



Using Virtual Prototypes to Improve the Traceability of Critical Embedded Systems

Jean-Michel Fernandez, Magillem
Larry Lapides, Imperas

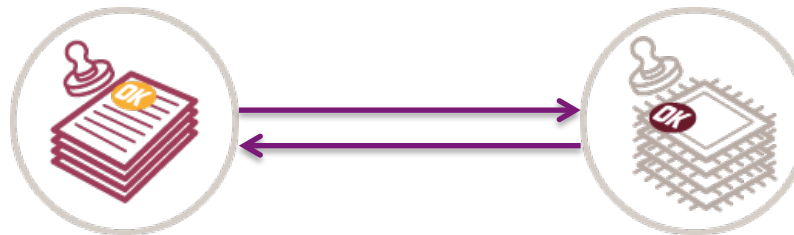


Agenda

- ➔ Scope
- ➔ Traceability
- ➔ Virtual Prototyping
- ➔ Linking the two worlds to improve the debug of Hardware-dependent-Software
- ➔ Illustration on a simple use case
- ➔ Future work and conclusion

Scope

- ➔ Designing critical embedded systems requires compliance with domain specific safety standards, such as DO-178B/C for avionics or ISO-26262 for automotive, which in turn require strong traceability from the functional specification down to the implementation of the complete system including Hardware and Software



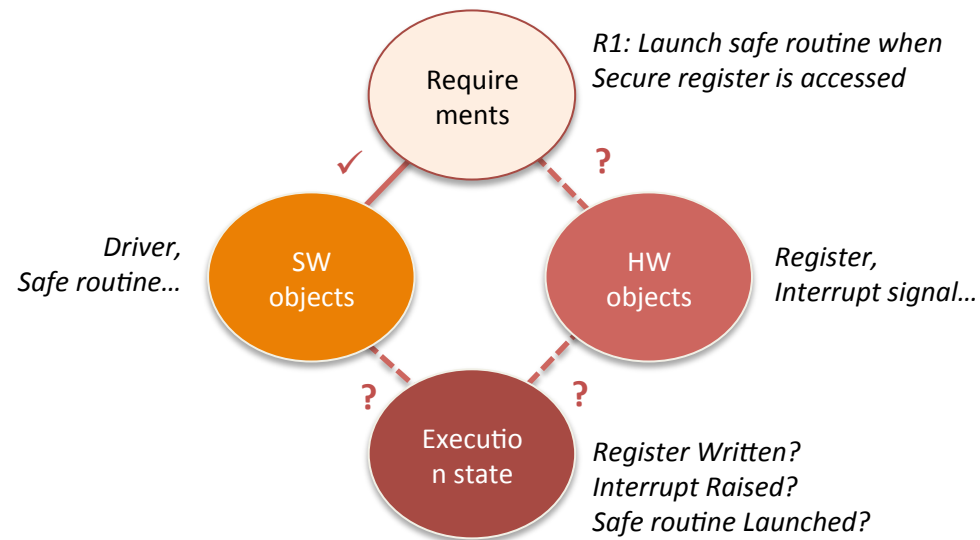
Traceability

- ➔ Traceability is one of the essential activities of requirements management:
 - Ensures that the right product is being built at each phase of the embedded systems development life cycle,
 - Measures the progress of that development
 - Reduces a the effort required to determine the impacts of requested changes.

Traceability tools miss some links

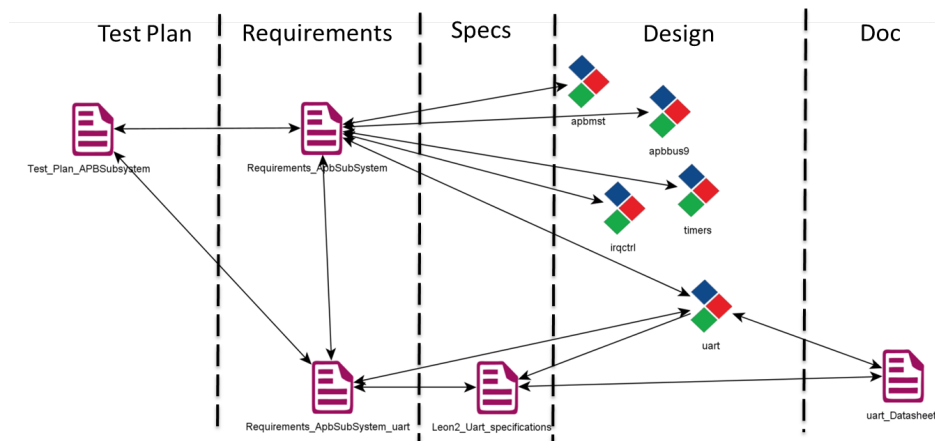
- ➔ Efficient tools exist to trace requirements by creating and managing specification-to-implementation links.
- ➔ But to unambiguously ensure the completeness of a requirement, these tools usually miss the links between the requirement specification, the hardware objects, the software objects and their execution state

Traceability tools miss some links



ISDD

- ➔ This clearly illustrates the need for Integrating Specification, Design and Documentation (ISDD), a novel approach invented by Magillem



ISDD

➔ Such an integration is enabled by the use of XML Metadata to represent any document fragments and manage any link between objects of any kind.

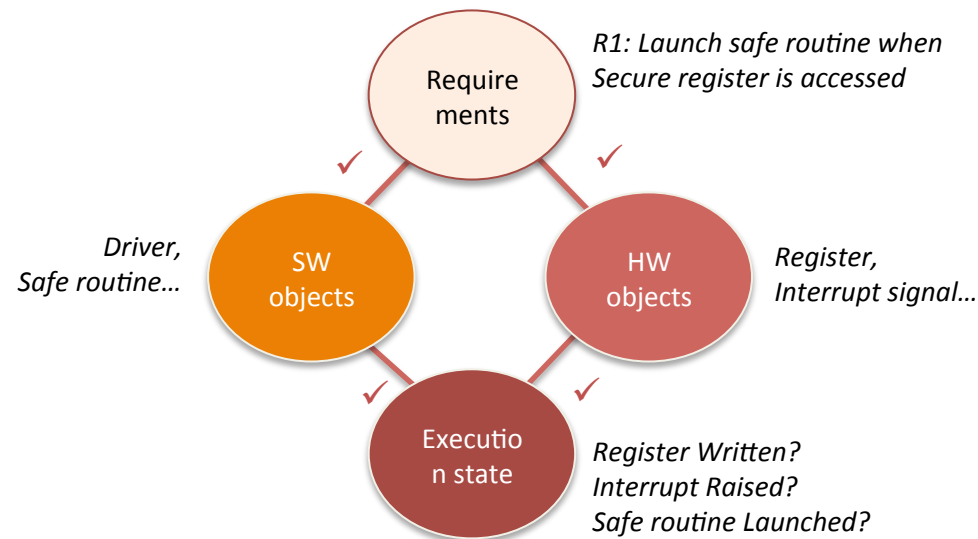
- IP-XACT (IEEE 1685) is an XML standard with a strong semantic to represent Hardware components potentially composed of multiple heterogeneous data
- META-X© is an XML innovative format that encompasses all the standards and de-facto standard document formats based on XML such as DITA or Microsoft Office formats.



Linking to HW details

- ➔ Adding a link to the HW design execution on the physical prototyping board may come too late in the design cycle to capture functional specification or requirements errors.
- ➔ Difficult to verify some functional requirements (e.g. inject/observe an error)
 - Some objects may not be observable on the physical prototype (e.g. register value, interrupt signal)
 - Some corner case states may not be reachable

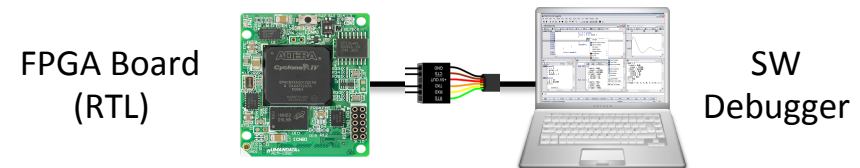
Advanced Traceability tools



- ➔ Moreover, in order to record the execution state of an object, this object must be observable.

Prototyping

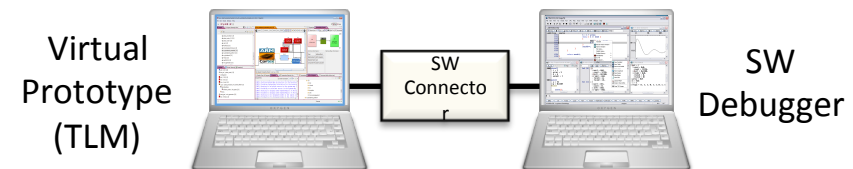
- ➔ Prototyping is a common process to help stakeholders discover problems by validating and verifying their requirements: it is more accessible than the system specification, it demonstrates the requirements, it is reusable and evolutive.



- ➔ It can take various forms ranging from a paper prototype of a computerized system to a formal executable model of the specifications.

Virtual Prototyping

- ➔ VPs are fully functional software simulation models of complete hardware systems that can execute unmodified production binary code at near real time speed.



- ➔ VPs enable early functional verification & debug of embedded software on the target hardware platform, usually months before the physical prototype is available.

Virtual Prototyping

- ➔ VPs can be seen as an executable specification.
- ➔ They are based on a fast simulator and on fast simulatable models of the hardware system, usually using the SystemC standard (IEEE 1666).
- ➔ VPs offer controllability, observability and flexibility.
- ➔ VP tools provide the necessary debug, monitoring and analysis features; usually implemented in a non-intrusive manner, without modifying or instrumenting the production embedded SW code.



Creating the VP

- ➔ Imperas and OVP provide all the SystemC building blocks to quickly build a Virtual Prototype of an embedded system.

 imperas

 OVP
Open Virtual Platforms

 magillem

- ➔ These blocks can be automatically packaged into IP- XACT metadata with Magillem tool
- ➔ And the hardware system can be seamlessly assembled, compiled and simulated together with the embedded Software in a unified Eclipse framework.

Creating the Links

- ➔ Magillem provides an intuitive framework for creating all the links between any fragments of documentation (requirements, specification, code, datasheet...).
- ➔ The fragment can be as detailed as needed, ranging from a complete document (e.g. the specification of an hardware controller device) to the finest unit object (e.g. the voltage or throughput parameter).
- ➔ Once the association is done the tool is able to analyze the impact of any change in any of the linked objects.

Creating the Links

ID	Name	Description
10	Uart need a 32-bit data storage reg...	desc 10
10.1	Uart data storage register need to ...	desc 10.1
12	Uart need to support APB2 protoc...	desc 12

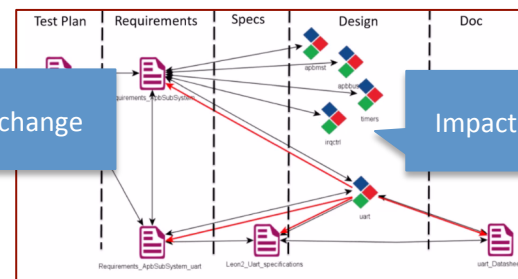
Requirement change

Component: [spiritconsortium.org, Leon2RTL, uart, 1.2]

Search:

Name*	Absolute Address	Access Type	Add...
data_storage	0x0 (computed)	read-write	0x0 (out...
status	0x4 (computed)	read-only	0x4
control	0x8 (computed)	read-write	0x8
scalarReload	0xc (computed)	read-write	0xc

Design change



Impact analysis

uart_Datasheet

HOME INSERT DESIGN PAGE LAYOUT REFERENCES MAILINGS REVIEW VIEW

6.1.1 data_storage

Address: Base Address = 0x0000, Reset Value = UNDEFINED

Name	Bit	Type	Description	Reset Value
data	[31:0]			UNDEFINED
data	[31:0]			UNDEFINED

The data in this register will be send through the following stream depending of the parity setting

parity: [Start] [D0] [D1] [D2] [D3] [D4] [D5] [D6] [D7] [Stop]

parity: [Start] [D0] [D1] [D2] [D3] [D4] [D5] [D6] [D7] [Parity/Stop]

Documentation consolidation

Impact View

Source	Target	Details	Validate
requirements_ApSubSystem_uart	uart	View	✓ ✗
requirements_ApSubSystem_uart	Test_Plan_ApSubSystem	View	✓ ✗

Impact consolidation

Debugging the embedded SW

- One of the most typical usages of a Virtual Prototype is to help writing, debugging and analyzing the embedded SW. Usually the bug comes at the boundary between HW and SW, and it is only visible at run time.
 - VP is a perfect tool for finding such bugs because it provides a very good observability of the HW registers and signals.
 - Such debugging capabilities are usually provided by traditional VP tools and help capturing many SW errors.
- But sometimes, the bug is not there...

Locating the errors

- ➔ The error may come
 - From a change (in the model or in the requirement) that was not properly propagated throughout the traceability chain.
 - Or from a misinterpretation of a requirement that led to an incorrect implementation of the behavior.
- ➔ The debugging tool must therefore be coupled with a traceability tool capable of tracing the path from the requirement, through the specification down to the HW implementation.
 - Such a trace would for example show that the register could only be written during the boot mode

Locating the errors

- ➔ Magillem provides such an environment
- ➔ Linking the VP to the IP-XACT representation helps accessing data that is not available in the VP
 - such as datasheet or non- functional properties such as power, voltage or frequency.
- ➔ Linking the IP-XACT representation to the requirements and specification documents allows capturing such misinterpretation errors
 - that would have taken hours or days to locate otherwise.

Case study: overview

- ➔ Simple system that controls the execution of critical tasks
- ➔ System description
 - Each executing task is represented by a LED
 - the LED is ON (highlighted) when the task is active (i.e. running)
 - The LED is OFF when it is suspended or stopped.
 - An extra (red) LED is highlighted when an error occurs.

Case study: System Requirements

<u>Req number</u>	<u>Description</u>
SR-1.0	The system shall be based on the <u>FreeRTOS</u> that includes the concept of co-routines and tasks.
SR-1.1	Tasks will be used for the Terminal display and co-routines for the LED display.
SR-1.2	The application code running on the RTOS shall create five flash co-routines and three tasks.
SR-1.3	One extra task (the idle task) is responsible for launching all the co-routines.
SR-1.4	The system will run on a Micro-controller based on an ARM M3 processor, connected to a bank of 8 LEDS and to a UART controlling the Terminal display.

<u>Req number</u>	<u>Description</u>
R-L1.1	The LED shall be used to indicate the system status.
R-L1.1.1	A flashing green or yellow LED shall indicate that the system is running as expected
R-L1.1.2	A flashing red LED shall indicate a fault condition.
R-L1.1.3	The correct LED shall flash on and off once every second. This flash rate shall be maintained to within 50ms.

Case study: Refined Requirements

<u>Req number</u>	<u>Description</u>
R-L1.1	The LED shall be used to indicate the system status.
R-L1.1.1	A flashing green or yellow LED shall indicate that the system is running as expected
R-L1.1.2	A flashing red LED shall indicate a fault condition.
R-L1.1.3	The correct LED shall flash on and off once every second. This flash rate shall be maintained to within 50ms.

LED Requirements (sample)

<u>Req number</u>	<u>Description</u>
RR-L1.1	The flash co-routines control LED's zero to four.
RR-L1.2	LED five is toggled each time the string is transmitted on the UART.
RR-L1.3	LED six is toggled each time the string is correctly received on the UART
RR-L1.4	LED seven is latched on when an error is detected in any task or co-routine. The error is detected by a check function (called by the idle task) that loads the general purpose registers with a known value, then checks each register to ensure the held value is still correct. As a low priority task this checking routine is likely to get repeatedly swapped in and out. A register being found to contain an incorrect value is therefore indicative of an error in the task switching mechanism.

LED Functional Requirements (sample)

Case study: HW Specifications

Req number	Description
RR-S-2.1	When the SW writes to the UART Data Register (DR), an interrupt is raised that will launch a SW interrupt routine.
RR-L-2.1	The LEDs are implemented as an 8 bits register. Each bit represents a LED: a one means highlight is ON, a zero means it is OFF.
RR-L-2.2	The UART tasks are mapped to the bits 5 and 6, represented by a yellow LED.
RR-L-2.3	The error task is mapped to the bit 7 and represented as a red LED.
RR-L-2.4	The flash co-routines are mapped to the bits 0 to 4 and are represented by a green LED.

LED Hardware Requirements (sample)

LED Hardware Implementation (IP-XACT)

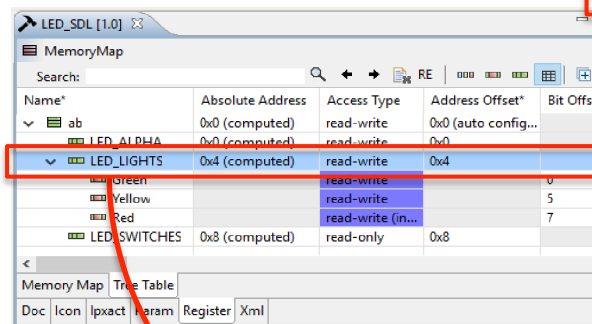
Name	Absolute Address	Access Type	Address Offset*	Bit Offset*
ab	0x0 (computed)	read-write	0x0 (auto config...	
LED_ALPHA	0x0 (computed)	read-write	0x0	
LED_LIGHTS	0x4 (computed)	read-write	0x4	
Green		read-write		0
Yellow		read-write		5
Red		read-write (in...		7
LED_SWITCHES	0x8 (computed)	read-only	0x8	



Case study: SW specifications

RR-L-2.1 The LEDs are implemented as an 8 bits register. Each bit represents a LED: a one means highlight is ON, a zero means it is OFF.

SW Driver for the LED peripheral



Name*	Absolute Address	Access Type	Address Offset*	Bit Offset
ab	0x0 (computed)	read-write	0x0 (auto config...	
LED_ALPHA	0x0 (computed)	read-write	0x0	
LED_LIGHTS	0x4 (computed)	read-write	0x4	
Green		read-write		0
Yellow		read-write		5
Red		read-write (in...		7
LED_SWITCHES	0x8 (computed)	read-only	0x8	

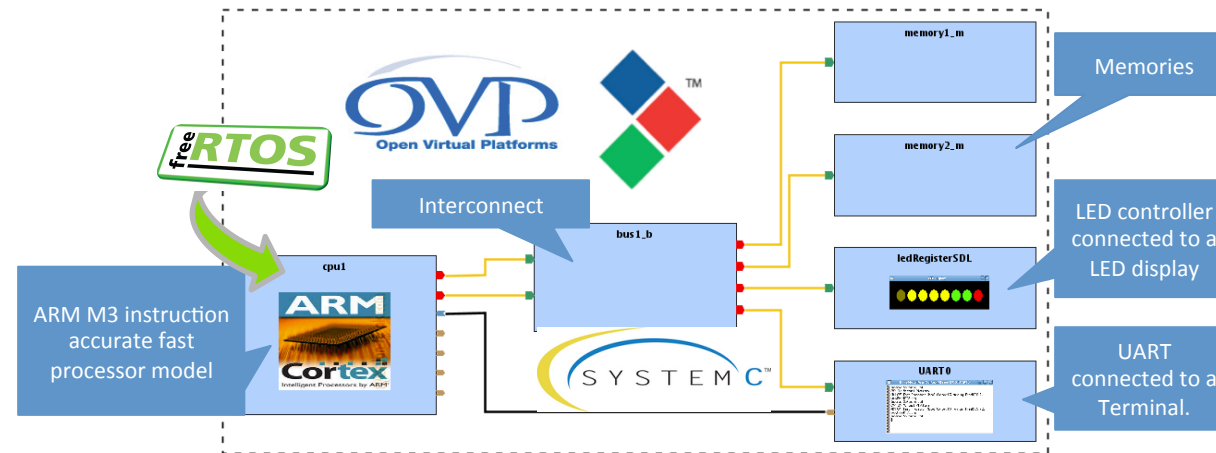
```
void LedInitialise( void ) {...}
void LedSet( unsigned int LED, boolean
value ) {
    unsigned char ucBit = ( unsigned char )
1;
    vTaskSuspendAll();
    {
        /* atomic section */
        ucBit = ( ( unsigned char ) 1 ) >> LED;
        if( ! value ) {
            ucBit ^= ( unsigned char ) 0xff;
            ucOutputValue &= ucBit;
        } else {
            ucOutputValue |= ucBit;
        }
        ledWrite(ucOutputValue);
    }
}
```

Hardware Abstraction
Layer (HAL) code,
usually part of the
Hardware BSP

```
#define LED_BASE_ADDRESS 0x40004000
#define LED() *((volatile char *)
LED_BASE_ADDRESS+4)
void ledWrite(unsigned char value) {
    LED() = value;
}
```


Case study: VP implementation

- VP platform based Imperas/OVP SystemC TLM models for each IP defined in the specification
- Each SystemC IP model has been automatically packaged in IP-XACT XML format



Case study: Links creation

- Links have been created between the VP (the HW/SW design part) and the Requirements.

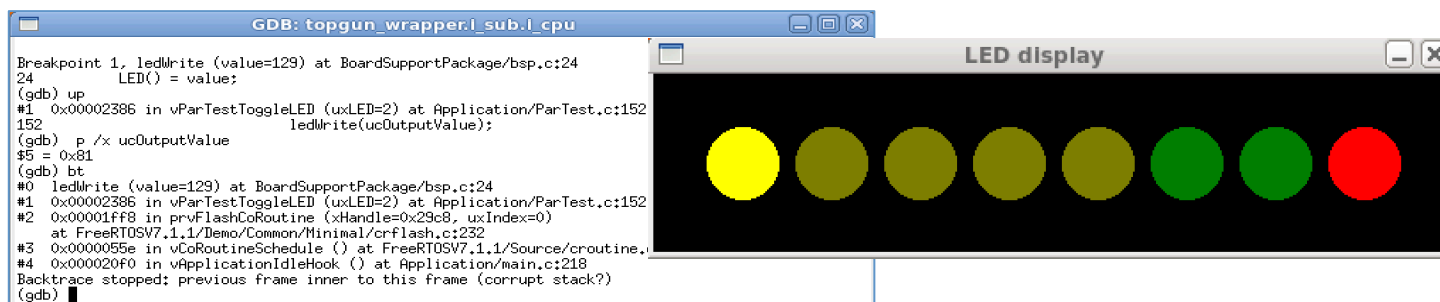
The screenshot displays the Eclipse IDE interface with several views open:

- Resources View:** Shows a project structure with folders like 'LED_demo_HW_requirements' and 'LED_demo_undefined_requirements'.
- Requirements View:** A table with columns 'Req number' and 'Description'.

Req number	Description
RR-S-2.1	When the SW writes to the UART Data interrupt is raised that will launch a SW
RR-L-2.1	The LEDs are implemented as an 8 bits represents a LED: a one means highlight it is OFF, Modification2
RR-L-2.2	The UART tasks are mapped to the bits by a yellow LED.
- Project Hierarchy View:** Shows a tree of project components including 'tm2bridge', 'tmDecoder', 'topgun_wrapper', 'topgun', 'UART_wrapper', 'UART', 'UARTinterface', 'wire_generator', and 'DesignConfigurations'.
- Diagram View:** A block diagram showing 'RTOS' and 'OVP SYSTEMC' components connected to 'LED' and 'UART' blocks.
- Console View:** Shows build output with warnings and errors.
- Hub View:** A diagram at the bottom showing links between 'LED_demo_undefined_requirements', 'LED_demo_HW_requirements', and 'LED_demo_undefined_requirements'.

Case study: System debug

- When simulating, we observe that the red LED is highlighted after some time.
- The VP flexibility allows to simply putting a breakpoint in both the SW and in the HW when the LED register is written and stop the simulation when the red LED is highlighted.



```
GDB: topgun_wrapper.l_sub.l_cpu
Breakpoint 1, ledWrite (value=129) at BoardSupportPackage/bsp.c:24
24      LED() = value;
(gdb) up
#1 0x00002386 in vParTestToggleLED (uxLED=2) at Application/ParTest.c:152
152      ledWrite(ucOutputValue);
(gdb) p /x ucOutputValue
$5 = 0x81
(gdb) bt
#0 ledWrite (value=129) at BoardSupportPackage/bsp.c:24
#1 0x00002386 in vParTestToggleLED (uxLED=2) at Application/ParTest.c:152
#2 0x00001ff8 in prvFlashCoRoutine (xHandle=0x29c8, uxIndex=0)
    at FreeRTOSV7.1.1/Demo/Common/Minimal/crFlash.c:232
#3 0x0000055e in vCoRoutineSchedule () at FreeRTOSV7.1.1/Source/croutine.c:
#4 0x000020f0 in vApplicationIdleHook () at Application/main.c:218
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) █
```

LED display

LED display showing 8 LEDs: Yellow, Green, Green, Green, Green, Green, Green, Red.

Case study: System debug

- ➔ Thanks to the link to the requirements, we can see that the LED seven is mapped to the Error condition.

RR-L1.4	LED seven is latched on when an error is detected in any task or co-routine. The error is detected by a check function (called by the idle task) that loads the general purpose registers with a known value, then checks each register to ensure the held value is still correct. As a low priority task this checking routine is likely to get repeatedly swapped in and out. A register being found to contain an incorrect value is therefore indicative of an error in the task switching mechanism.
---------	---

- ➔ And that in normal mode (no error) the value of the LED7 (bit7 of the LED register) should 0 (not 1).

RR-L-2.3	The error task is mapped to the bit 7 and represented as a red LED.
----------	---

Case study: Conclusion

- ➔ More debugging demonstrated that the SW side was the root of the error.

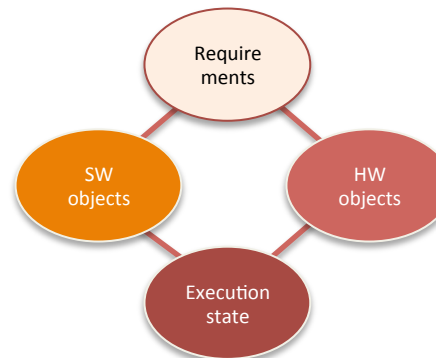


The location of the SW error was in the LED driver: shift right instead of shift left

```
void LedInitialise( void ) {...}
void LedSet( unsigned int LED, boolean
value ) {
    unsigned char ucBit = ( unsigned char )
1;
    vTaskSuspendAll();
    {
        /* atomic section */
        ucBit = ( ( unsigned char ) 1 ) >> LED;
        if( ! value ) {
            ucBit ^= ( unsigned char ) 0xff;
            ucOutputValue &= ucBit;
        } else {
            ucOutputValue |= ucBit;
        }
    }
    ...
}
```

RR-L-2.3	The error task is mapped to the bit 7 and represented as a red LED.
----------	---

Case study: Conclusion



- ➔ The error only shows up when linking together the Specification and the Implementation, and when executing the SW with the HW.
- ➔ A direct pointer to the requirement could immediately separate out the HW and the SW responsibilities, saving hours of debug and iterations between HW and SW teams.

Future work

- ➔ Such an integrated environment with immediate impact analysis to help debugging complex systems is even more useful when some requirement changes or when the spec changes or when the implementation changes (e.g. when a bug is fixed).
- ➔ Of course this assumes that the links have been properly created and that they fully cover the requirements. Additional techniques need to be developed to automate the creation of the links and to verify the links are complete.

Conclusion

- ➔ Combination of innovative traceability techniques with advanced VP execution environment helps locating SW errors by tracing the dependencies all the way through from requirements down to the embedded system execution
- ➔ ... and vice-versa.
- ➔ This is the beginning of a long avenue of developments to improve the consistency and coherency between the functional requirements and their implementation through early validation on VPs.

Thank you

