow communications with all subsystems and **OBC**.

Test **D**riven **M**ethodology **TDD** has been used, performing a big effort to create all the stubs to test functionalities without needing the real hardware, to run over the **ERC32** simulator (**TSIM**), and managed with **Cantata**, a commercial tool that allows to check also code coverage.

Profiling tools has also been created to check the hard CPU load and RAM requirements, for example with the **Stress Tests**, created to check the behaviour of the application and the resources use on hard demanding conditions.

Simulators and Emulators

Hardware to run and test the software has not been available since the beginning. Using **TSIM**, the **ERC32** simulator, simulation and tests have been possible while developing, using custom plugins for **TSIM** to simulate the existing DMU hardware: Memories(**EEPROM**,**RAM**), **DDS** equipment,**Summit MIL-STD-1153** Bus chip.

Once we can run on simulated or real **DMU** hardware, we need to emulate the **OBC** behaviour. It has been achieved by a partial **OBC** emulator, a **SCOE TM/TC-Frontend**, capable of run customizable and dynamic python sequences of **OBC** execution steps.

Several tools to emulate internal status and data provided by **LTP** subsystems has been implemented. Injecting pre-defined data, allow us to check expected output from expected input on a hardware system.

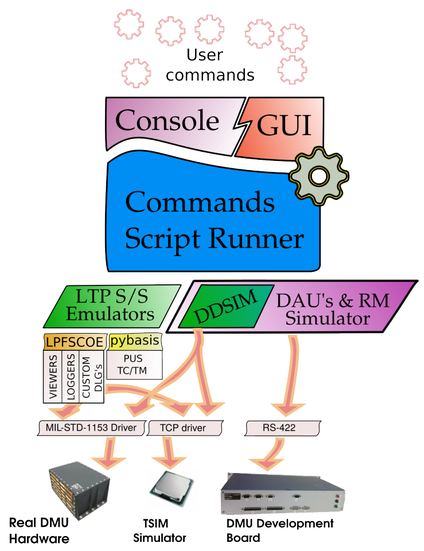

Figure: Blocks diagram of the Emulators/Simulators interactions

User script commands can be run using customizable GUI, or by console, that allow batch execution. These scripts can be run over:

- **DMU** real hardware allowing us to check diagnostic items for example.
- **TSIM** To run fast checks and stress tests over simulators .
- **Development Board** It has no DAU, so communication with DAU can be checked using **ddsim**.

Simulators and Emulators (cont.)

Using custom drivers to communicate data between these scripts to **TSIM** (TCP) or real hardware (**MIL-STD-1153**) , is transparent for the user if it communicates to real or simulated hardware.This way, the same **SCOE** commanding scripts can be run on real **DMU** hardware or **TSIM** simulators.

To check **D**ata **M**anagement and **D**iagnostics **S**ub-System (**DDS**,DAUs and RM) behaviour and communication, a tool called **ddsim** has been created. This tool, that simulates the LPF diagnostics items performance, can be attached to real hardware (**RS-422**) or to **TSIM** (**TCP**) environment.

Software Verification Facility (SVF)

Two **SVF** made from scratch by our team run over these simulators and emulators:

- One that can run with simulated or real hardware, running on GNU/Linux
- One dedicated to run with development **DMU** model, running on Windows

Both have been used to check functionality on every **ASW** release, but also for the day by day development.

All the python tools to do that, can run under console, or managed by a customizable GUI interface that allows easy and fast interaction.

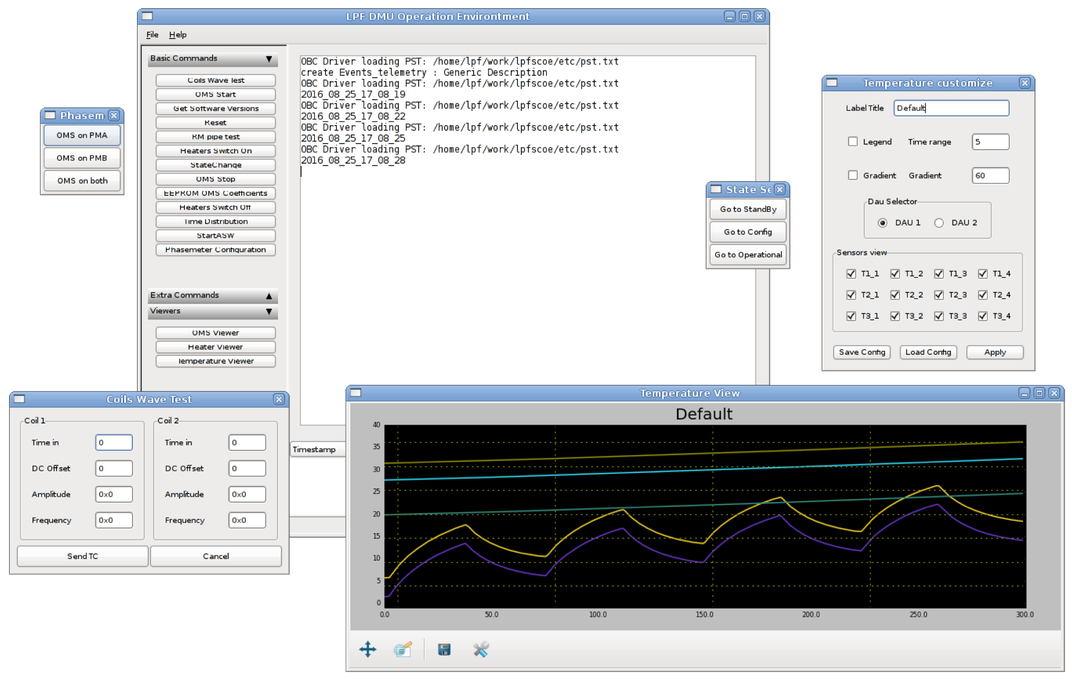

Figure: SVF GUI snapshot

Numbers

Some numbers about Main Application (**BSW** not included):

ASW (Application)

Module	*Lines of code*
Basis (ANSI C)	16411
ASW (ANSI C)	40983
Total	57394

TESTING

Module	*Lines of code*
Unit Test (ANSI C)	82850
Validation	
Functional Validation (ANSI C)	14519
Functional Validation (Python)	208170
Common (ANSI C)	8400
Common (Python)	32254
Stress Test (Python)	5263
Comms Validation (Python)	9745
Total (ANSI C)	105769
Total (Python)	255432
Total	361201

418595 Lines of code to produce DMU Main Application.

Conclusion

As a critical piece, the **DMU** software has been designed, implemented, and tested with special effort in guarantee its correctness. Apart from code itself, a huge infrastructure has been built to help achieve all the targets for this complex system:

- **Validation:** Full set of tools to automatically check system requirements, with customizable GUIs.
- **Unit Testing:** To achive Test Driven Development reliability.
- **Simulators and Emulators:** For testing and day by day developing.
- **Communication drivers:** To check components agains real hardware or different simulators.

So, it is expected to be a built and extendible infrastructure, able to be updated and re-used for future satellite / embedded mission software.

Acknowledgments

Support from MICINN via contract code AYA2010-15709 is thankfully acknowledged.

WWW: http://gwart.ice.cat
Mail: gesa@ieec.cat / vmartin@ieec.cat
Institut de Ciències de l'Espai (CSIC-IEEC) - Barcelona, Spain