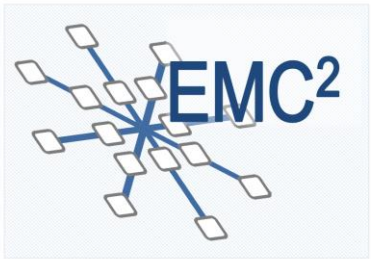


Embedded multi-core systems for mixed criticality applications in dynamic and changeable real-time environments

Project Acronym:

EMC²

Grant agreement no: 621429

Deliverable no. and title	**D9.5– Space Applications Detailed Description**	
Work package	WP9	Space Applications
Task / Use Case	T9.1-T9.5	Space Applications
Subtasks involved	UC3.1_T1-UC3.1_T8, UC3.2_T2-UC3.2_T5, UC3.3_T2-UC3.3_T7, UC3.4_T3	
Lead contractor	Infineon Technologies AG Dr. Werner Weber, mailto:werner.weber@infineon.com	
Deliverable responsible	Thales Alenia Space Spain Tres Cantos, Madrid Elena García, elena.garciavalderas@thalesaleniaspace.com Dr. Manuel Sánchez manuel.sanchez@thalesaleniaspace.com	
Version number	v1.0	
Date	17/12/2016	
Status	Final Version	
Dissemination level		

Authors

Partici-pant no.	Part. short name	Author name	Chapter(s)
64	15I INTEGRASYS	Javier Valera	Use Case Optical Payload Applications
62	15E TECNALIA	Elena Terradillos Daniel Múgica	Use case MPSoC Hardware for Space
70	15Q ITI	Sergio Saez	Use Case MPSoC Software and Tools for Space
19	02D TTT	Arjan Geven	Use case MPSoC Hardware for Space
63	15F TASE	Dr. Manuel Sanchez	Use Case Optical Payload Applications
50	11J TASI	Dario Pascucci	Use Case Platform Application

Document History

Version	Date	Author name	Reason
v0.1	13/12/2016	WP9 Partners	First version
v0.2	15/12/2016		Project internal review
v1.0	17/12/2016	Alfred Hoess	Final editing and formatting, deliverable submission

Publishable Executive Summary

Scope of this document is to provide a detailed description of Space Applications in the frame of WP9 Living Lab. Figure 1 provides an overview of the WP9 organization. Target of this document is to support T9.1- T9.5 by the structuring of the use case building blocks.

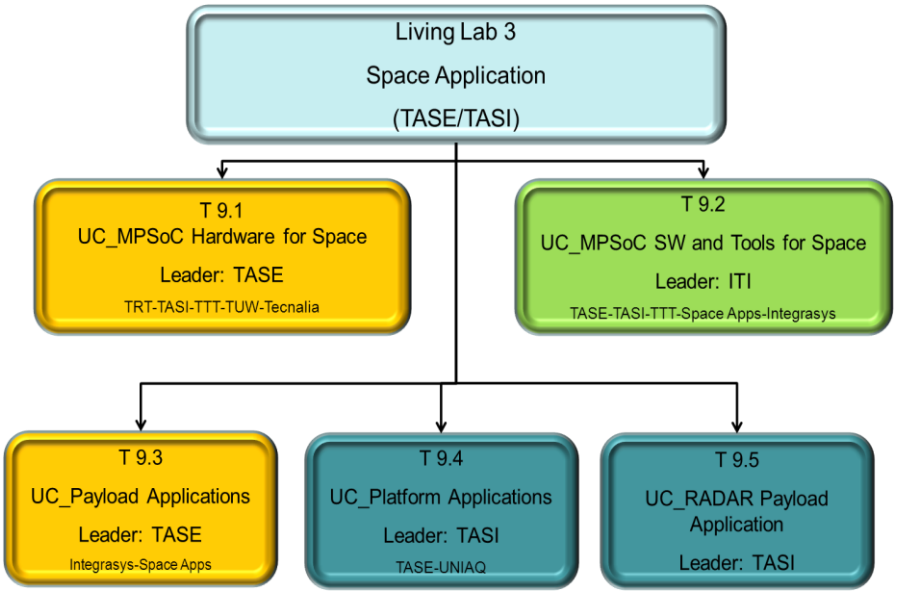


Figure 1: WP9 Organization

This deliverable provides an explanation of the use cases and the contributions of the EMC² WP9 partners.

Table of contents

List of figures

List of tables

1. Introduction

1.1 Objective and scope of the document

Scope of this document is to summarize the space applications of the use cases related with WP9 (Space Applications).

The description comprises the hardware and software involve in each use case. MPSoCs suitable for space applications have been selected in order to obtain fault-tolerance processors.

1.2 Structure of the deliverable report

The document is organized as follow: Section 2 provides the use case of hardware for space. Section 3 provides the use case of software and tools for space, while Section 4 provides the use case of optical payload applications. Finally Section 5 provide use case platform applications.

2. Use case MPSoC Hardware for Space

2.1 Fault-Tolerant and Self-Healing Dynamic Reconfiguration Manager

A Reliable and Self-Healing Dynamic Reconfiguration Manager (DRM) is the solution proposed by TASE and TECNALIA in order to perform partial reconfiguration of a Virtex-5QV FPGA in a safe (reliable) way in the space environment, which is the main technical breakthrough pursued in this WP9 use case. This solution has been designed in the context of WP4, while its robustness will be evaluated in WP9. A basic DRM, that is, a DRM without fault-tolerance elements, running on the Virtex-5 FPGA of the LADAP board and performing partial reconfiguration under command of LEON processor was demonstrated during the second year review. Now, all the specified fault-tolerance elements have been included to make the DRM reliable and self-healing. Figure 2 shows the final architecture block diagram of the Reliable and Self-Healing DRM designed by TECNALIA and implemented in the Virtex-5 device of the LADAP platform. It is externally controlled and monitored by a LEON processor implemented by TASE in the RTAX device of the hardware platform.

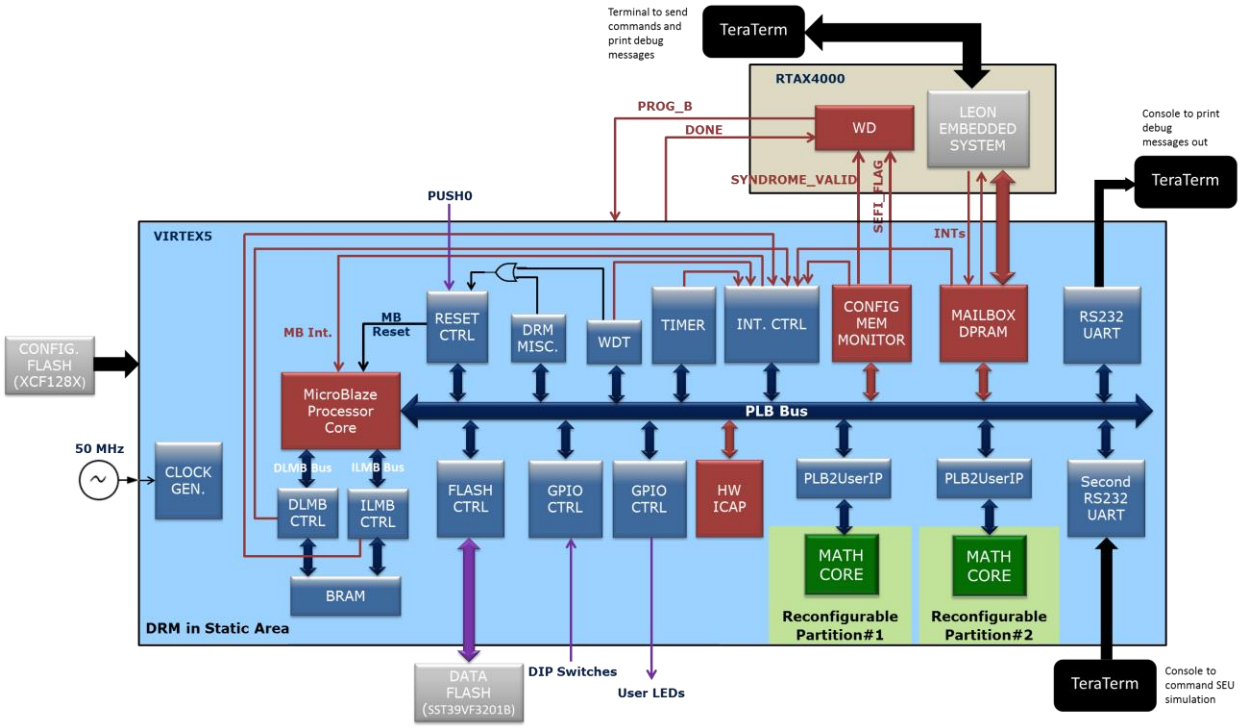


Figure 2: System architecture diagram of the Reliable and Self-Healing DRM

The DRM hardware implementation details have been described in D4.9 [1]. The rest of this section describes the software implementation details including the MicroBlaze application, the memory map details and the final LEON-DRM communication protocol.
Results of the robustness and self-healing evaluation of this implementation will be presented in the final deliverable D9.6.

2.1.1 DRM software application description

The DRM software application is executed by MicroBlaze. It consists of three parts: initialization process, main loop and interruption routines.

Initialization

The DRM application starts with an initialization process that performs the following tasks:

- Initialization of global variables that behave as interruption or action flags, and global statistics counters.

- Driver initialization for the data and instruction BRAM controllers and initialization of the ECC bits in the MicroBlaze BRAM memory.
- Driver initialization for the secondary RS232 UART.
- Installed flash memory check to verify that it consists of two Microchip SST39VF3201B devices.
- HwIcap controller driver initialization.
- GPIO driver initialization to control the user LEDs. All the GPIO signals connected to LEDs are set as outputs, and the LEDs are turned off.
- Full device image data flash area check to verify whether the current design version is loaded at data flash memory offset 0x0 or not. If not, the full device configuration data (block type 000 only) is read through ICAP and written into flash.
- Interrupt controller setup which includes driver initialization, interrupt controller self-test, exception table initialization and interrupt controller start.
- Timer interruption setup (interruption enabled, but timer still disabled).
- Mailbox DPRAM interruption setup.
- Configuration Memory Monitor interruption setup. At this point, the Configuration Memory Monitor is also enabled and it starts working.
- Interruption setup for the MicroBlaze data and instruction BRAM controllers.
- Watchdog Timer initialization and interruption setup.

Main loop

After initialization, the application enters a continuous loop executing the following flow:

1. Check if the Configuration Memory Monitor has detected some error. If yes, perform the correcting scrubbing action. The corresponding pseudo-code is as follows:
   if (scrub_cmd_req != 0) {
   - Set MbAccessOn bit to 1
   - Check FAR register
   - If FAR checking ends successfully, check for fake SEFI, perform scrub action and increase the corresponding statistic counter
     Else, set SEFI_FLAG high
   - Set MbAccessOn bit to 0
   - Clear scrub command request flag (scrub_cmd_req = 0)

   }
2. Check if the PLB Timer has expired. If it has, perform full device scrubbing. The corresponding pseudo-code is as follows:
   if (periodic_scrub_on == XTRUE) {
   - Set MbAccessOn bit to 1
   - Check FAR register
   - If FAR checking ends successfully, check for fake SEFI, perform full device scrubbing and increase the corresponding statistic counter
     Else, set SEFI_FLAG high
   - If there is a pending scrubbing request from the Configuration Memory Monitor, cancel it since a full device scrubbing has been done
   - Set MbAccessOn bit to 0
   - Clear periodic full device scrub request flag (periodic_scrub_on = XFALSE)

   }
3. Check if LEON has written a new command. If it has, process it and answer back. The corresponding pseudo-code is as follows:
   if (new_cmd_req == XTRUE) {
   - Read command from Mailbox DPRAM and check its syntax
   - Process command
   - Write answer back into Mailbox DPRAM
   - Assert interruption for LEON processor
   - Clear new command request flag (new_cmd_req = XFALSE)

- If a DRM reset has been request (through the Configure SEU Mitigation command), set the DRMReset bit in the DRM Miscellaneous core to 1.

}

4. Check if a valid SEU simulation command has been received from the secondary UART. If yes, execute it.
5. Refresh MonitorEn (high) and MbAccessOn (low) bits in Configuration Memory Monitor. This is a sanity action to correct unexpected changes in these bits due to SEU.

Interruptions

The interrupt sources from more to less priority are: WatchDog Timer (WDT), Configuration Memory Monitor, ILMB BRAM Interface Controller, DLMB BRAM Interface Controller, Mailbox DPRAM and PLB Timer. The actions done in their callbak functions are:

1. WDT interrupt callback:
   - Restart the watchdog timer by executing the XWdtTb_RestartWdt function.
2. Configuration Memory Monitor interrupt callback:
   - Update the scrub_cmd_req, scrub_addr, scrub_fr_num and scrub_bit_index variables with the values read from the cfg_mem_mon_reg2 and cfg_mem_mon_reg3 registers.
3. ILMB/DLMB BRAM Interface Controller interrupt callback:
   - Read LMB BRAM Interface Controller Interrupt Status register
   - If correctable bit error, correct memory content and increase statistic counter
   - If uncorrectable bit error, increase statistic counter
4. Mailbox DPRAM interrupt callback:
   - Set global variable new_cmd_req = XTRUE
5. PLB Timer interrupt callback:
   - Set global variable periodic_scrub_on = XTRUE
   - Perform MicroBlaze BRAM scrubbing

The DRM application has been created and compiled with Software Development Kit (SDK), included with the Xilinx EDK 14.7 design tools. The application project is named DRMApp. The next table summarizes the content of the different project files.

File name	Description
DrmAppMain.c	Application top file. It includes the main function.
commands.c commands.h	These files include the functions to read commands from and write answers to the Mailbox DPRAM, and to process and execute the actions derived from the received commands. They also include the top level functions related with memory scrubbing, SEFI detection and SEU simulation.
interrupts_utilities.c interrupts_utilities.h	These files include the functions to configure the Interrupt Controller, setup the interruptions of the different peripherals, and the interrupt routines.
HwICAP_fns.c HwICAP_fns.h	These files include the functions related with the access to the ICAP port that are not included in the HWICAP driver.
xbram_fns.c xbram_fns.h	These files include the functions for BRAM controllers and BRAM ECC bits initialization.
flash.c flash.h	These files make up the data flash driver.
mailbox_dpram.h	Simple driver for the Mailbox DPRAM.
config_mem_monitor.h	Simple driver for the Configuration Memory Monitor.
plb_timer.h	Simple driver for the PLB Timer.

drm_misc.h	Simple driver for the DRM Miscellaneous core.
GlobalPkg.h	General include file with constants, variables and types definitions.

Table 1: DRM application files

2.1.2 Memory Map

The DRM system memory map is summarized in the next table:

Peripheral	Base Address	High Address
BRAM	0x0000 0000	0x0000 FFFF
User LEDs	0x8140 0000	0x8140 FFFF
DIP Switches	0x8142 0000	0x8142 FFFF
Interrupt Controller	0x8180 0000	0x8180 FFFF
Data Flash	0x8200 0000	0x827F FFFF
WDT	0x83A0 0000	0x83A0 FFFF
PLB Timer	0x83C0 0000	0x83C0 FFFF
Main UART	0x8400 0000	0x8400 FFFF
Secondary UART	0x8420 0000	0x8420 FFFF
Debug Module	0x8440 0000	0x8440 FFFF
HWICAP	0x8600 0000	0x8600 FFFF
MATH#0	0x8700 0000	0x8700 00FF
MATH#1	0x8800 0000	0x8800 00FF
Mailbox DPRAM/registers space	0x8900 0000	0x8900 FFFF
Mailbox DPRAM/memory space	0x8910 0000	0x8910 FFFF
Configuration Memory Monitor	0x8A00 0000	0x8A00 01FF
DRM Miscellaneous	0x8A10 0000	0x8A10 00FF
DLMB BRAM Interface Controller	0x8B00 0000	0x8B00 00FF
ILMB BRAM Interface Controller	0x8B10 0000	0x8B10 00FF

Table 2: DRM system memory map

The LADAP platform has two flash memories:

- A Xilinx Platform Flash XCF128X-FTG64C, used for FPGA configuration after power-up or PROGRAM_B pin assertion. This is the CONFIGURATION FLASH in Figure 2.
- A Microchip SST39VF3201B (2 devices*, 64 Mbits = 2M x 32 bits), used to store the partial bitstreams and the block type 000 configuration frames of the initial full FPGA design. This memory will be accessed by the DRM during reprogramming of the reconfigurable partitions and during configuration memory scrubbing to correct bit errors due to SEUs. This memory is organized in 64 blocks of 128 Kbytes. This is the DATA FLASH in Figure 2.
  (*) Note: there is a third device to implement EDAC, but it is not used in this implementation.

For a scenario with two reconfigurable partitions and up to four images per partition, the data flash memory is organized in the following way:

- Full image section: it contains the block type 000 configuration frames of the initial full FPGA design. It is updated by DRM during the initialization phase when the version of the stored image is not equal to the version of the design loaded in the FPGA. Later, DRM accesses this section when it has to perform full device scrubbing or scrubbing of one frame with two bit errors.
  For a XC5VFX130T device, this image size is equal to 25,978 frames x 41 dwords/frame x 4 bytes/dword = 4,260,392 bytes.
- Partial bitstreams section: it contains the partial bitstreams. For the currently defined reconfigurable partitions, the partial bitstream size is equal to 39,436 bytes. A 128KB area (one flash memory block) has been reserved per partial bitstream. DRM will access this section to store the partial bitstreams received from LEON, to read the partial bitstream to reprogram a reconfigurable partition, and to get

the good configuration data when performing full device scrubbing or scrubbing of one frame with two bit errors.

Offset (bytes)	Content
0x000000	full device image
0x5FFFFF	
0x600000	partial bitstream 1.0
0x61FFFF	
0x620000	partial bitstream 1.1
0x63FFFF	
0x640000	partial bitstream 1.2
0x65FFFF	
0x660000	partial bitstream 1.3
0x67FFFF	
0x680000	partial bitstream 2.0
0x69FFFF	
0x6A0000	partial bitstream 2.1
0x6BFFFF	
0x6C0000	partial bitstream 2.2
0x6DFFFF	
0x6E0000	partial bitstream 2.3
0x6FFFFF	
0x700000	Reserved
0x7FFFFF	

Table 3: Data Flash memory map

The first two 32-bit words of the full device image section will be used to store:

- A key word (=0xAABBCCDD) to indicate if there is valid data in that section or not.
- Version code.

The first two 32-bit words of each partial bitstream subsection will be used to store:

- A key word (=0xAABBCCDD) to indicate if there is a valid bitstream or not.
- Partial bitstream length as number of bytes.

Each partial bitstream is identified by two digits: the first one represents the associated reconfigurable partition (starting from 1), and the second one represents the specific bitstream associated to that partition (starting from 0). For example, bitstream 1.2 will be the third bitstream of the first reconfigurable partition.

The Readback CRC logic reads a total of 26,849 configuration frames numbered from 0 to 26,848. Frame number 0 is a dummy frame. Frames with number from 1 to 25,978 are type 000 frames; those with number from 25,979 to 26,738 are type 010 frames; finally, those with number from 25,739 to 26,848 are type 011 frames. The Readback CRC logic doesn't scan type 001 frames, since they refer to BRAM content. Actually, the number of frames in the device containing configuration cells is slightly less than 26,848. This number results from the fact that there are two dummy frames whenever the readback logic starts scanning a new row.

Our design does not use type 010 and 011 frames. There is no configuration data for these frames in the full bitstream. If we read them back, we see that the 41 dwords are equal to 0x00000000. So, if scrubbing of any of these frames should be done, MicroBlaze will write all 0s.

Regarding type 000 frames, we can distinguish three ranges:

- Those with frame number from 9,439 to 9,674 belong to reconfigurable partition#2.
- Those with frame number from 12,037 to 12,272 belong to reconfigurable partition#1.
- The rest of type 000 frames belong to the static area.

When performing full device scrubbing or scrubbing of one frame with two bit errors, MicroBlaze takes into account those ranges and the image currently loaded into each reconfigurable partition, in order to know the flash address(es) it has to read from.

2.1.3 LEON-DRM Communication Protocol

The final hardware implementation details of the Mailbox DPRAM are described in D4.9 [1]. The defined custom communication protocol was presented in D4.7. Since the first specification, some messages have been updated and the status error codes in the answer messages have been defined. The message format has not changed. The final implemented messages are described next.

2.1.3.1 Status commands

- **Status (0x00):** this is a keep-alive check of the DRM.

Syntax	Length (bytes)	Value
Request		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x00
COMMAND_ID	1	0x00
END	1	0x03
Answer		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x01
COMMAND_ID	1	0x00
STATUS	1	0: OK, command execute correctly 1: Error: wrong command request length
END	1	0x03

- **Error Answer Message (0xFF):** this message is only defined for answers. It is sent instead of the expected answer type when the command request is erroneous.

Syntax	Length (bytes)	Value
Answer		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x01
COMMAND_ID	1	0xFF
STATUS	1	1: Start byte not found in the command request 2: Value of LENGTH field in the command request greater than allowed maximum 3: Unknown COMMAND_ID 4: End byte not found in the command request
END	1	0x03

2.1.3.2 Reconfiguration Control commands

- **Store Partial Bitstream (0x10):** this command requests the DRM to store a new partial bitstream (included in the message) in the data flash memory. As a partial bitstream is too large to be stored in the mailbox DPRAM, it will be sent in multiple messages. The bitstreams will be sent divided in 2 Kbytes segments (the last segment can be shorter). Upon reception, integrity check will be performed for each segment; if an error is detected in one of them, the DRM answers with an error message and the LEON processor has to send the segment again.

Syntax	Length (bytes)	Value
Request		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	MSB Bit: '1' for all messages but for the last one Bits (14-0): 0x08 0x02 for all messages but for the last one
COMMAND_ID	1	0x10

RM ID	1	Identifier of the reconfigurable partition associated to the partial bitstream
BITSTREAM ID	1	Bitstream Identifier
BITSTREAM	variable	New bitstream data
END	1	0x03
Answer		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x01
COMMAND_ID	1	0x10
STATUS	1	0: OK, command execute correctly 1: Bitstream integrity error 2: Bitstream size error 3: Flash delete failure 4: Flash recording failure 5: Length mismatch between message length and bitstream data length 6: Invalid RM ID 7: Invalid BITSTREAM ID 8: Unexpected bistream data packet number 9: Unexpected RM ID value 10: Unexpected BITSTREAM ID value …
END	1	0x03

- **Load Partial Bitstream (0x11):** this command requests the DRM to reconfigure some area of the FPGA with a previously stored partial bitstream.

Syntax	Length (bytes)	Value
Request		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x02
COMMAND_ID	1	0x11
RM ID	1	Identifier of the reconfigurable partition associated to the partial bitstream
BITSTREAM ID	1	Bitstream Identifier
END	1	0x03
Answer		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x01
COMMAND_ID	1	0x11
STATUS	1	0: OK, command execute correctly 1: Wrong command request length 2: Invalid RM ID 3: Invalid BITSTREAM ID 4: Key word not found 5: Error writing to ICAP
END	1	0x03

- **Request Loaded Bitstreams (0x12):** this command requests for the number and ID of the non-black box loaded partial bitstreams.

Syntax	Length (bytes)	Value
Request		
START	1	0x02
NOT LAST MESSAGE +	2	0x00 0x00

LENGTH		
COMMAND_ID	1	0x12
END	1	0x03
Answer		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	Not Last Message bit = '0' Length = 1 + 2 * (number of RMs) when STATUS = 0 Length = 1 when STATUS ≠ 0
COMMAND_ID	1	0x12
STATUS	1	0: Ok, command executed successfully 1: Wrong command request length
LOOP (for number of RMs) – content is present only when STATUS = 0		
RM ID	1	Reconfigurable Area Identifier
BITSTREAM ID	1	Bitstream Identifier: (0 for blackbox)
END LOOP		
END	1	0x03

2.1.3.3 SEU Mitigation Control and Monitoring commands

- **Retrieve SEU statistics (0x20):** this command requests for statistics about SEU detection and correction in the Virtex-5 device.

Syntax	Length (bytes)	Value
Request		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x00
COMMAND_ID	1	0x20
END	1	0x03
Answer		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x0B when STATUS = 0 0x00 0x01 when STATUS ≠ 0
COMMAND_ID	1	0x20
STATUS	1	0: OK, command executed successfully 1: Wrong command request length
Single-Error Configuration Memory Frames	1	Number of corrupted frames with one bit error found during readback
Double-Error Configuration Memory Frames	1	Number of corrupted frames with two bit errors found during readback
Full Scrubbing Processes	1	Total number of full scrubbing processes required because of unidentified uncorrectable bit errors
Periodical Full Device Scrubbing Processes	1	Number of full device scrubbing processes performed when periodical blind scrubbing is enabled
DP1SBitErrCnt	1	Number of single bit errors detected when reading from the mailbox DPRAM where LEON stores the command requests
DP1DBitErrCnt	1	Number of double bit errors detected when reading from the mailbox DPRAM where LEON stores the command requests
DP2SBitErrCnt	1	Number of single bit errors detected when reading from the mailbox DPRAM where DRM writes the command answers
DP2DBitErrCnt	1	Number of double bit errors detected when reading from the mailbox DPRAM where DRM writes the command answers
MbBramCECnt	1	Total number of correctable single bit errors detected and corrected in the MicroBlaze LMB BRAM memory
MbBramUECnt	1	Number of uncorrectable errors detected in the MicroBlaze LMB BRAM memory
END	1	0x03

- **Configure SEU mitigation (0x21):** this command configures SEU detection and correction parameters.

Syntax	Length (bytes)	Value
Request		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x04
COMMAND_ID	1	0x21
MASK	1	Bits 7 – 3: reserved for future use Bit 2: mask bit associated to DRM_CONTROL bit 1 Bit 1: mask bit associated to DRM_CONTROL bit 0 Bit 0: mask bit associated to PERIODIC FULL SCRUBBING When a mask bit is 0, the associated field/bit can be ignored. When a mask bit is 1, the associated field/bit has valid content and DRM must apply it.
PERIODIC FULL SCRUBBING	2	Bit 15: Enable/Disable periodic blind full device scrubbing Bit 14-0: Period in seconds (time unit in seconds only for evaluation purposes; for a final system it should be in hours or days) Note: for implementation easiness, this timer is also used as the indication to perform periodic scrubbing of the entire MicroBlaze LMB BRAM and all MicroBlaze internal BRAMs.
DRM_CONTROL	1	Bits 7 – 2: reserved for future use Bit 1: Enable (1) or Disable (0) DRM's Internal WDT Bit 0: 1 => command a reset of the DRM; 0 => do nothing
END	1	0x03
Answer		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x01
COMMAND_ID	1	0x21
STATUS	1	0: OK, command execute correctly 1: Wrong command request length
END	1	0x03

2.1.3.4 Peripheral Control commands

- **Perform Operation (0x30):** this command provides the two 32-bit operands for the mathematical peripheral and returns its result.

Syntax	Length (bytes)	Value
Request		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x09
COMMAND_ID	1	0x30
RM ID	1	Identifier of the reconfigurable partition to interact with
Operand 1	4	First operand
Operand 2	4	Second operand
END	1	0x03
Answer		
START	1	0x02
NOT LAST MESSAGE + LENGTH	2	0x00 0x05 when STATUS = 0 0x00 0x01 when STATUS ≠ 0
COMMAND_ID	1	0x30
STATUS	1	0: OK, command executed successfully 1: Wrong command request length 2: Invalid RM ID

RESULT	4	Result of the operation (only present when STATUS = 0)
END	1	0x03

2.1.4 LEON2-FT processor and Watchdog Timer module

The hardware system developed for the RTAX device has the required features of fault-tolerance to warranty reliability and self-healing mechanisms. Basically, it is formed by a LEON2-FT processor, hardened with fault-tolerance characteristics which provide greater robustness, and a Watchdog Timer (WDT), an autonomous module that monitors continuously the status of scrubbing processes in Virtex-5, acting accordingly.

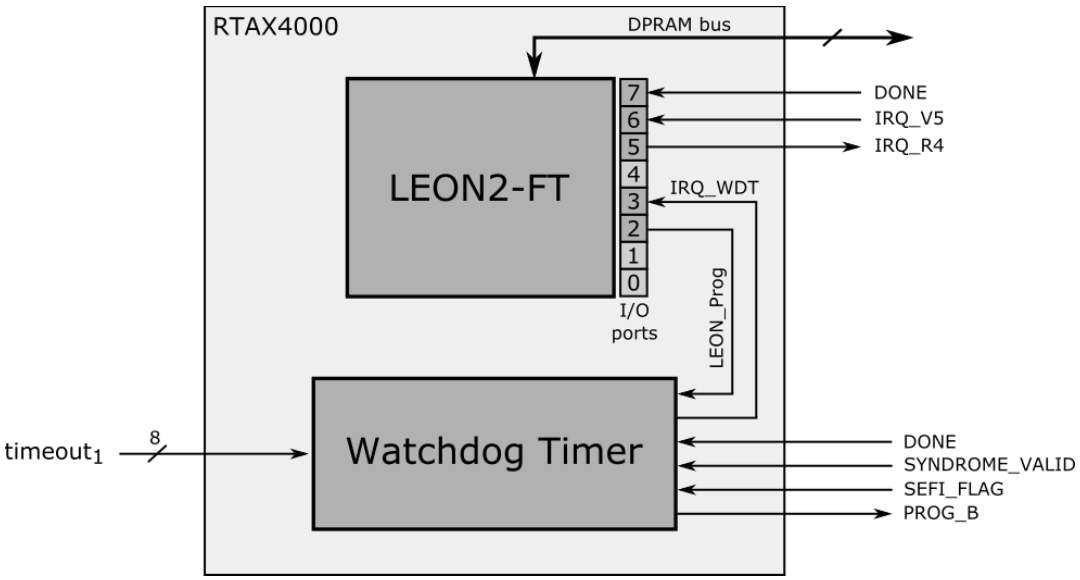


Figure 3: Modules implemented in RTAX4000 device

As a result of the scrubbing performed in Virtex-5 some signals are asserted: SYNDROME_VALID from FRAME_ECC_VIRTEX5 primitive, which reports integrity of configuration frames during Readback, and SEFI_FLAG from MicroBlaze that informs when SEFIs are detected. A brief description of signals used in this system is shown in Table 4.

Table 4: Signals interfaces in RTAX4000 device

Signal	Interface	Description
DPRAM bus	RTAX-V5	Control and data signals to manage DPRAM memory: clk, address[11:0], data[15:0], chip select (cs), write enable (we), output enable (oe)
IRQ_V5	RTAX-V5	Interrupt sent from Virtex-5. It notifies that LEON processor can access to DPRAM memory to process new data
IRQ_V4	RTAX-V5	Interrupt sent from LEON. It notifies that MicroBlaze can access to DPRAM memory to process new data
DONE	RTAX-V5	Configuration status of Virtex-5
PROG_B	RTAX-V5	Programming signal for Virtex-5
SYNDROME_VALID	RTAX-V5	Active one cycle for each uncorrupted frame during a readback operation (signal associated with FRAME_ECC_VIRTEX5 primitive)
SEFI_FLAG	RTAX-V5	SEFI indicator sent by MicroBlaze
IRQ_WDT	LEON-WDT	Interrupt sent from WDT to LEON warning that a reconfiguration of Virtex-5 has been done
LEON_Prog	LEON_WDT	Direct order from LEON processor to reprogram Virtex-5

2.1.5 **LEON2-FT processor**

Systems of each device are communicated through a Dual-Port RAM (DPRAM) memory, implemented in the Virtex-5, acting as Mailbox between them. Each one has a reserved area to avoid conflicts during memory accesses. Interrupts are used to manage a synchronize access from both sides, notifying the other device that data are ready to be processed after finishing the write process.

Additionally, there is a communication interface between LEON and Watchdog Timer which allows the processor to have direct access to reprogram Virtex-5. Some capabilities are possible through I/O ports: assert PROG_B signal for Virtex-5 device, manage interrupt sent from WDT and analyze Virtex-5 DONE signal.

2.1.6 **Watchdog Timer**

This custom module designed by TASE will check SYNDROME_VALID, SEFI_FLAG and DONE signals coming from Virtex-5 to verify the integrity of its configuration memory. In case of failure, it is in charge of total reconfiguration of Virtex-5 asserting PROG_B signal.

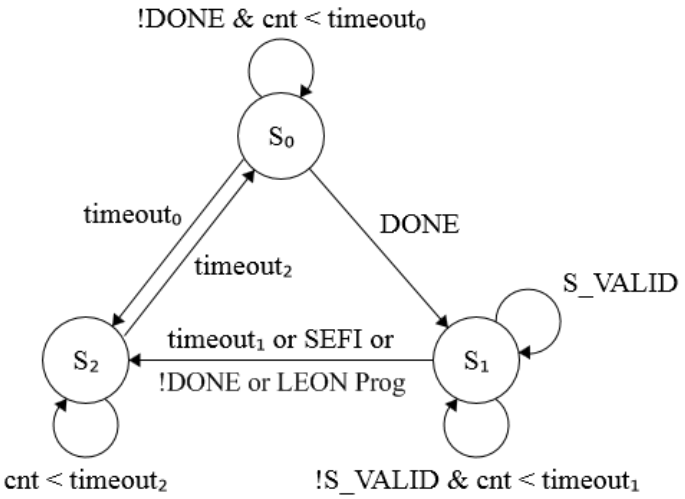


Figure 4: FSM diagram of Watchdog Timer

The implementation of the Watchdog module is based on a 3-states FSM (Figure 4) and a configurable timer in each one. After each programming of Virtex-5 (S_0), WDT waits for assertion of DONE signal with a predetermined $timeout_0$ (See Table 5). If configuration is successful, WDT passes to a normal working state (S_1) waiting for the corresponding $timeout_1$ of the periodic SYNDROME_VALID signal. If counter reaches timeouts, or SEFI_FLAG signal is asserted, or DONE signal is deasserted, or a programming order comes from LEON, WDT enters in a programming state (S_2) during a period fixed by $timeout_2$. After this time, WDT comes back to S_0 waiting for Virtex-5 gets programmed.

Table 5: Timeout's values used in each FSM state

Timeout	Time	Comments
Timeout₀	160 ms	Maximum Configuration Time for XC5VFX130T (48 MHz): 119.1 ms
Timeout₁	10 ms – 2,56 s	Externally configurable using R4_SPEC_IO[7:0] pins. For real-time systems it will be managed by LEON processor.
Timeout₂	280 ns	Minimum Program Pulse Width: 250 ns

LEON is informed by an interrupt every time WDT leaves S_0 state. Hence the processor always know when a full reconfiguration has been performed and must check the status of DONE signal to find out if the result of that process has been successful or not.

2.1.7 **Validation design**

A test design have been developed to validate previous functionalities. This design (Figure 5), which has been implemented in Virtex-5 FPGA, contains a FSM that controls a DPRAM memory and simulates the behavior of the MicroBlaze, and a Frame ECC emulator of Virtex-5.

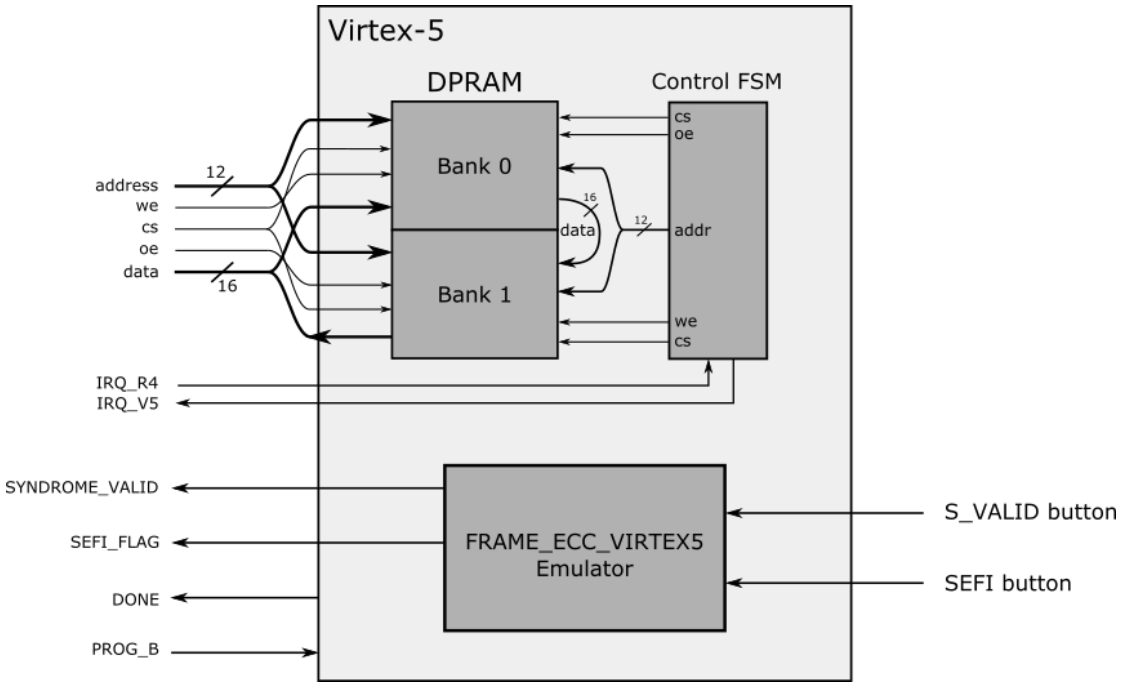


Figure 5: Modules implemented in Virtex-5 to validate the design

The goal of the first test is to validate correct access to Dual-Port RAM from RTAX device. For this purpose, two DPRAM are implemented with a control FSM. This state machine acts the same as the MicroBlaze: it attends the interrupt when LEON stops writing, then it processes the information and finally it notices LEON when have finished. In this case, to copy the content from one memory to the other one is the data processing.

Externally, LEON will write and read from the same addresses, however the aforementioned procedure takes place inside the Virtex-5. To validate this test, LEON must read the same data it wrote previously, after sending the corresponding interrupt.

The purpose of the second test is to validate the self-healing capabilities that provides the Watchdog module. To emulate SYNDROME_VALID signal like FRAME_ECC_VIRTEX5 logic would do is the methodology used. A periodic signal with pulse width of 0.48 s. and non-active during 1.55 s. (time for *full device scrubbing*, the worst case) has been chosen to make easier its visualization. Furthermore, it is possible to handle SYNDROME_VALID and SEFI_FLAG signals using the push buttons of the platform. SYNDROME_VALID is non-active while pressing PUSH2 button, and PUSH3 emulates a SEFI in the system.

Table 6 shows the signals assignation to LEDs and buttons of the LADAP platform.

2.2 Fault-Tolerant network for distributed space environment

The fault-tolerant network for distributed space applications is provided by the TTEthernet network as described in deliverable D9.3 (Chapter 2 Use case MPSoC Hardware for Space).

An Ethernet network, and thus also a TTEthernet network, consists of two main types of components:

- End-systems (also referred to as Network Interface Component, NIC or node) are senders and/or receivers of communication
- Switches (also referred to as bridges) facilitate the forwarding of the communication.

The connection between switches and end-systems is depicted in Figure 6. In the figure, the network is depicted without redundant components, thus providing a simple network that connects four nodes together.

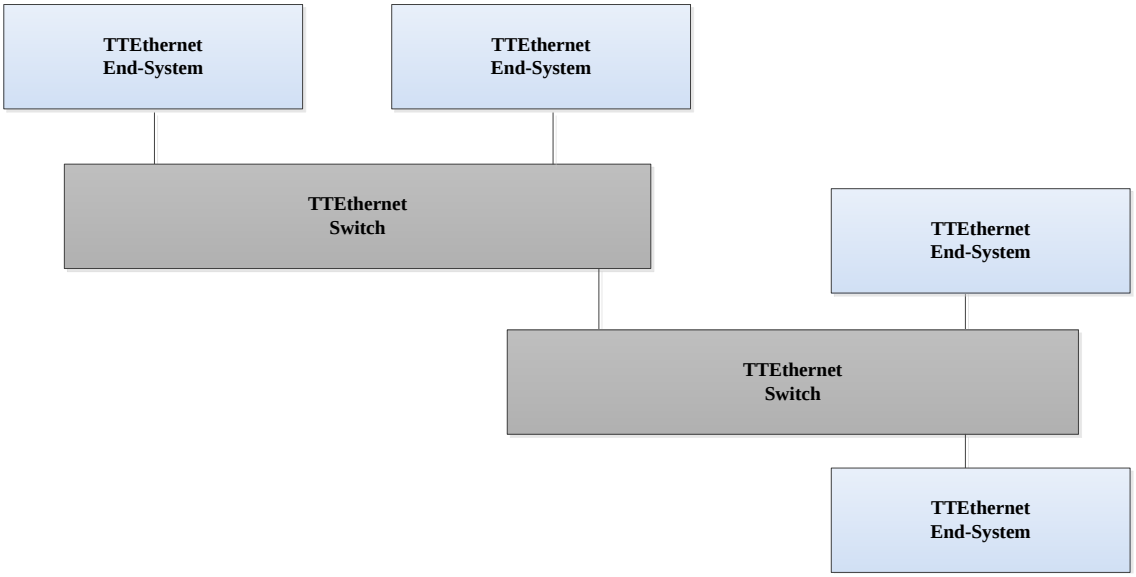


Figure 6: Simple TTEthernet network with 2 switches and 4 end-systems

In practice, in a redundant architecture used for critical space applications, the TTEthernet switches are replicated in order to provide redundant communication paths between end systems. This implies that each end-system hosts multiple Ethernet ports and frames are replicated for each port, sent redundantly across the network, and received by a voter at the receiving side.
The work in EMC2 has focused specifically also on end-system implementation of IP solution for space applications. The TTEthernet End System controller is a digital component and does not include any analog functionality. The End System is capable of up to three Ethernet ports. The base frequency of the End System Controller is 12,5MHz for the use of the Ethernet ports with 10/100Mbps speed or 125MHz for 10/100/1000Mbps. The standard interface of these ports is GMII in 125MHz mode or MII in 12,5MHz mode. This has been implemented on the Xilinx Virtex platform (Virtex-5 and Virtex-7 families).
The schematic overview of the TTEthernet End System Controller is depicted in Figure 8, providing also an overview of the main services at the receiving and sending sides:

- RX: Frame reception (integrity checking and redundancy management, port and memory mapping)
- TX: Frame transmission (traffic shaping, scheduled message transmission, port and memory mapping)
- Clock synchronisation (distributed clock synchronisation according to the SAE AS6802 service)
- Status and control and diagnosis interfaces

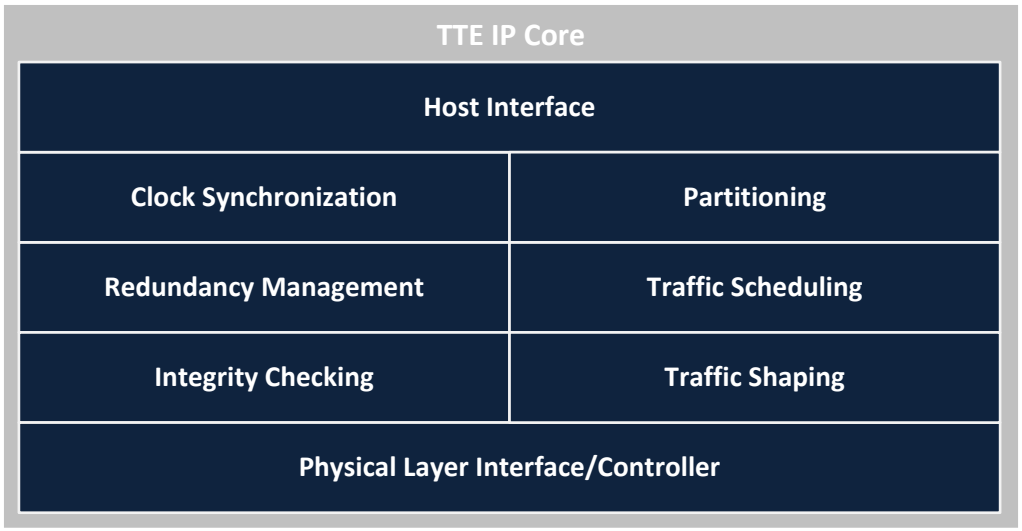


Figure 7: Services of the TTEthernet network controller

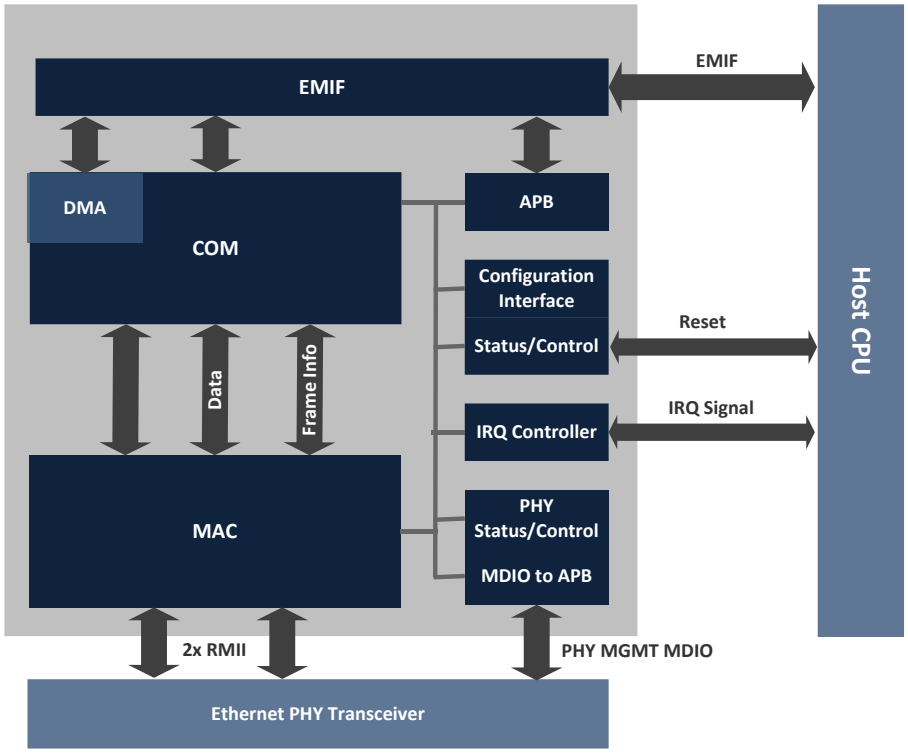


Figure 8: Schematic overview of the Network ES Controller

A second aspect has related to the SW integration of the TTEthernet controller into the partitioned environment:

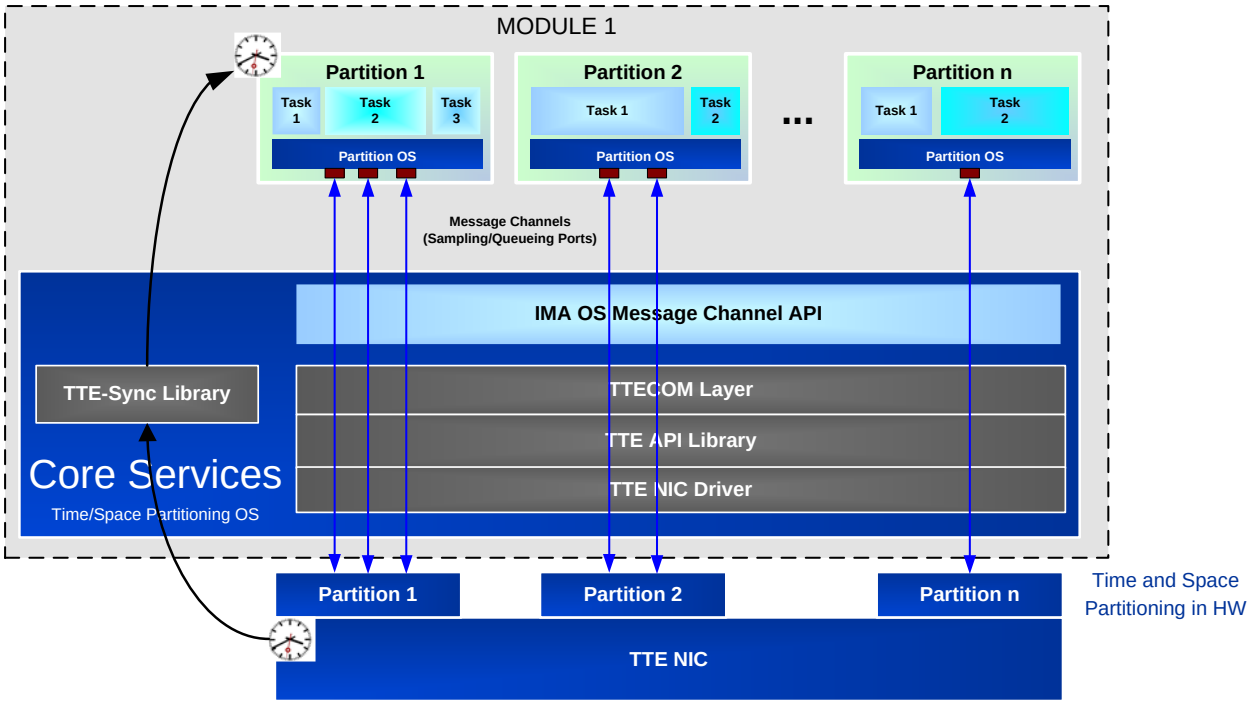


Figure 9: TTEthernet SW stack integration

The software library can integrate by means of the TTE-Sync library to the partitions of the overlaying operating system or hypervisor by means of the Core Services through a time/space partitioning operating system. The components developed by TTT are the driver for the network controller, the API layer and the COM wrapping layer providing the mapping between the physical ports, the logical ports and the application ports mapped to the operating system and application API provided by it (Figure 9).

When connecting these nodes together, a networked integrated modular architecture approach as depicted in Figure 10 is created.

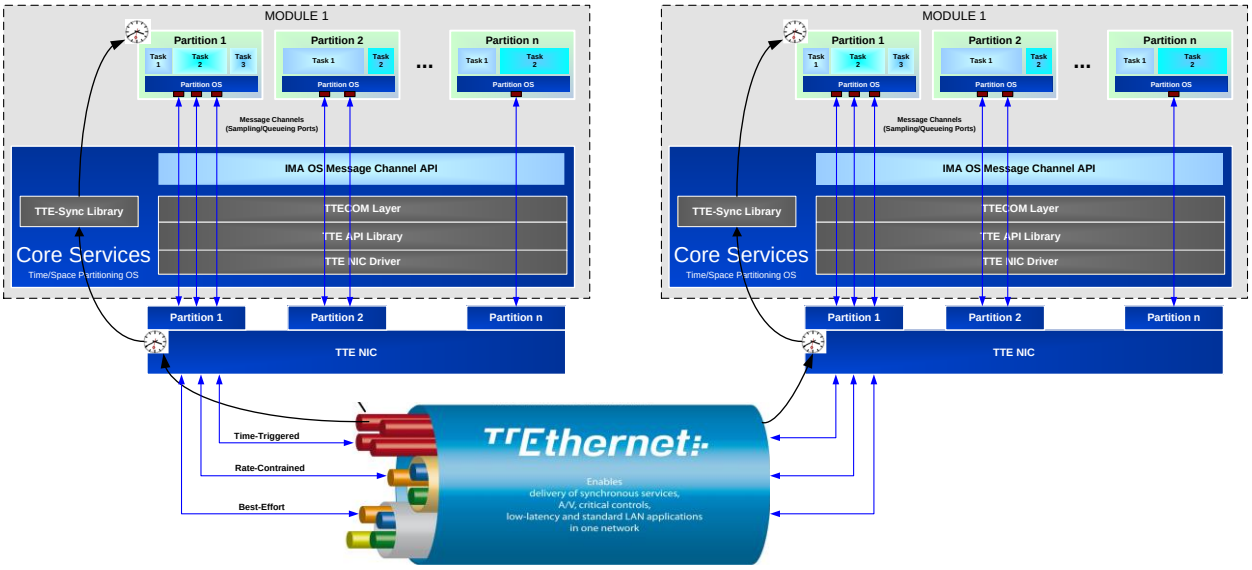


Figure 10: Distributed partitioned architecture

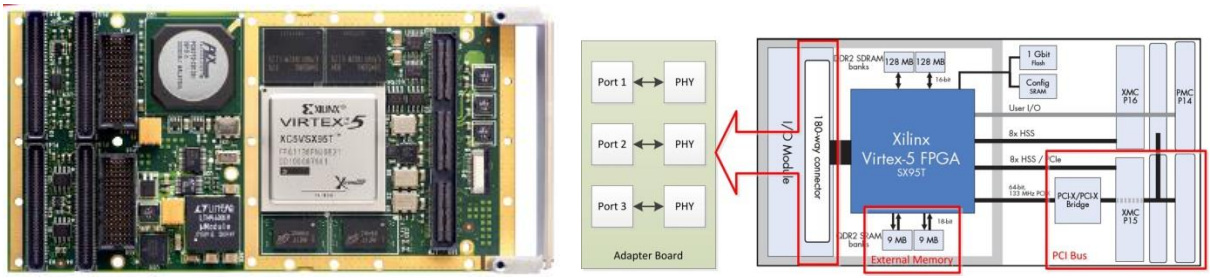


Figure 11: ES Virtex-5 evaluation implementation on XMC-FPGA05D

The IP has been first integrated on the Xilinx Virtex-5 platform using an evaluation board provided by Curtis Wright (XMC-FPGA05D) with an hardware adapter board providing the respective three-port Ethernet PHYs for providing the necessary redundancy.

Table 6: LEDs and buttons used in the validation design

Element	Signal
LED1	Interrupt from LEON
LED2	Interrupt from VIRTEX-5
LED3	Chip Select (DPRAM)
LED4	Write Enable (DPRAM)
LED5	SYNDROME_VALID
LED6	SEFI_FLAG
PUSH0	Virtex-5 Reset
PUSH2	Deassert SYNDROME_VALID
PUSH3	Assert SEFI_FLAG

2.3 Accurate host-compiled simulation of LEON3 with VIPPE tool

2.3.1 Introduction

Nowadays, virtual platform models are quite important in embedded system development. Traditionally, embedded software development and verification was performed by running code on a prototype of the hardware platform. However, this solution is only available late in the design process, and it is typically unreliable in terms of quality and hard to use, especially as the complexity of embedded systems grows.

Virtual platforms offer a powerful alternative to hardware prototypes. Instead of requiring a real implementation, software models of the key components of a processor platform are combined to form an executable sub-system where the application SW can be run. These models have enough functionality to execute the code correctly, but retain a level of abstraction that reduce the time required to build an executable platform and dramatically increase the possibility of evaluating multiple HW alternatives.

As a result, virtual platform models have the following advantages:

- Enable early SW development: As these models are available earlier than their hardware equivalent, embedded software development can start sooner, reducing product's time-to-market.
- Increase system controllability and simplify debugging: virtual platforms enable accessing the internal information of platform components, which makes HW/SW analysis and debug much easier than in real HW platforms, where there is no opportunity to change or control the hardware or software execution.

- Provide good performance and accessibility: Hardware platforms often have limited availability during early production stages, which limits its use. Furthermore, if constructed carefully, virtual platforms can execute faster than the actual final hardware, allowing extended evaluation and testing steps.

Several alternatives have been proposed to simulate processor operation in order to create virtual platforms. These alternatives propose working at different levels of abstraction, providing different tradeoffs in terms of accuracy vs. performance and usability.

Currently, host-compiled simulation is the state-of-the-art solution that provides best performance and easier use. This technique is based on static annotation of the original SW source code with the expected performance it would have in the target platform. After that, compilation and execution of the annotated code in the host computer is done.

This approach has as theoretical maximum speed the native execution of the original code, which is the maximum speed possible for a functional simulation. Furthermore, it does not require porting the SW (such as the compiler or device drivers) to the target platform. This possibility enables easy exploration of multiple design alternatives with minimum effort. However, its accuracy is still lower than in other techniques, begin an active research area.

In this project, the University of Cantabria, in collaboration with TASE has worked in a novel solution to increase host-compiled simulation accuracy. It has been applied to simulate and estimate the expected performance of SW applications running on top of different LEON3 platforms.

2.3.2 **Work done**

The work performed in this project has focused on the extension of a previous tool developed by the University of Cantabria, called VIPPE. VIPPE is a tool oriented to create virtual platforms based on host-compiled simulation. It can model multi-core platforms, including SystemC peripherals, and integrates an abstract model of the Operating System, providing several APIs, such as POSIX, which enables to simulate embedded Linux platforms. In this project, the tool has been improved with a new annotation engine and the internal changes required in the tool to use it have also been made.

Typically, there are two solutions to estimate the performance of a basic block of SW code in the target platform, which is the main information required for the annotation:

- cross-compile for the target platform, correlating the blocks of the resulting assembler with the basic blocks in the original code
- generate a sort of compiler back-end that generates the information from the compiler intermediate code, as VIPPE do.

The first alternative has several problems with compiler's optimizations, since optimizations can reorganize the code structure, making impossible a full relationship between the source and the assembler code. The second option has difficulties since details of the real back-end are not considered in the estimation.

The solution proposed in this work combines both approaches. The original code is cross-compiled to obtain accurate annotations, but the relationship is not done with the original source code, but with the intermediate code generated by the compiler. As a result, the annotation is performed at the compiler intermediate level. The benefit obtained from working at the intermediate level results from the fact that compilers' back-ends do not dramatically modify the code structure when generating the assembler from the intermediate code. Modifications done by the real back-end can be easily considered, enabling adequate annotation of the host-compiled code.

To implement this approach, the solution applied is based on the use of the LLVM compiler (clang), since it was the compiler used in the previous version of VIPPE. Moreover, the access of intermediate code is easier in LLVM than in GNU/GCC, which makes feasible to demonstrate the possibilities of the approach with the effort available in the project.

2.3.2.1 Annotation steps

To annotate the source code, the new annotation engine developed in the project performs the following steps:
1.- Pre-process the code with the native headers
2.- Cross-compile the pre-processed code for the target platform (LEON3), and obtain the intermediate code
3.- Obtain the assembler for the target processor (LEON3)
4.- Estimate the number of cycles required to execute each basic block of the assembler code in the target platform.
5.- Annotate the intermediate code with the estimated cycles, together with cache operation annotation.
6.- Compile the annotated intermediate code for native execution, together with VIPPE simulation libraries.
7.- Run in the host computer and obtain results

2.3.2.2 Relationship between assembler and intermediate code

Although assembler and intermediate code structures are similar, since compiler back-end typically perform minor changes to then, there are some differences that have to be managed to perform correct annotations.
Basically, modifications that appear in the structure of the assembler code are the result of the following elements:

- Compiler back-end can reorder the blocks of “if” structures, changing the comparison and the order of the then/else statements.
- Compiler back-end change “switch” structures in the intermediate code by a list of chained “if” structures.
- Intermediate code can contain “select” clauses that are implemented as “if” structures, resulting in several additional basic blocks.
- Compiler back-end change “if” conditions created by merging several conditionals with “and”/”or”/”xor” operations by multiple chained “if” structures
- Compiler back-end has freedom to use “return” statements that identify the end of code functions, reorganizing the last basic blocks of the function codes as required to simplify the resulting codes

To overcome these differences, the annotator developed has the capability of analyzing the new basic blocks added by the back-end. Then, it modifies the intermediate code before native compilation in two ways:

- when modifications in the code structure are minimal (typically a new single if statement), conditional annotations are added in the original basic blocks.
- otherwise, the original intermediate code is modified replicating the structure of the assembler code. Thus, functionality is unchanged but annotation is performed with minimal overhead.

2.3.3 **Modeling of LEON3 internal details**

Together with the development of the process need to extract the relationship between assembler and intermediate code, which is mainly generic for any processor, the modeling of LEON3-based platforms also requires integrating LEON3 specific details within the annotation tool.
The details that has been evaluated and integrated are the following:

- Assembler instruction identification: The first step of the annotation process is to know the type of each assembler instruction in the code from its opcode: operators, branches, memory access, …
- Clock cycles required for each assembler instruction: since not all assembler instructions can be executed in one cycle, it is required to obtain the overall number of cycles required by each instruction before basic block annotation
- Pipe dependencies: data dependencies among near instructions provoke processor stalls, stopping a different number of cycles depending the distance and the type of instruction.
- Branch stalls: Depending on the result of the condition in branches, jumps can result in stalls or not, involving different number of cycles to execute
- Write buffer delay: By default, LEON3 has a write buffer of 3 positions to perform store instructions. When the buffer is full, the processor stalls.

Moreover, estimations of the overhead resulting from the use of LINUX in the LEON3 platform have been done, feeding the simulation engine with the resulting values.

2.3.4 **Comparison with the real board**

To evaluate the accuracy of the tool developed, a real implementation has been used. For such purpose, a XILINX ML506 platform (Virtex 5) board has been used. This board enables integrating platforms with 1 and 2 LEON3 cores. A “FPU-lite” unit has been added to perform floating point instructions.

Regarding the SW platform, both comparisons in bare metal, and with Linuxbuild OS have been performed.

Since the LLVM has a port for “sparc v8” but not specifically for LEON3, some problems with the assembling step have appeared when generating executables for the target board. To solve them, the assembly code obtained with LLVM was transformed into binary with the corresponding GNU tool (“as”) instead of with the LLVM tool.

To obtain information about number of cycles and instructions in the real platform, the peripheral “l3stat” provided in the standard distribution of LEON3 VHDL codes (version 1.4 onwards) has been used:

- When operating in bare metal, a new “main” function designed to wrap the application main function has been created. This new function is in charge of activating and printing the counters before and after the execution of the original “main” by reading/writing the registers of the peripheral.
- When operating under Linux, the wrapper has been modified to use the standard “devmem2” command to access the l3stat registers.

Both operations add some overhead, since counter activation and deactivation take some cycles. To solve this problem, the overhead has been measured using an empty “main” function. Then, this value is subtracted from the real measures before putting them in the tables below. In the case of 2 cores with OS, the overhead is not constant, and thus values measured has some small uncertainty. This uncertainty can be estimated comparing the execution time of both cores, which should be the same.
In order to check and demonstrate the tool, some small applications and a CCSDS 122 has been executed both in the real platform and in the VIPPE simulator.

Results for small examples in bare metal can be found in the following tables:

Application	Parameter	Real Board	VIPPE	Error (%)
Bubble:	# Instructions	5007000,0	5007002,0	0,0
	# Cycles	6512362,0	6510066,0	0,0
	Time(ms)	108,5	108,5	0,0
Factorial:	# Instructions	627260,0	627264,0	0,0
	# Cycles	631778,0	631812,0	0,0
	Time(ms)	10,5	10,5	0,0
Hanoi:	# Instructions	7520477,0	7602295,0	-1,1
	# Cycles	10300486,0	10224023,0	0,7
	Time(ms)	171,7	170,4	0,7
Queens:	# Instructions	20642953,0	20657162,0	-0,1
	# Cycles	27357936,0	26965264,0	1,4
	Time(ms)	456,0	449,4	1,4

As it can be seen, accuracy of the developed tool for the checked small examples is really high, resulting in errors less than 1.5 %.

For the CCSDS example, estimation of execution times with input images of 200x200 and 1000x1000 pixels have been compared running under Linux OS, with a real platform containing 1 and 2 cores. The application SW has only 1 thread, so no parallelism can be obtained in the case of 2 cores, but different overhead due to the OS is found.

In the case of 2 cores, the mapping of tasks to cores change on each execution, so instructions and cycles on each processor cannot be compared separately. In this case, comparison is only possible analyzing the overall execution time.

Image Size/ Cores	Parameter	Real board CPU 0	Real board CPU 1	VIPPE CPU 0	VIPPE CPU 1	Error (%)
200x200	# Instructions	18728649,0		18019820,0		3,8
1 core	# Cycles	33774199,0		31839385,0		5,7
	Time(ms)	562,9		542,4		3,6
200x200	# Instructions	1885836	17523978	17963948	55872	N.A.
2 cores	# Cycles	34381362	34833050	31695156	124019	N.A.
	Time(ms)	573,0	580,6	541,9	541,9	5
1000x1000	# Instructions	417138440		413174727		0,9
1 core	# Cycles	816991237		745355733		8,7
	Time(ms)	13616,5		12731,27		6,5
1000x1000	# Instructions	387605066	39716098	413141813	32904	N.A.
2 cores	# Cycles	827185646	829349583	745636554	61417	N.A.
	Time(ms)	13786,4	13822,5	12737,7	12737,7	7,6

As the table shows, when considering a large example the accuracy of the estimation tool is still quite good since all errors are below 10%.

2.3.5 **Following steps**

Once the estimation tool has been extended to provide accurate results, it has to be used to analyze different design alternatives.

In this context, the next step is to parallelize the application and obtain results with different number of threads for the parallelization.

Additionally, some SW code modification that can impact on application performance will be evaluated. Finally, other examples, such as an AES, will be also evaluated and explored.

3. Use Case MPSoC Software and Tools for Space

3.1 Art2kitekt introduction

The tool suite art2kitekt has been used to design and evaluate a space domain application. Both software and hardware elements have been modeled and analized with the tool.

First of all, a multi core system from the space domain context based on the GR712RC Dual-Core LEON3-FT board has been modeled for evaluation purposes. Then, an example of a "satellite" task set has been also modeled. Afterthat, taking advantage of the "Response Time Analysis with automatic processor allocation" offered by the System Analysis stage, tasks and flows have been mapped to the corresponding cores in the execution platform in a way that system feasibility is guaranteed. Finally, with the newly available Code Generation stage, a low-level skeleton of the source code is automatically generated and prepared to run in a RTEMS operating system.

In tight collaboration with TASE, the tool suite art2kitekt has been developed and adapted to guide the engineer through the steps of system design and analysis for a space domain task. Following an agile methodology for the tool suite art2kitekt software development, since march of this year, which was the deadline for deliverable D9.4, three internal releases (including several targets each one) have been reached:

- **16.04 – 6th internal "release" (2016/04/01)**
  - Application management: Improved project isolation by means of group and user attributes as a system of project access permission.
  - System Analysis stage: Added algorithms to compute CPU utilization (CPU) and best response time (BRT).
  - Application Model & System Analysis stages: Extended Response Time Analysis algorithm to provide support for OFFSETS and JITTER.
  - System Analysis stage: Added division of algorithms among three categories according to <Processor allocation>, <Feasibility analysis> and <Resource utilization> criteria.

- **16.07 – 7th internal "release": (2016/07/01)**
  - System Analysis stage: Improvements in the feedback framework, providing hints to solve design flaws found in the results by the analysis algorithms.
  - System Analysis stage: Achieved an isolated demo with SENSITIVITY analysis, including a new graphical representation of the task schedule.
  - System Analysis stage: Added WORST FIT analysis algorithm.
  - Application management: Improved functionality to access the application with user/password policy.

- **16.10 – 8th internal "release: (2016/09/30)**
  - New stage: Released proof of concept for the last developed stage "Code Generation".
  - Stage Code Generation: Preliminary functionality available to automatically generate the skeleton source code in C language for a POSIX platform.
  - Application Model: Implemented the required user interface functionality to allow a manual mapping of tasks to processors in order to allow for the analysis algorithms the scheduling of tasks in an multi-core execution platform.

Applications Detailed Description

3.2 Description of the UC9.2 and WP2.4 tool application

From WP02 “Executable Application Models and Design Tools for Mixed-Critical, Multi-Core Embedded Systems”, and specifically from Task 2.4 “Code generation and offline analysis tools”, the new tool suite art2kitekt has been further developed and adapted with UC 9.2 and thus to space domain requirements. Once more, valuable feedback has been provided to the art2kitekt tool suite development in different aspects as analysis algorithms requirements, usability tips, and source code examples to automate code generation.

Use Case 9.2 demonstrator is currently using a specific execution platform model based in the features of the GR712RC Dual-Core LEON3-FT board, as well as a manually written example of RTEMS code in order to help with the ongoing code generation functionality which will be soon available with the next relases of the tool suite scheduled before the end of EMC2 project.

3.3 Detailed description of the application and the use case

Execution platform and the application software have been modeled in their corresponding stages, manually mapping some flows to specific processors. Different versions of the Response Time Analysis algorithm are provided in order to balance load among processors, or to minimize the number of required processors.

If a design error is found with a given analysis, the tool suite will suggest a change in the application model parameters in order to achieve a feasible task set for the current execution platform and application model co-design.

3.3.1 Platform Model description

A reduced set of hardware devices and its corresponding parameters can be modeled for each execution platform at the so called Platform Model stage. A compact graphical representation of the modeled platform is shown to the user as an intuitive linked graph. This functionality is shown at the “Board Model“ panel and is aimed to easily found any hardware component incompletly defined, with no link to other components.

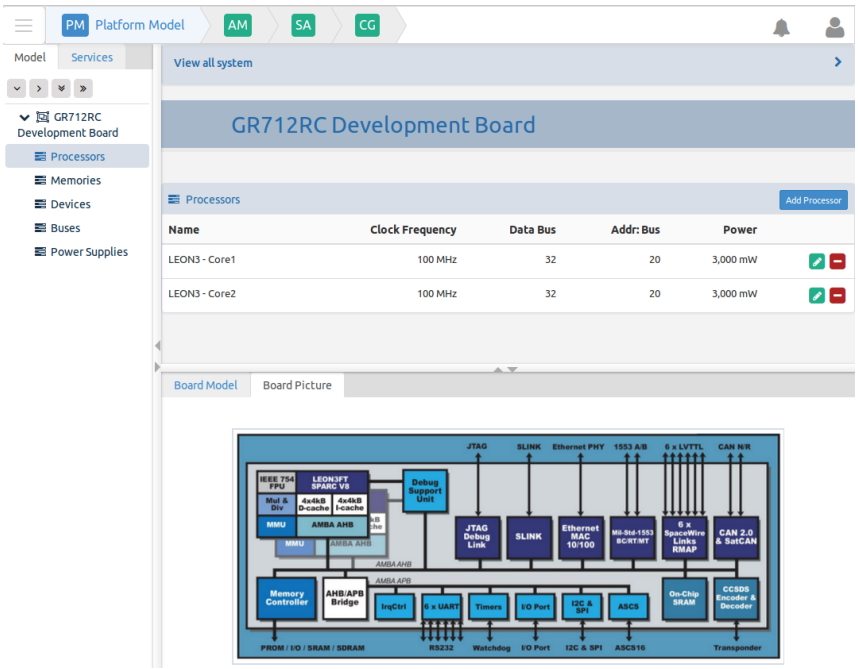


Figure 12: GR712RC platform model

Morevover, there is also a "Board Picture" panel which can be used to import a block diagram of the execution platform which will be used as a reference to the execution platform definition.

3.3.2 Application Model description

Systems, Subsystems, Flows and Tasks can be hierarchically defined in order to model a task set at the Application Model stage. As an example, a snapshot of a flow defined to model the UART utilization into the "Satellite" Use Case it is shown, where several flow parameters have been defined as <Period>, <Deadline> and <Priority>. Besides, a manual mapping of the <Processor> has been assigned.

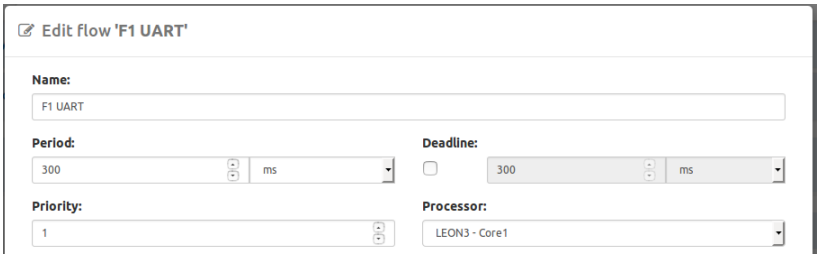


Figure 13: Data modeled for the first flow of the task set

Next, a figure with the flow description introduced at the graphical user interface of the art2kitekt suite it is shown. This info description can be display at any moment just by clicking the "i" button available for every item documented.

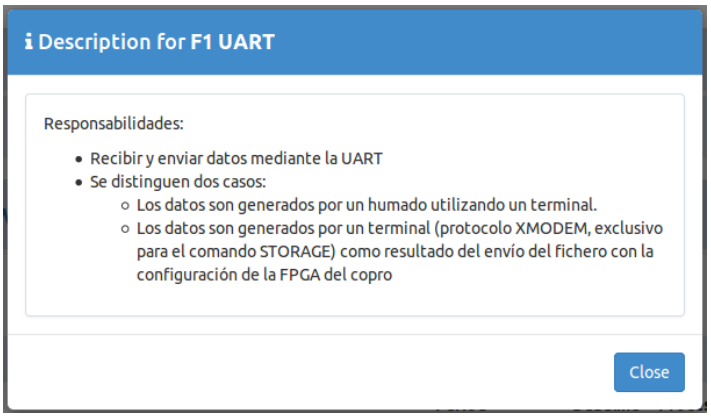


Figure 14: Brief description of the flow

This feature is very convenient to help the engineer with a plain explanation of every task and flow that has been previously documented. It is also used when a project report is exported from the web interface, which can be done in different formats: PDF, XLS and JSON.

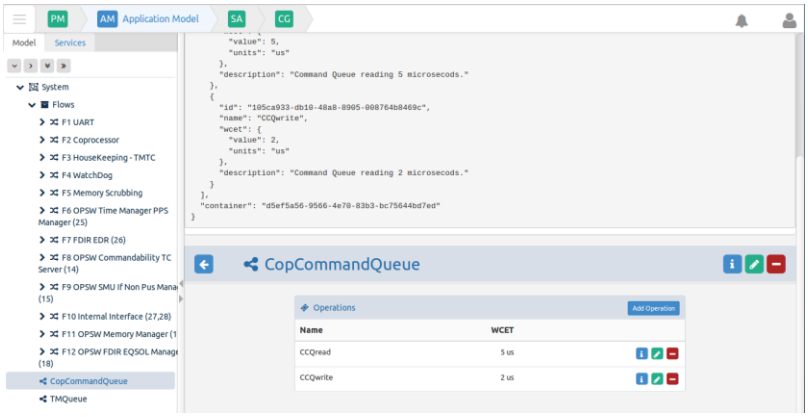


Figure 15: Example data for a Shared Resource element

Finally, an example of a shared resource element can be seen at the figure, where a coprocessor queue has been modeled with two possible operations: read and write. The corresponding JSON structure internally saved for the model is also visible at the UI.

3.3.3 System Analysis description

The engineer is able to specify the kind of system analysis she wants to perform. It is also possible to interact with the tool to fix any issues that makes the system unfeasible. The implementation of these analyses is fast enough to be executed “on the fly”, so the user can easily test several possible conditions and see the result immediately.

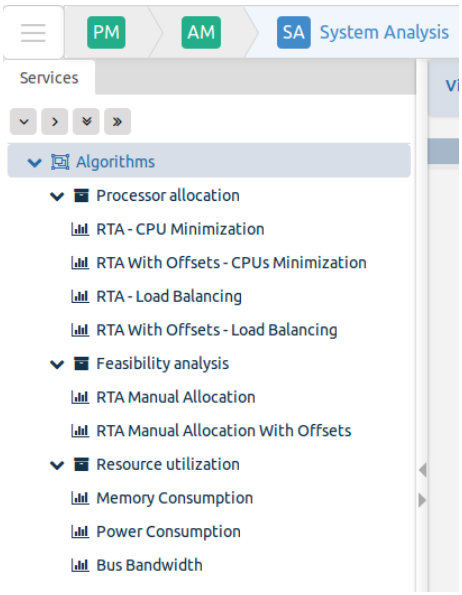


Figure 16: Set of algorithms to check system analysis

Two version of the <Response Time Analysis> algorithm have been run and documented for the "Satellite" use case: <CPU Minimization> and <Load Balancing>. Next two snapshots of the web results obtained after running the analyses can be seen. Although both analyses result in feasible schedules, it is clear how the second one is able to evenly load the tasks between the two processors.

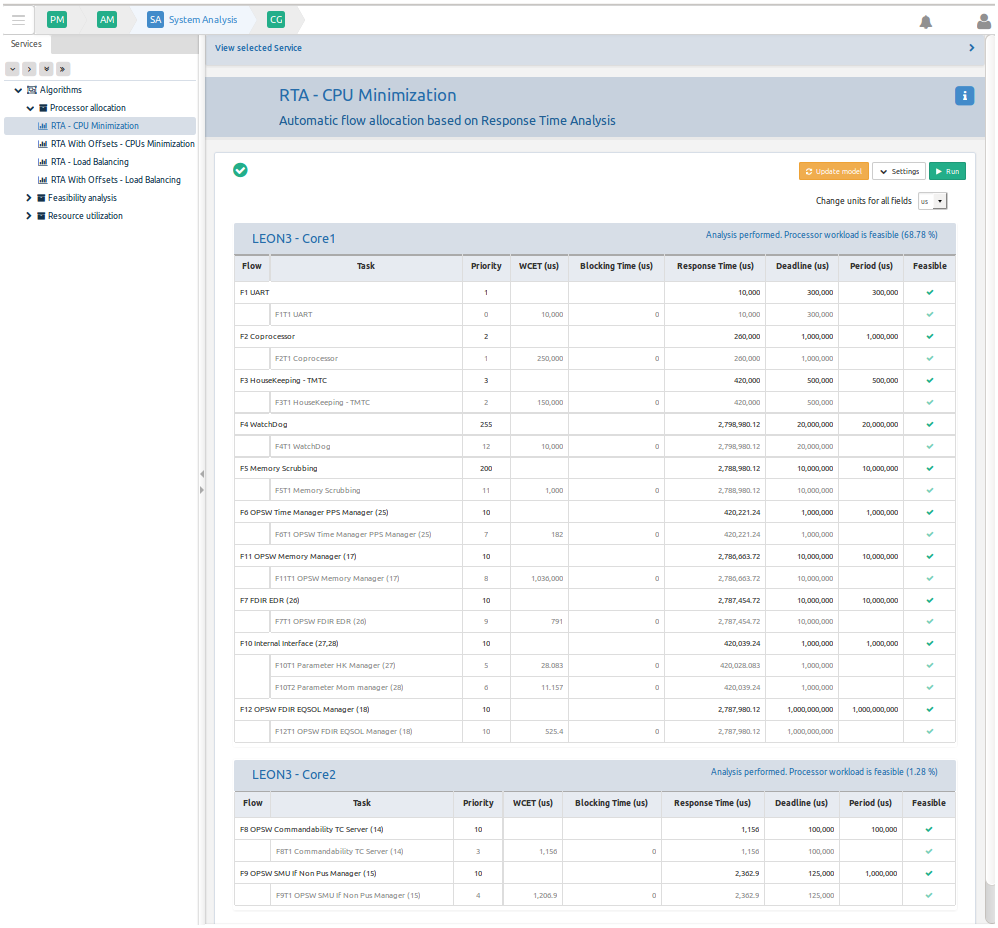


Figure 17: Response Time System analysis with CPU Minimization

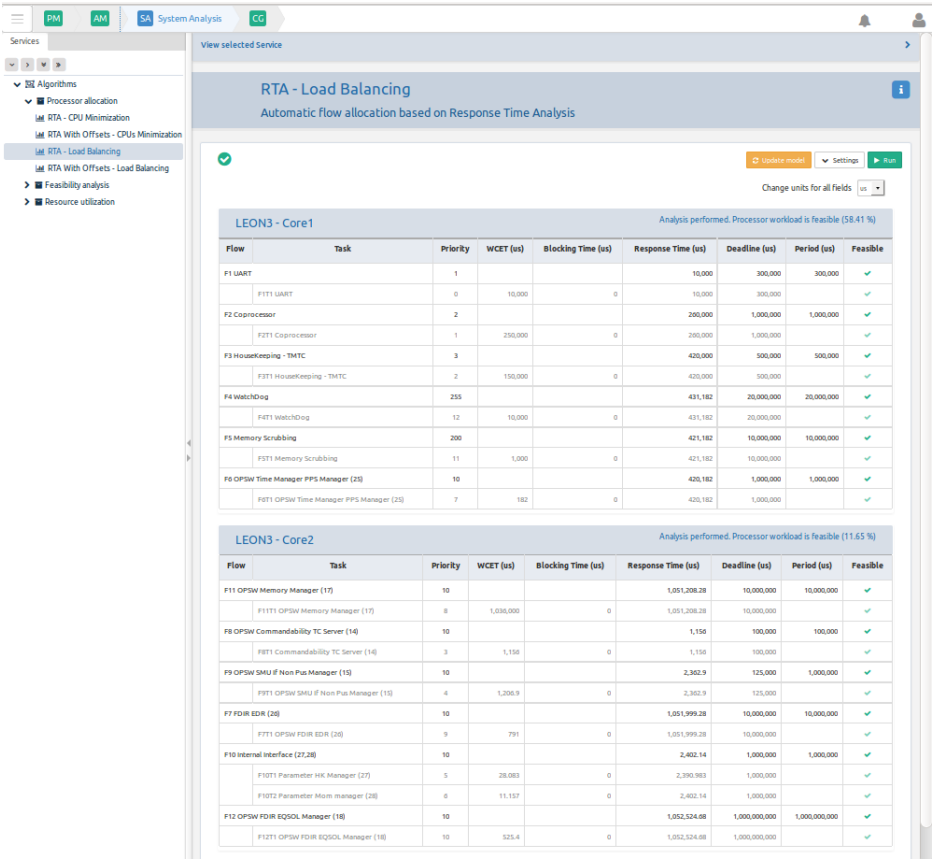


Figure 18: Response Time System analysis with Load Balancing

3.3.4 Code Generation description

A target platform can be addressed to automatically generate the high level code to manage flows and tasks.

In the next example, source code has been automatically generated for a POSIX target platform and a simple task set which is only meant to illustrate the proof of concept for Code Generation currently available from the web user interface of the art2kitekt tool suite:

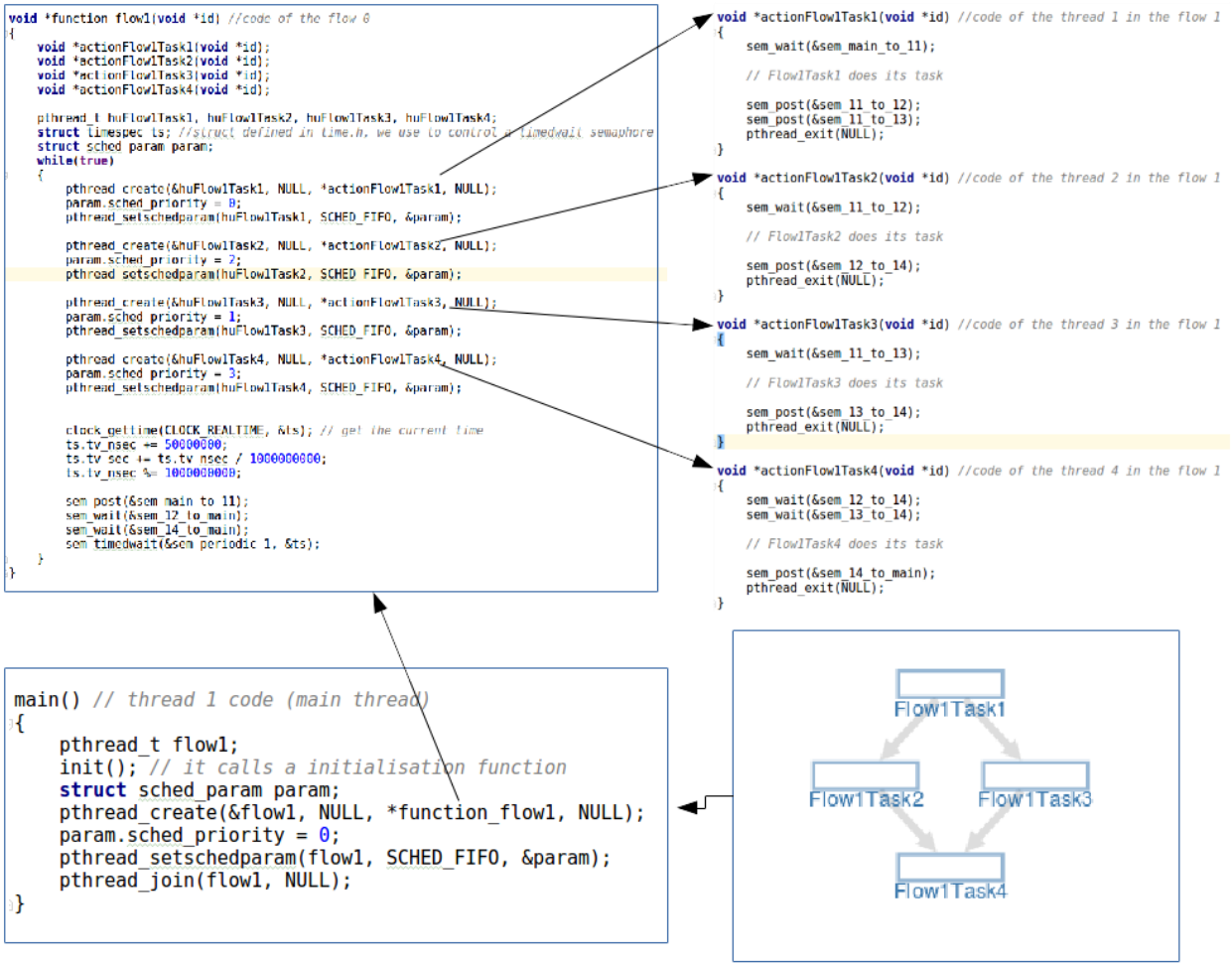


Figure 19: Code Generation for POSIX

However, not only POSIX code will be automatically generated for a given project, but also RTEMS will be available in the near future.

3.4 Distributed platform TTEthernet configuration tooling

In the context of EMC2, the TTEthernet toolchain has been further developed in order to support the network configuration for the network nodes targeted for space platform integration. The TTEthernet toolchain allows for the planning of the TTEthernet network with its networking nodes, the creation of physical and logical channels in the network and mapping the corresponding virtual links to the devices connected in a so-called “network description” (ND). The tool performs checking and validation of the ND and checks if the critical traffic can be scheduled according to its timing constraints (i.e. period, latency, bandwidth allocation). Based on this calculation, the schedule for time-triggered traffic is provided and corresponding Network Configuration files (NC database) are created which provide the individual configuration files for each of the devices in the network. This includes the

- Network configuration file, comprising:
  - Devices and Ports mapping
  - Synchronization parameters
  - Virtual Link and messages parameters

- Device specification files, containing

- Device type
- Traffic routing information
- Schedule parameter for TT traffic
- BAG and Rates for RC traffic

- Device Target mapping
  - Maps device to specific hardware

After this step, the toolchain allows to handle the TTEthernet network configuration XML database with a graphical user interface (GUI) and provides a table-based editor of the network configuration XML database. The Network-Configuration XML database is guided through the GUI in Eclipse environment, where individual VLs going through the network can be displayed and edited, manually change timing. In the configuration tool, new time-triggered and rate-constrained (TT/RC) virtual links going through the network can be created, and timing parameters of the Virtual Links (VLs) can be adjusted.

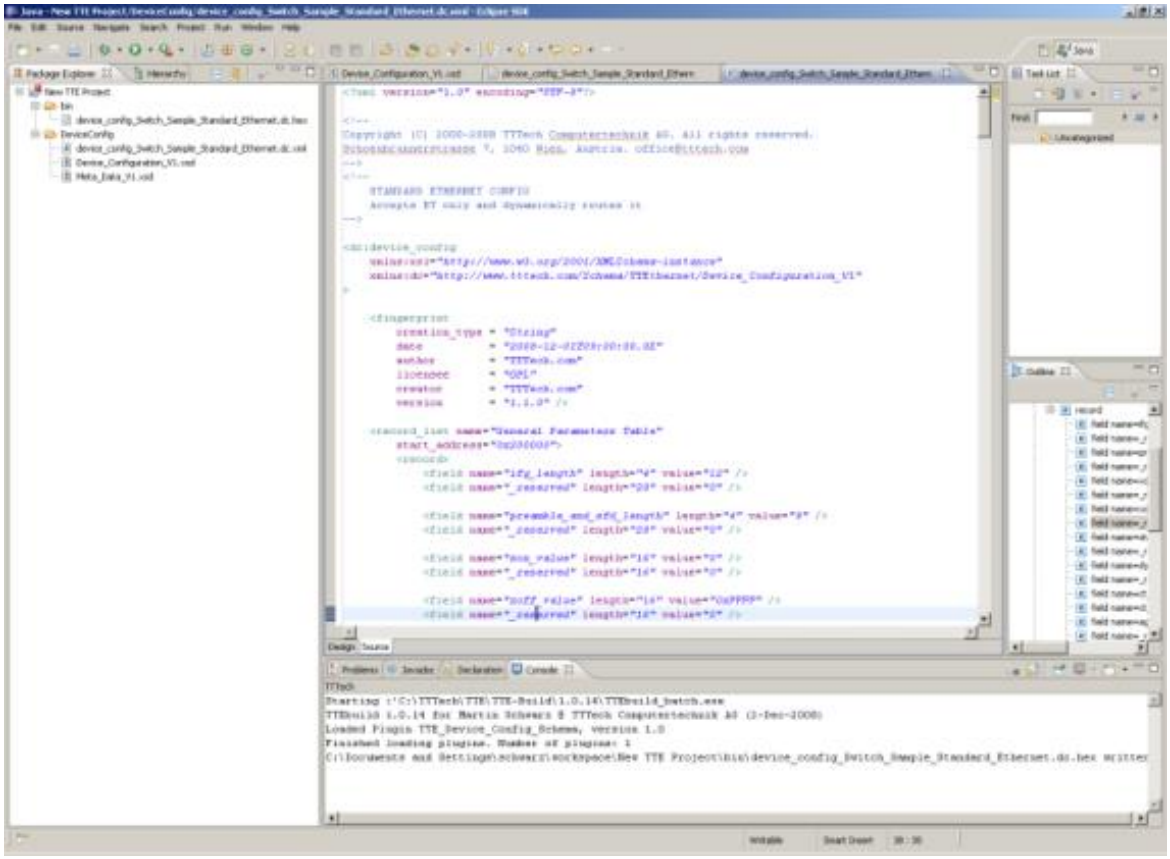


Figure 20: Network configuration editor

After this phase, the configuration files are translated to the respective binaries for each of the devices in the network. The overall workflow is depicted in the figure below:

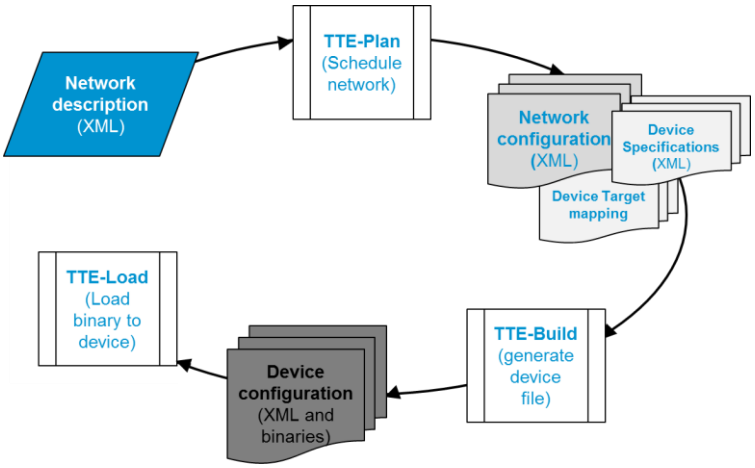


Figure 21: Network configuration toolchain

4. Use Case Optical Payload Applications

4.1 Integrasys Satellite Communications Link Emulator

In the following subsections Integrasys' Satellite Communications Link Emulator it is described.

4.1.1 Overall Structure and Purpose

Integrasys has undertaken the development of a Satellite Communications Link Emulator for multicore platforms. The emulator allows monitoring the quality of the carriers that are transmitted to a satellite transponder, checking the status of the channel and tracking the possible interferences that may arise in a satellite communiactions link which degrade the performance of the service. The adoption of parallelisation techniques applyable to multicore platforms greatly improves the performance of the signal processing application (Vectorsat) running in the emulator.

Below one can see a block diagram of the overall Satellite Communications Link Emulator.

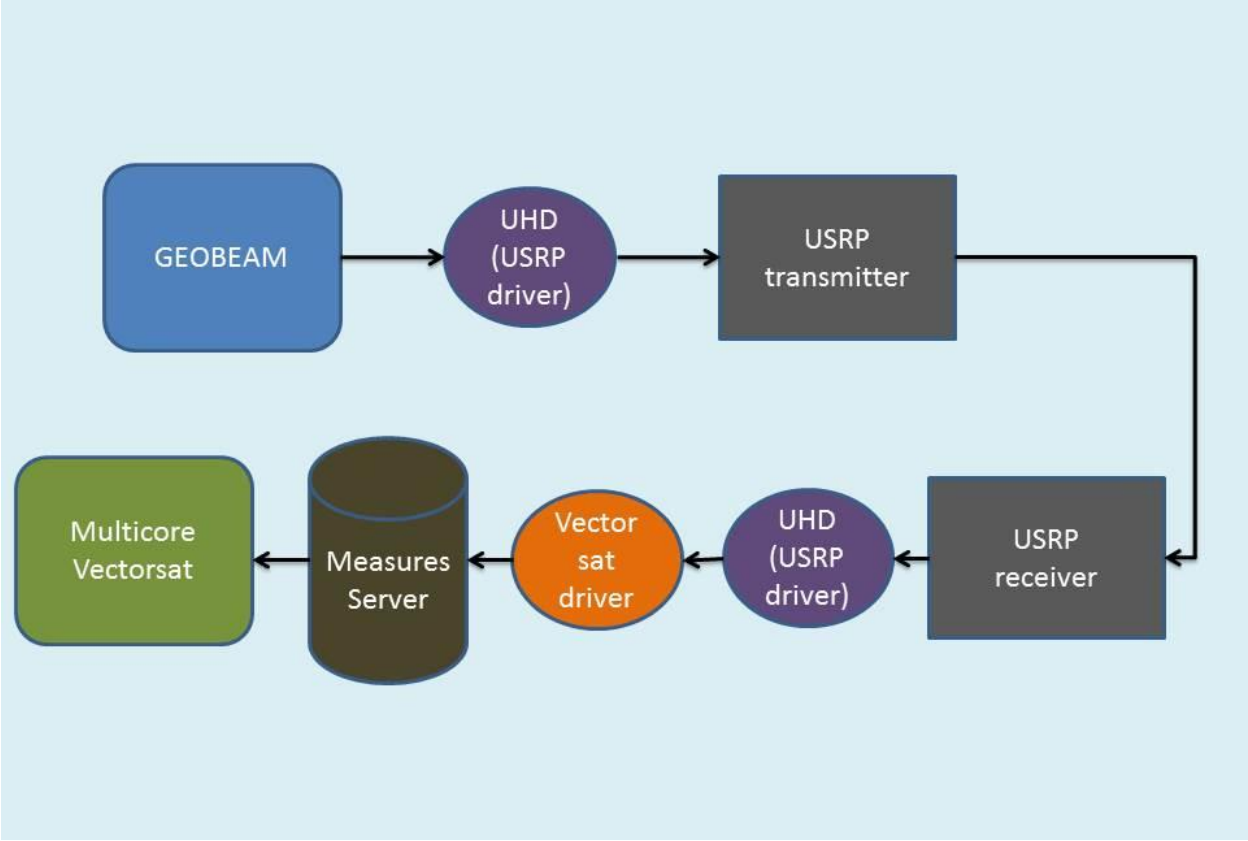


Figure 22: Satellite Communications Link Emulator Block Diagram

The emulator comprises the three main elements of a communications link, namely transmitter, channel and receiver. The transmitter and receiver have been implemented by means of a SDR (Software Define Radio) Platform. We have used the Ettus Research USRP B200 model as transmitter and receiver. A software called GEOBEAM designed for satellite communications planning is used to perform all the calculations for the link and customising the transmitter parameters. Another software called Vectorsat designed for signal monitoring, visualisation and analysis is running in a multicore platform.

4.1.2 Operation and Use

In this subsection, we describe the above depicted blocks that are used and how they interconnect when the emulator is in operation.

4.1.3 GEOBEAM Application

This application allows modelling a satellite communications network, with all the elements that are part of it, to achieve network planning and performance results. The tool includes, among others, VSAT, Hub, Satellite and Carrier Objects that make it possible to easily model a satellite communications link. An attractive and easy to use GUI that features a three-dimensional Virtual Globe from NASA's WorldWind shows the setup of the network in a geo-located manner.

The application can have many uses such as satellite footprint modelling, network planning or physical layer performance analysis but for the Satellite Communications Link Emulator that we are interested in our use case we will only need the module which models a transmitter and a carrier which are the ones that will be used in the emulated communications link as well as the Link Budget Calculations module which will compute the received carrier and interference power that will be received for any emulated satellite link.

In our Emulator, GEOBEAM therefore configures the link parameters and sends those parameters via a Socket connection to the UHD (USRP Hardware Driver), which communicates with the USRPs.

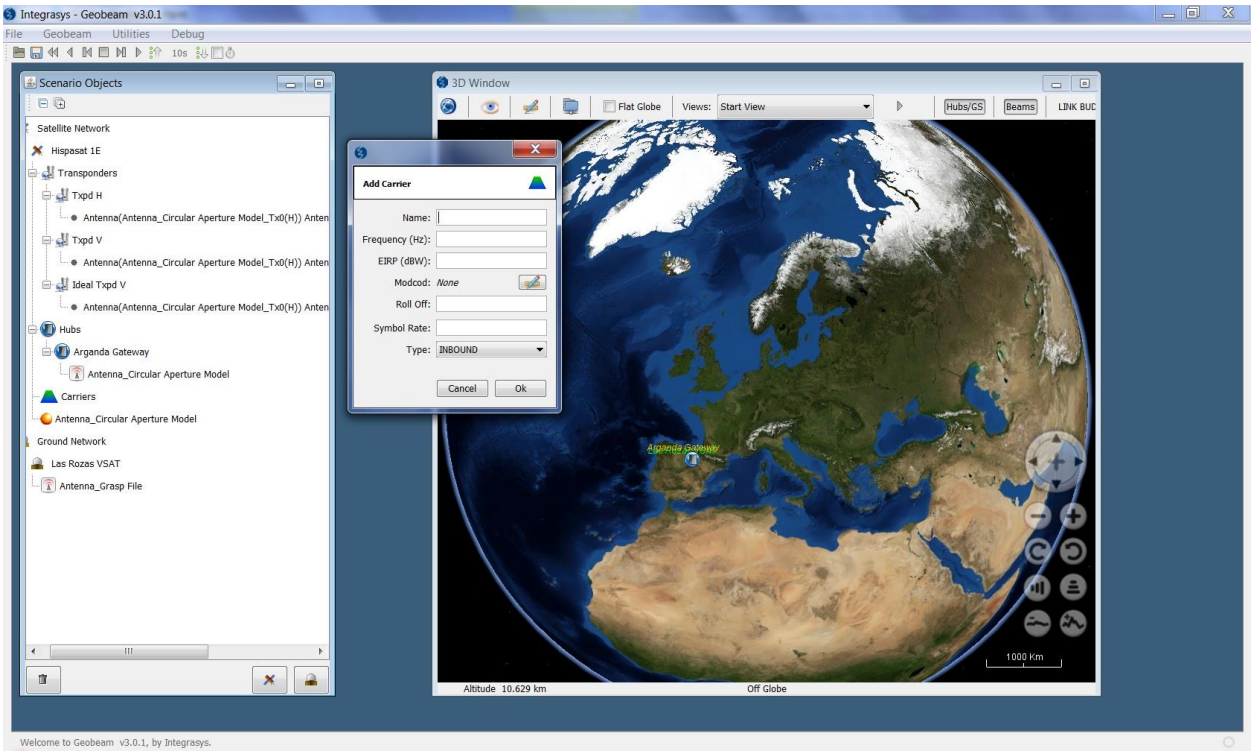


Figure 23: GEOBEAM Graphical User Interface

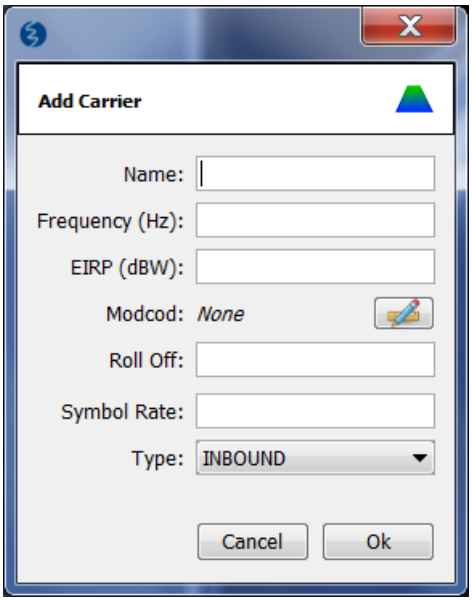


Figure 24: Carrier Setup Menu

4.1.4 USRP B200 Software Defined Radios

The hardware used to trasmit and receive the electromagnetic signals to simulate the carriers is the well-known USRP B200 Software Defined Radio from Ettus Research. These are unexpensive but flexible sofware radios that can be interfaced through its UHD driver. The RF frontend has individually tunable receive and transmit chains. All frontends have individual analog gain controls, where the receive frontends have 73 dB of available gain; and the transmit frontends have 89.5 dB of available gain. The analog frontend has an adjustable bandwidth of 200 kHz to 56 MHz.

In order to implement the transmit and receive modules we have employed the widespread GNU Radio C++ libraries. The C++ modules are interconnected in Python scripts.

Figure 25: USRP B200 Transmitter and Receiver

4.1.5 **VECTORSAT Application**

This application is the end part of the whole Emulator layout. The Vectorsat Server receives and processes the signal measurements from a RF device driver (in our use case, the USRP) and process all of them before sending them for visualization to the Vectorsat Client running on another computer. By implementing parallesisation techniques for some calculations involved in the signal processing chain, it is possible to greatly reduce the computation time and therefore performing more tasks at the same time or in a more efficient manner. This in turn, means added benefits for the reliability of the satellite communications link such as the possiblity of tracking several carriers at the same time with a single spectrum analyser and therefore being able to detect and select the less affected channels by often-present interferent signals.

Therefore, the Vectorsat applications let us demodulate, visualise and analyse the signal that we have received on the second USRP, after being modelled and sent with the first USRP. In tests we performed, the Vectorsat application running on single-core mode was able to able to provide 10 to 12 traces per second on 512 points for the FFT (Fast Fourier Transform) while when increasing the number of points to 1024, performance decreased to 8 to 9 traces per second. On the other hand, multi-core version provides more than 20 traces per second, which makes more advanced and reliable features like several carrier simulataneous monitoring possible. There are several additional features or benefits that can be derived from the use of the Emulator such as transponder characterization, 1-dB compression point computation or uplink power control.

Figure 26: Vectorsat Client demodulating an 8-PSK carrier with interferences

5. Use Case Platform Applications

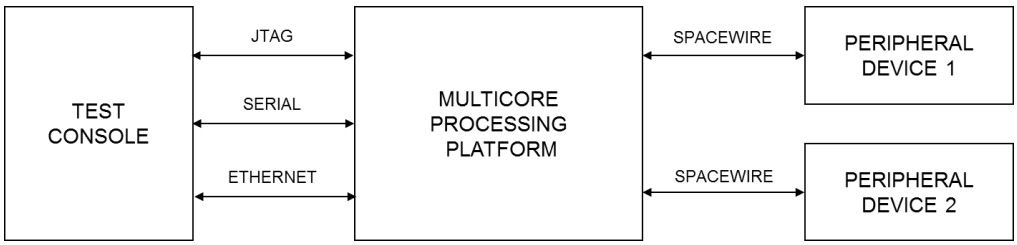
This document describes the demonstrator developed by TASI and UNIVAQ in the scope of EMC2 project, a satellite reference platform based on a multicore processor. The goals of the demonstrator are:

a) To evaluate and benchmark the ESA Next Generation Microprocessor (NGMP)
b) To analyze the benefits and drawbacks related to the use of multicore platforms in mixed-criticality (MC) aerospace applications
c) To evaluate the use of hypervisors as a feasible solution to implement MC application over multicore platforms
d) To define proper design strategies for Mixed Criticality applications over a virtualized multicore platform.

This document contains an updated version of the software application design. It is also describes the main implementation details, with specific reference to the considered virtualization solutions.

5.1 Demonstrator Overview

The demonstrator setup models a simplified satellite platform.



Proposed demonstrator is intended to model:
- The satellite's telecommand and telemetry management.
- The on-board management of large file transfers.

Specifically, the computing platform is able to receive telecommands and send back telemetries, as per ECSS-E-70-41A Standard. The unit is able to communicate with the peripheral devices using Spacewire Links and implementing the Remote Memory Access Protocol (RMAP, as per ECSS-E-ST-50-52C).

TASI and UNIVAQ jointly worked in the hardware selection and software design. Two different hypervisors have been considered of interest for the study:
- SYSGO PikeOS;
- FentISS XtratuM;

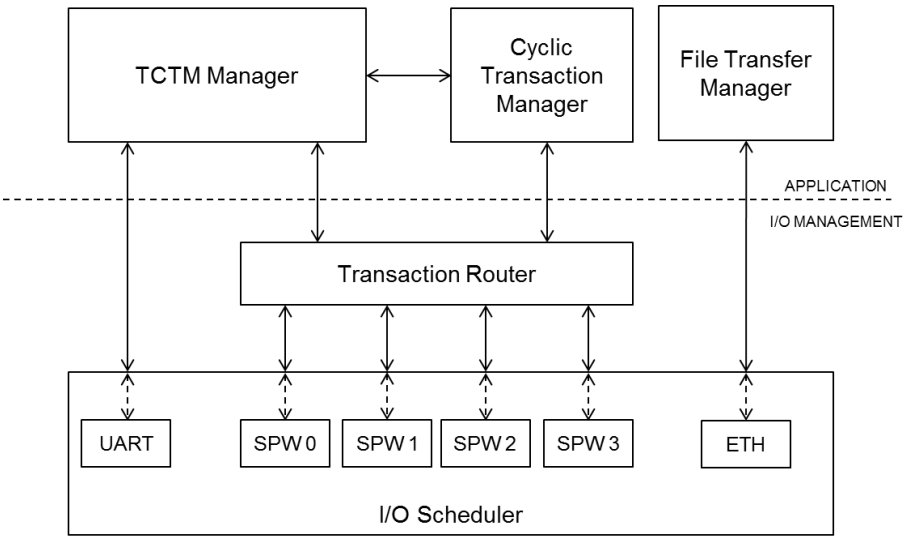
5.1.1 **Hardware platform**

The platform includes the following components:
- A LEON4 multicore processing platform: a commercial board (Cobham - Gaisler GR-CPCI-XC4V/LX200) including a quad-core Leon4 processor. The board models the satellite Platform Control Computer and execute the reference application software.
- A Test Conduction Console (TCC): a unit managing the Leon4 processing platform. The test console communicates with the Leon4 board via a serial link, sending commands and receiving data. Moreover, it receives data also over an Ethernet link and uploads the software by means of the JTAG access.

- Peripheral devices: generic devices generating data and emulating the Remote Terminal components of a satellite. The boards communicates with the LEON4 board by means of three SpaceWire links.

5.1.2 **Reference software application**

A preliminary version of the software architecture of the reference application has been partially described in Deliverable D9.4. The following figure details the current status of the software architecture.



The entities included by the application layer have not been modified since the last deliverable. Specifically, this layer includes:

- A *TCTM Manager*, in charge of managing all the telecommands received from the test console and of generating and sending the platform telemetries. This component also manage the activity of the Cyclic Transaction Manger.
- A *Cyclic Transaction Manager*, in charge of periodically acquire values and generate telemetries from peripheral devices.
- A *File Transfer Manager*, in charge of managing the forwarding of large files coming from the Ethernet link. These process may represent the management of a payload unit.

The structure of the I/O management has been further refined since with respect to previous specification, including the following entities:

- A *Transaction Router*, handling the RMAP requests coming from the different on-board software entities, and forwarding them to the Spacewire interfaces (by means of the I/O Scheduler).
- An *I/O Scheduler*, forwarding the data provided by the device interfaces and taking care of timing requirements.

The I/O Scheduler relies on device drivers in order to manage the access to the physical links used by the application. As reported in the previous deliverables, the following drivers have been included:

- A Serial (UART) device driver, allowing the usage of the serial console to send and receive characters. This interface can be used to receive telemetry data and to send telecommands.
- A SpaceWire device driver, allowing the usage of the SpaceWire links in order to send and receive SpaceWire packets between the LEON4 and the attached peripherals.
- An Ethernet device driver, allowing the transmission of large files..

5.2 General implementation concepts

This section describes the implementation of the platform reference application. The reference software supports the following features:

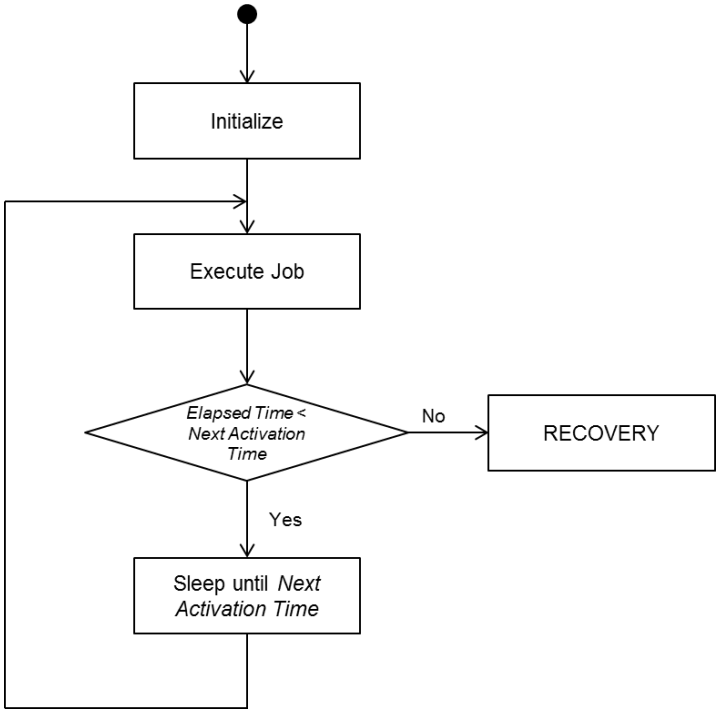
- Reception of telecommands from a Test Console connected to the serial interface
- Acquisition of telemetries from remote terminals connected to the SpaceWire interfaces. This acquisition can be:
  - Sporadic, when started by a dedicated telecommand.
  - Cyclic, if triggered by the application on a periodic basis.
- Dispatching of acquired telemetries to a test console connected to the serial interface.
- Enabling/disabling of periodic telemetry acquisition.
- Enabling/disabling of periodic telemetry download.
- Retrieval and download of a Large File from a device connected to the Ethernet interface.

The software application supports three different transaction typologies:

- High priority transactions, associated with the most critical TMTC transactions and characterized by hard real time requirements;
- Medium priority transactions, associated with TCTM transactions of intermediate criticality and characterized by soft real time requirements;
- Low priority transactions, associated with large file transfers operations and modeled as background transactions with no specific real time requirements.

5.2.1 Description of software components

Described software components are implemented as periodic software tasks. The following figure shows the generic task execution model.



Each task is characterized by:

- A job, consisting in the activities to be performed.
- An initial activation time, statically defined by design. The task will start to execute its job for the first time at this specific instant.

- A period, i.e. the interval between two successive task activation.

A task is created and started using the specific hypervisor's functionalities. It is assumed that each task is able to complete the initialization phase before its initial activation time.

Each task will have an execution deadline, as it shall complete the job before its next activation time. If a task misses its deadline, a software-based recovery action is triggered.

The following subsections describe in details the function allocated to each software component and the job actually performed.

5.2.1.1 TCTM

The TCTM is a component responsible for telecommand and telemetry management. Specifically, it is in charge of:

- Receiving from the I/O Scheduler the telecommands sent to the platform by means of the test console.
- Enabling and disabling the cyclic acquisition of telemetries upon reception of related telecommands, sending proper internal directives to the Cyclic Transaction Manager.
- Periodically downloading the telemetries acquired by the Cyclic Transaction Manager. Specifically, it will read the telemetries from a shared memory area and will send them to the I/O Scheduler.
- Starting and stopping the download of periodic telemetries upon reception of related telecommands.
- Managing the sporadic acquisition of telemetries, upon reception of related telecommands. Specifically, it will send the related RMAP requests to the transaction Router and will generate the proper telemetries based on received RMAP replies. Generated telemetries will be sent to the I/O Scheduler.

5.2.1.2 Cyclic Transaction Manager

The Cyclic Transaction Manager is a component responsible for the periodic acquisition on on-board telemetries. Specifically, it is in charge of:

- Starting and stopping the periodic acquisition of a certain telemetry upon reception of related directive from the TCTM.
- Periodically sending the proper RMAP requests to the Transaction Router.
- Managing the RMAP replies received from the Transaction Router, generating the related telemetries and storing them in a dedicated buffer (Telemetry Buffer).

5.2.1.3 File Transfer Manager

The File Transfer Manager is a component responsible for the download of a large file towards the test console. Specifically, it will:

- Read the data from the Ethernet Interface.
- Send the data to the Test Console.

The component will interact with the I/O Scheduler to execute its transactions.

5.2.1.4 Transaction Router

The Transaction Router is a component responsible for RMAP data flow management. Specifically. It is in charge of:

- Receiving the RMAP requests from the various on board sources (e.g. the periodic requests from the Cyclic Transaction Manager, the sporadic requests from the TCTM) and forwarding them to the I/O Scheduler.
- Receiving the RMAP replies from the I/O Scheduler and sort them to the request original sender.

As said, the Transaction Router is able to keep track of the various transactions, by assigning a unique identifier to each one and using this identifier to sort back the reply to the correct destination.

5.2.1.5 I/O Scheduler

The I/O Scheduler is a component responsible for managing the processor communication interfaces. It manages both the priority and the timing of the various requests. Specifically, the I/O Scheduler:

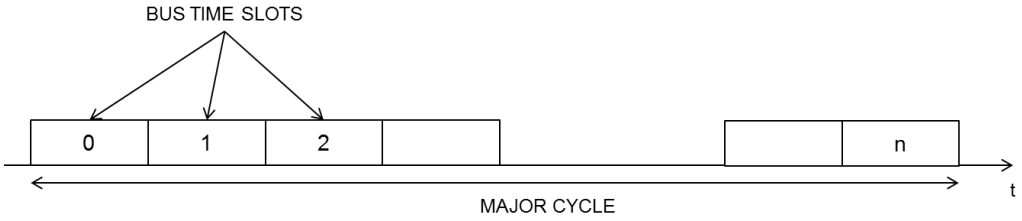
- Manages the serial communication interface, by means of a proper device driver. It supports the reception of telecommands sent by the test console and the download of telemetries generated by the application.
- Manages the SpaceWire router interfaces, by means of a proper device driver. It supports the sending of RMAP requests to peripheral devices and the reception of RMAP responses.

The I/O Scheduler will:

- Read the telecommands from the serial interface and forward them to the TCTM
- Receive the RMAP requests from the Transaction Router and forward them to the SpaceWire interfaces.
- Read the RMAP replies from the SpaceWire interface and forward them to the Transaction Router.
- Receive the telemetries from the TCTM and forward them to the serial interface.

The I/O Scheduler is implemented as a table-driven cyclic executive, and is able to manage the access to the various interfaces in a strictly deterministic way. The scheduler defines:

- A series of elementary frames (bus time slots), each one dedicated to the management of one or more transaction.
- A major cycle, constituted by the succession of the various bus time slots and cyclically repeated.



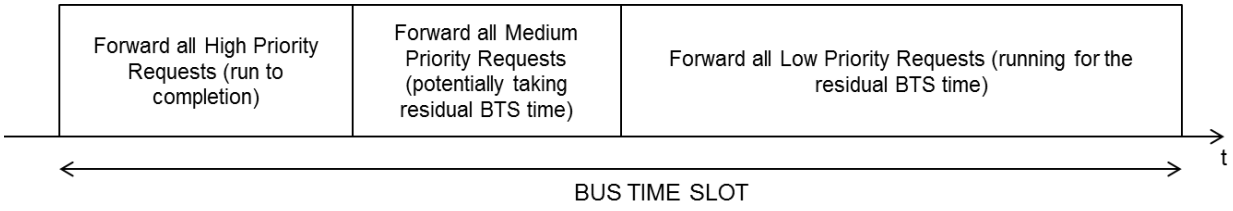
As said, the scheduler is table driven: the operations to be executed in each Bus Time Slot are statically defined at design time.

In general, the scheduler is able to manage both the priority and the timing of the transactions. For each transaction, it is possible to specify:

1. The Bus Time Slot in which the transaction has to be executed.
2. The priority of the transaction.

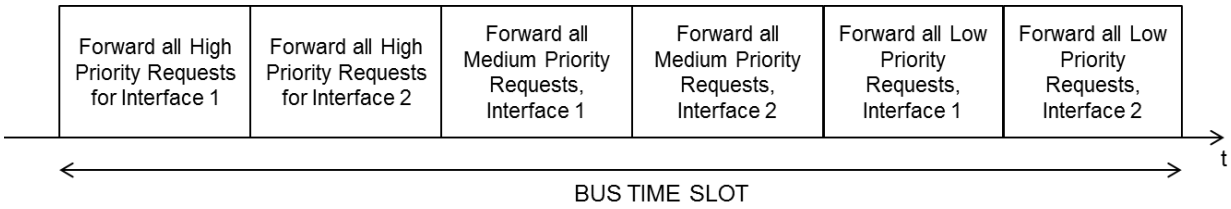
As said, the concept of priority is specifically associated to request to the peripheral devices. Replies from the devices are carried on by the I/O scheduler at predefined times (BTS). There is no explicit concept of priority for replies, but they are implicitly managed as high priority transactions.

Inside the single Bus Time Slot, the transaction management will implement the logic detailed in the following picture.



If high priority transactions over a specific interface are enabled in a specified time slot, the I/O Scheduler will manage all the queued requests. This activity will continue until all pending transactions will have been forwarded. If medium priority transactions are enabled, the I/O Scheduler will estimate the maximum number of executable transactions on the basis of the remaining BTS time. It will manage the pending medium priority transaction, up to the estimated maximum number. If the remaining BTS time is not sufficient to manage all the pending transactions, remaining transactions will be remapped in the next available time slot. If all high priority and medium priority transaction have been forwarded and there is still some elapsed BTS time, the I/O Scheduler will manage the pending background transactions allocated to the slot.

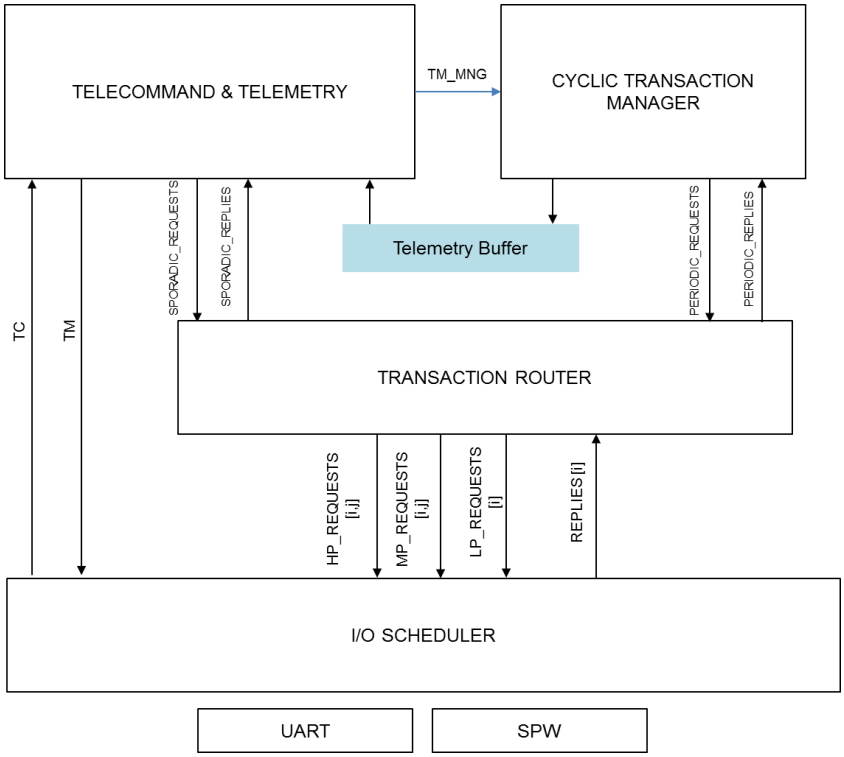
This concepts can be generalized in the case of multiple interfaces to be manages in a single Bus Time Slot.



All the high priority request shall be managed in the time slot for which they are scheduled. Failing in correctly scheduling an high priority transaction will result in a critical error and a recovery procedure for the system. Medium priority transactions can simply be rescheduled, avoiding deadline missing and recovery procedures.

5.2.2 **Telecommands &Telemetries Interfaces and Data Flow**

The following figure specifies the interfaces and communication mechanisms involved in the telecommand and telemetry management.



The different software entities are able to communicate by means of:

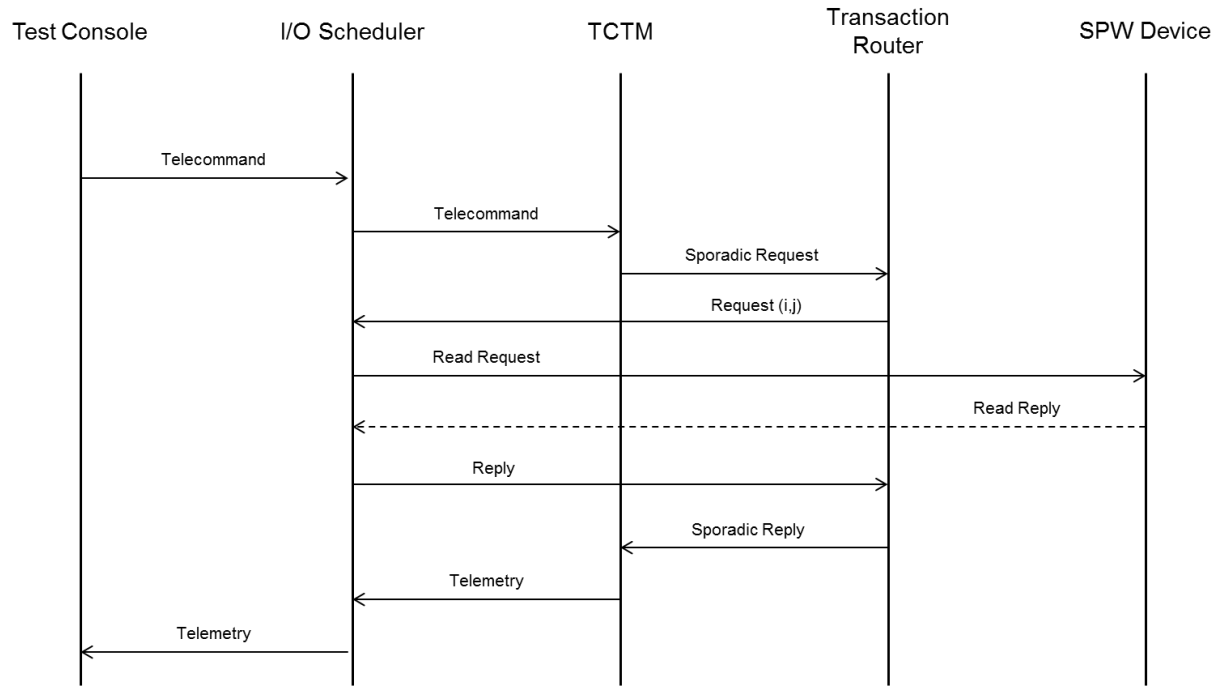
- Message passing channels, implemented as FIFO queues.

- Shared memories areas, statically defined at design time.

Specifically:

- The TCTM Manager is able to receive telecommands and send back telemetries communicating with the I/O Scheduler by means of two dedicated communication channels.
- Each component managing RMAP request will have one sending channel and one receiving channel connected to the Transaction Router.
- The I/O Scheduler allows to manage the different priority levels and timing by exposing multiple communication interfaces to the Transaction Router. Specifically, the *(i,j)* channel will be associated to the transactions over *i-th* interface, scheduled to the *j-th* time slot.
- The Cyclic Transaction Manager will store the periodically acquired data inside a shared memory area (Telemetry Buffer), that will be read by the TCTM Manager.

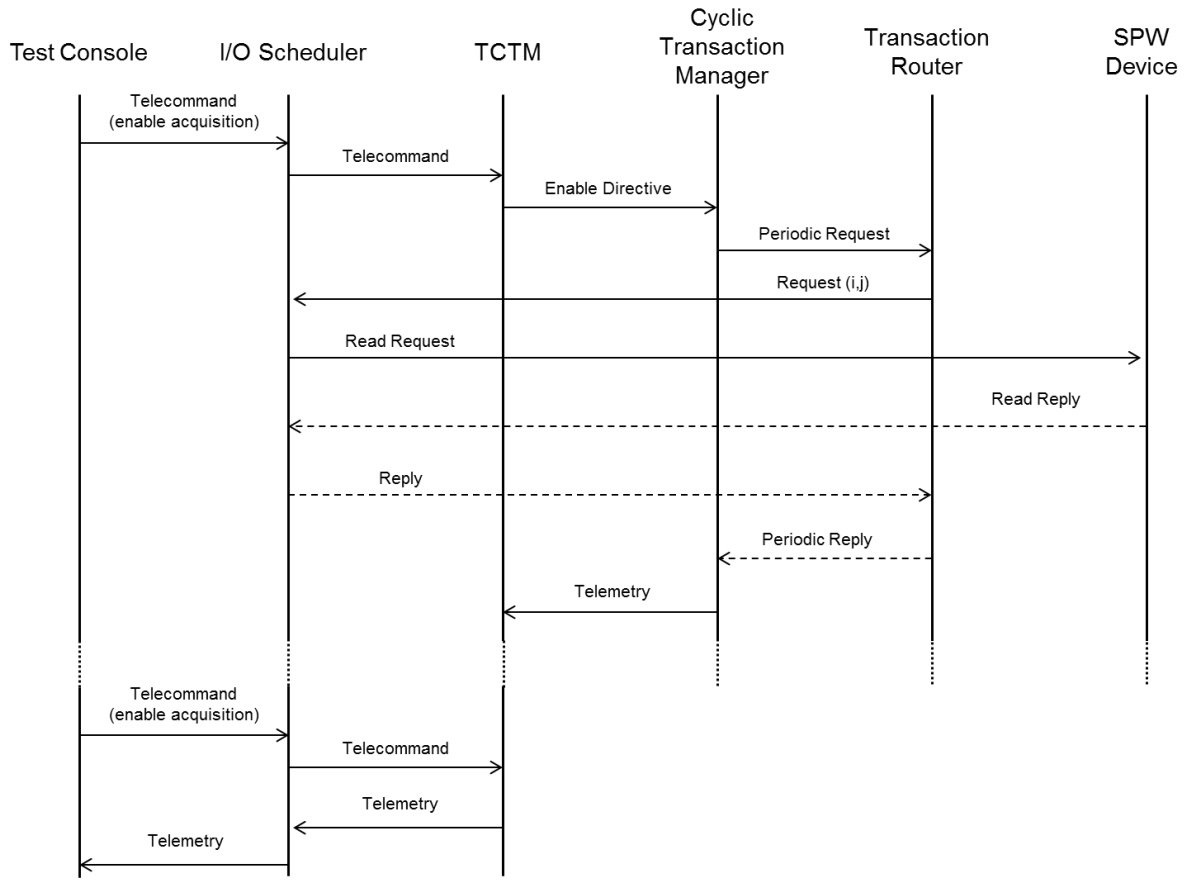
The following diagram schematizes the acquisition and download of sporadic telemetries.



The process is initiated via the test console, and involves the following steps:

1. The telecommand is received by the I/O scheduler, which periodically polls the serial interface.
2. The I/O scheduler will forward the command to the TCTM process.
3. The TCTM Manager will verify the received command and issue a RMAP request to the transaction router.
4. The transaction router will receive the RMAP request, generate an identifier for the transaction and forward it to the I/O Scheduler.
5. The I/O Scheduler will send the request to the remote terminal (SpaceWire peripheral) receiving back an RMAP reply.
6. The RMAP reply will be forwarded to the Transaction Router by the I/O Scheduler.
7. The Transaction Router will dispatch the reply to the TCTM (as the reply transaction identifier is associated to a request issued by the TCTM).
8. The TCTM will create a telemetry packet and send it to the test console by mean of the I/O Scheduler.

The following diagram schematizes the process of enabling, acquisition and downloading of the periodic telemetries.



In order to enable the periodic acquisition of telemetries, the user shall send an enable command over the test console. Then, the following steps will be executed by the software:

1. The command will be received by the I/O Scheduler, and forwarded to the TCTM Manager.
2. The TCTM Manager will analyze the command and send a directive to the Cyclic Transaction Manager. The directive will enable the periodic acquisition of a specific telemetry.
3. The Cyclic Transaction Manager will receive the directive and start to periodically acquire the specified telemetry.

The process of periodic data acquisition is autonomously managed by the Cyclic Transaction manager, by means of the following steps.

1. Sending of the RMAP request to the Transaction Router
2. Dispatching of the RMAP request and reception of the RMAP reply (as in steps 4-7 of the sporadic requests process)
3. Generation of the telemetry on the basis of received RMAP reply. The telemetry will be stored in the dedicated Telemetry Buffer.

Generated telemetries can be downloaded to the test console, by enabling the cyclic download. The process will involve:

1. The I/O scheduler will receive the command and forward it to the TCTM.
2. The TCTM will periodically read the Telemetry Buffer and send the telemetries to the I/O Scheduler.
3. The I/O Scheduler will forward the telemetries to the Test Console.

5.3 Hypervisor-specific implementation concepts

The reference application can be modeled as a mixed-criticality application, since it includes different functions with different criticality levels. Specifically, the same platform should support the following different types of functions:

- Management of telecommands & telemetries: a set of critical data flows, with specific real time requirements. It is possible to distinguish:
  - High priority transactions, associated to the acquisitions and actuations on attached peripherals .
  - Medium priority transactions, associated to sporadic telemetry acquisition.
- Management of large file transfers: a set of less critical data flow, with no real time requirements.

The hypervisor allows to enforce the spatial and temporal isolation between the two functions, by allocating them to different software partitions. The mapping of different partitions to the various CPU is currently under finalization.

5.3.1 PikeOS implementation details

The following design choices have been made for the development of the PikeOS based version of the application:

- Use of custom inter-partition communication mechanisms. This has been motivated mainly by the application performance requirements.
- Implementation of kernel-level device drivers, in order to improve the system responsiveness.

5.3.2 XtratuM implementation details

The following design choices have been made for the XtratuM-based development:

- Development of a custom partition-level tasking abstraction.
- Use of native inter-partition communication mechanisms.
- Use of partition level device drivers.

6. References

[1] D4.9 – Description of first demonstrator and start of final evaluation phase; v0.5, 21/09/2016.
[2] D4.7 - Description of detailed design of architecture and API for error handling and redundancy of accelerators; v1.3, 30/09/2015.
[3] Platform Flash XL Configuration and Storage Device, Xilinx (UG438 (v3.0) August 5, 2015)
[4] Virtex-5 FPGA Data Sheet: DC and Switching Characteristics, Xilinx (DS202 (v5.5) June 17, 2016)

7. Abbreviations

Table 7: Abbreviations

Abbreviation	Meaning
EMC²	Embedded multi-core systems for mixed criticality applications in dynamic and changeable real-time environments
µC	Micro-Controller
WDT	Watchdog Timer