Public

FP7-ICT-2013- 10 (611146) **CONTREX**

Design of embedded mixed-criticality CONTRol systems under consideration of EXtra-functional properties

Project Duration	2013-10-01 – 2016-09-30	Type	IP

	WP no.	Deliverable no.	Lead participant
	WP3	**D3.2.3**	**EUTH**

Implementation of execution platform and cloud service models (final)

Prepared by	**Paolo Azzoni (EUTH)**
Issued by	**EUTH**
Document Number/Rev.	**CONTREX/EUTH/R/D3.2.3/1.0**
Classification	**CONTREX Public**
Submission Date	**2016-09-30**
Due Date	**2016-09-30**

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)

History of Changes

ED.	REV.	DATE	PAGES	REASON FOR CHANGES
EUTH	0.1	2016-09-02	48	Initial version
OFFIS	0.2	2016-09-14	56	Update OFFIS contribution
POLIMI	0.3	2016-09-23	58	Update PoliMi contributions
EUTH	0.4	2016-09-28	58	EUTH update
INTEL	0.5	2016-09-28	57	INTEL update
EDALAB	0.6	2016-09-29	57	EDALab update
ST	0.7	2016-09-30	63	ST update
ST-POLITO	0.8	2016-09-30	94	ST-Polito update
IX	0.9	2016-09-30	97	iXtronics update
EUTH	1.0	2016-09-30	97	Final release

Contents

1 Introduction

This deliverable describes the final implementation of the execution platform and of the cloud service layer. It focuses on the description of the hardware nodes composing the execution platform and on the cloud platform, the core element of the cloud services implementation. For each use case, the deliverable provides more details about the services definition and characterization, previously outlined in deliverable D3.2.2. UC1 and UC3 are based on OVP simulation environment, while the device-to-cloud paradigm, which combines the execution platform and the cloud service layer, represents the technological baseline for UC2 scenario. Finally, the deliverable describes the adopted models and tools, outlining their relations with the runtime environment management and with the extra functional properties monitoring.

2 The execution platform description

2.1 iNemo-M1

The INEMO-M1 is a 9-degrees-of-freedom system-on-board (SoB), combining the latest advances in ST MEMS-based technology with the powerful computational core of theSTM32 family. The INEMO-M1 platform has been designed to target miniaturization, flexibility, low power consumption and cost effectiveness to provide a high-performance and versatile module suitable for a wide range of applications.

Figure 1. The iNEMO-M1 SOB from STMicroelectronics

INEMO-M1 embeds the 6-axis digital e-compass module LSM303DLHC, the 3-axis digital gyroscope L3GD20 and the STM32F103REY6 high-density performance line ARM-based32-bit microcontroller.

The INEMO-M1 exploits the wide set of peripherals supported by the STM32F103REY6 in order to offer maximum flexibility in communication. Thanks to this wide set of communication peripherals and its extremely reduced dimensions, the INEMO-M1 can be directly integrated into a large variety of advanced motion-sensing platforms allowing simplification of the platform itself and increasing performance by distributing system intelligence.

The INEMO-M1 may be used in combination with ST’s leading edge sensor fusion software to seamlessly implement high-performance 9-degrees-of-freedom applications.

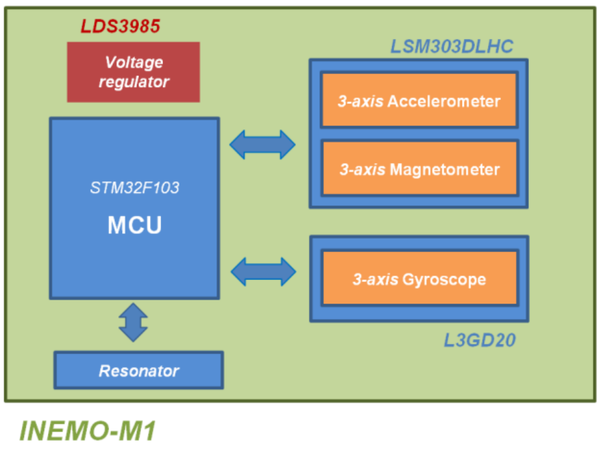


Figure 2. iNEMO-M1 functional block diagram

It can operate with an externally regulated power supply in the range of 2.4 V- 3.6 V or, by using the on board LDS3985M33R voltage regulator, it could be supplied by an external voltage in the range of 3.6 up to 6 V.

2.1.1 iNEMO M1 – STM32F103 MCU

The STM32F103REY6 high-density performance line microcontroller is the computational core of the INEMO-M1 module: it operates as the system coordinator for the onboard sensors and the several communication interfaces.
It is based on ARM Cortex™-M3 processor core, the latest generation of ARM processors for embedded systems. With its 72 MHz maximum frequency, a 512-Kbyte embedded Flash and a 64-Kbyte SRAM accessed (read/write) at CPU clock speed with 0 wait states, it is suitable for storing programs and data.
Exploiting the features of the MCU, the INEMO-M1 offers a wide set of peripherals and functions such as 12-bit ADCs, DAC, general-purpose 16-bit timers plus PWM timers, I2C, SPI, I2S, USART, USB and CAN, that enable different operative conditions and several communication options, making the module a flexible solution for effortless orientation estimation and motion-tracking in embedded applications.

2.1.2 iNEMO-M1 Sensors

The INEMO-M1 embeds two sensors: the LSM303DLHC, 3-axis accelerometer and 3-axis magnetometer in one single package, and the 3-axis digital gyroscope L3GD20.
The e-compass module LSM303DLHC is a system-in-package featuring a 3D digital linear acceleration sensor and a 3D digital magnetic sensor. The accelerometer has full scales of ±2g/±4g/±8g/±16g and the magnetometer has full scales of 1.3/±1.9/±2.5/±4.0/±4.7/±5.6/±8.1 gauss. All full scales available are selectable by the user. The L3GD20 is a low-power digital gyroscope able to sense the angular rate on the three axes. It has a full scale of ±250 / ±500 / ±2000.

2.1.3 Discovery M1 application board

The Discovery M1 opens the INEMO-M1 ecosystem to the worldwide developer community. It offers a means to explore and evaluate the INEMO-M1, providing complete access to all the benefits of the first 9-axis system-on-board (SoB) in the iNEMO® module family. The accelerometer, gyroscope, magnetometer and ARM® Cortex™-M3 32-bit MCU are fully accessible to the user firmware, enabling the implementation of complex and versatile inertial applications in fields such as robotics, personal navigation, gaming, and wearable sensors for healthcare/fitness or sport kinetics.

Figure 3. Discovery M1 evaluation board

The pin-out of the two on-board connectors offers immediate access to all of the communication interfaces and peripherals of the INEMO-M1.

2.1.4 Discovery M1 Key Features

- Two power supply options: through the USB bus or from one of two external supply voltages, VEXT (from 3.6 to 6 V) or D_VDD (from 2.4 V to 3.3 V)
- INEMO-M1: 9-axis SoB, 13x13x2 mm size factor
- LPS331AP: MEMS pressure sensor 260-1260 mbar absolute digital output barometer
- INEMO-M1 pinout available on two double connectors
- SWD connector for programming and debugging.
- Two pushbuttons (reset and user)
- Two LEDs: user LED, power-on LED
- RoHS compliant

2.1.5 Software development kit

A software package has been developed to fully support the application design activity on iNEMO platform, including a Software Development Kit for sensors configuration and the source code of Kalman filter algorithm implementation for attitude angles reconstruction based on data measured by integrated inertial motion MEMS sensors.

A list of provided software tools and firmware libraries is detailed below:

- A GUI (Graphical User Interface) to exploit the Discovery M1 board features and display the sensor data
- An SDK (Software Development Kit) to develop personalized PC applications
- A Compass application demo
- A 3D Cube demo to show the AHRS (Attitude Heading Reference System) capability of iNEMO Engine Lite
- A complete set of firmware libraries to manage the INEMO-M1 on-board sensors
- A firmware framework based on FreeRTOS™

2.2 SecSoc

STM provides a SoC specifically intended for image processing captured by a low-power CMOS camera. The architecture of the SoC (shown in Figure 4) extends the typical structure of a high-end microcontroller with specific modules for image processing, ultra-low power analog modules, and power island and clock gating capabilities. The SoC is based on the multi-core R4MP processor.

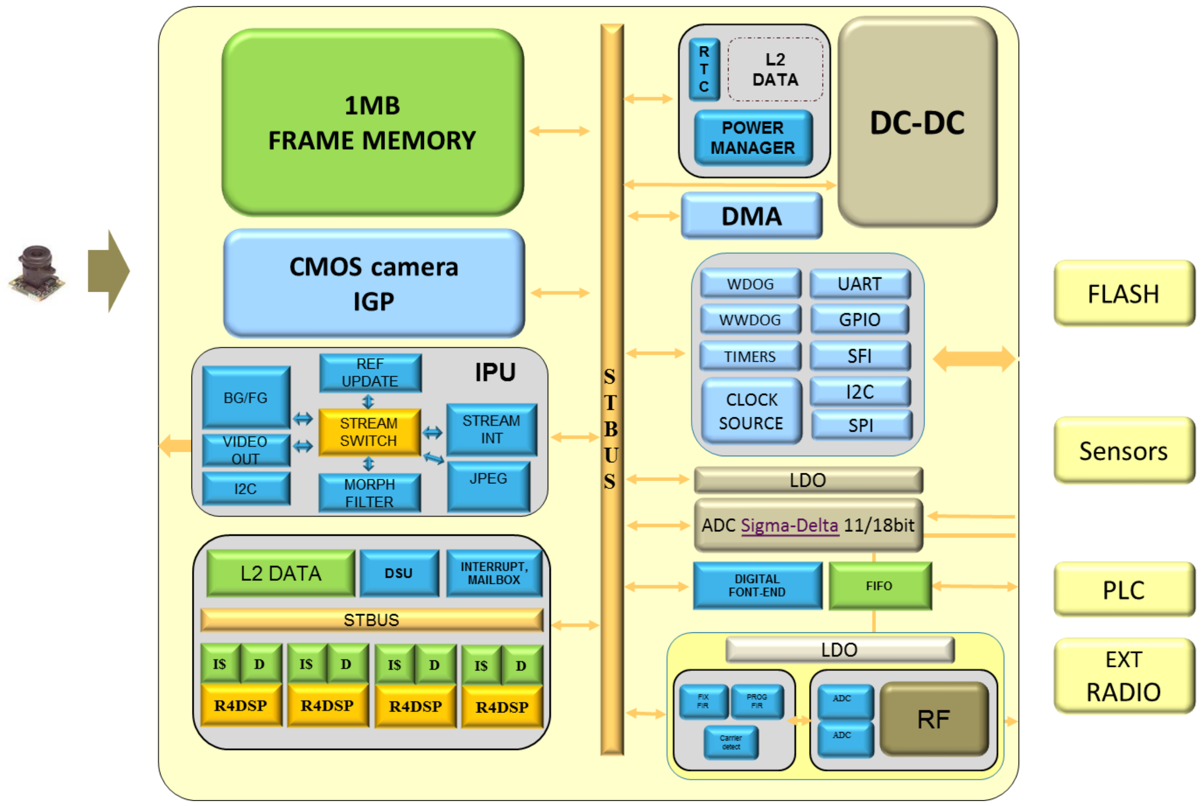


Figure 4: SeCSoC architecture.

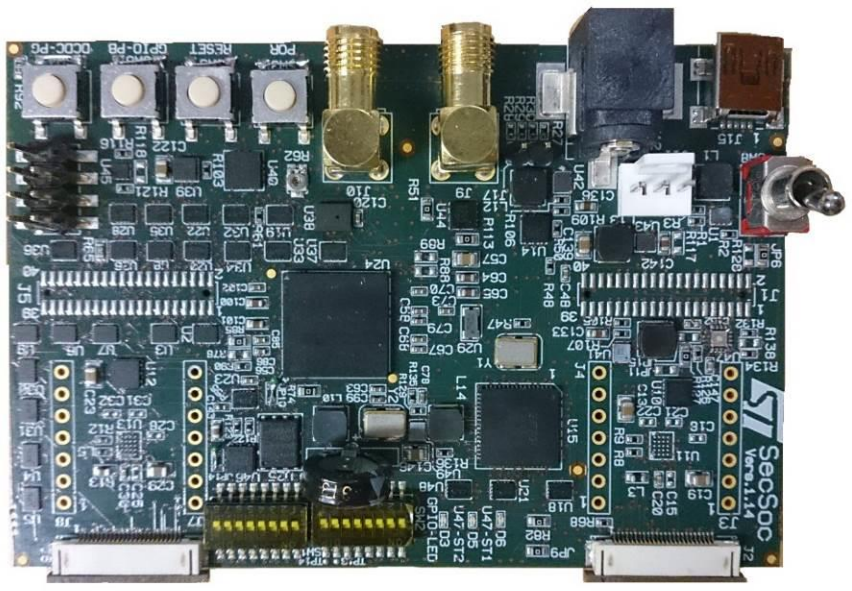


Figure 5: SeCSoC board.

The board that has been provided has a size similar to a credit card, and it includes two vga sensors, optional sensors connectors, usb, spi, uart, jtag host connectivity, a digital stereo microphones and pressure and temperature sensors. The adoption of multi-core technology also in some of the sensing units allows for an improved management of extra-functional aspects such as power consumption, execution time, quality of service, security and reliability. The proprietary cores can be programmed through a development toolchain (gcc+GNU binutils based) released by ST. Moreover, peripheral library APIs are available.

2.3 Zynq

The Xilinx Zynq-7000 family is a rich System on Chip (SoC) which combines a heterogeneous ARM Cortex-A9 MPCore and a programmable logic that is realized through an FPGA part in the chip. Thus, the chips of the Zynq-7000 family are representing an all programmable SoC with highest flexibility in programming the hardware, software and the inputs and outputs, which makes them interesting for a wide range in on-chip design space exploration.

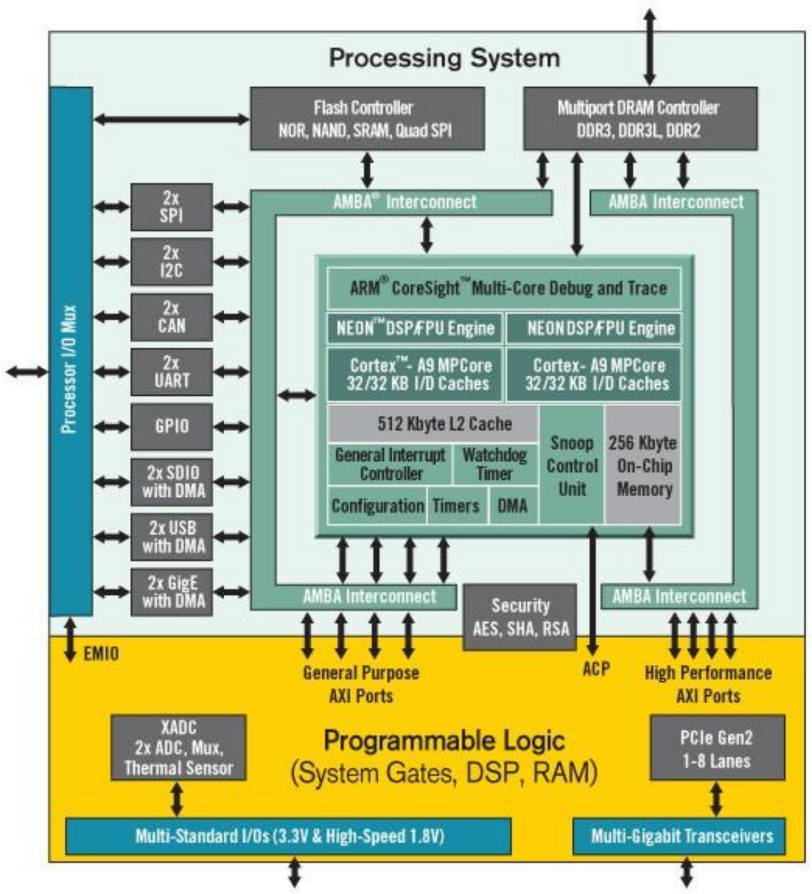

Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
AMBA® Interconnect
AMBA Interconnect
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure 6: Overview of Zynq-7000 architecture.

Figure 6 shows a rough overview of the SoC's architecture. At the top of the figure the processing system is presented with its ARM dual core in the center surrounded by the AMBA interconnect. This interconnect defines the connections to the I/O interfaces, the Flash and DRAM controllers as well as to the programmable logic. The hardware I/O interfaces can be configured and connected to pins by the user. While the programmable logic is also tied to the AMBA interconnect the ability to build further interfaces is given to the user. There is also the possibility to use the programmable logic to instantiate other IP cores, e.g. softcores, DSPs, etc. With the SDIO or flash interface it is possible to boot a Linux kernel or other operating systems on the ARM cores as well as loading bit streams and applications for the programmable logic. These possibilities make the Zynq-7000 family to a very powerful platform which has been be used for use-case 1.

2.3.1 Evaluation board selection

Within CONTREX no new PCB for the targeted controller will be designed. Instead, an existing board has been chosen to be used within the use cases. Figure 7 shows the industrial board which has been used for the multi-rotor system in use-case 1. The board occupies one ynq 7020, which operates at up to 766MHz for the ARM dual core and that offers 85k logic cells within the programmable logic. This version of the SoC can be used with the free WebPACK license of the Xilinx tools. Next to the Zynq the 40x50mm industrial board contains 1GB DDR3 RAM and offers access to 152 I/O ports for the programmable logic as well as 14 MIOs (Multiplexed I/Os) of the processing system. This industrial board has been put on a carrier board which provides the hardware interfaces for the connection to the other components of the multi-rotor system.

Figure 7: Industrial board TE0720-02-2IF from Trenz Electronic [2]

2.3.2 Run-time power management and resource management capabilities of selected platform

In the context of CONTREX the Zynq provides mechanisms for run-time power management as well as for resource management capabilities.

Regarding power management, one of these capabilities is the dynamic frequency scaling (DFS). This feature of the processing systems gives the availability to control the frequency of the ARM dual core or the programmable logic. Together with an external voltage regulator which is configurable on the fly by the operating system it would be possible to realize a system with dynamic voltage and frequency scaling (DVFS). The same method can be used for the external DDR RAM or peripherals. It is also possible to shut down the unused clocks for saving power. If an asymmetric processing of the ARM cores is used, it is possible to use CPU hotplug. With this feature it is possible to bring up or shut down the secondary processor core if it is or is not needed at the time. A wake up of a core can be done by an interrupt for example.

Regarding resource management, a Linux kernel has been used on the ARM cores offering Linux control groups including the subsystems *cpu*, *cpuset*, *memory*, and *cpuacct*. Thus, it should fulfil all requirements of the Barbeque Open Source Project[3] that is envisioned to be used in the project.

2.4 EUTH Minigateway

The EUTH Minigateway is a compact size device designed to support M2M (Machine to Machine) applications where low cost, low power, performances and communications capabilities are required. It has been conceived as an industrial grade smart device that provides full support to Kura framework for M2M systems integration in industrial and automotive applications. The device has been designed to be RoHS compliant and to satisfy the following standards: CE, FCC, IC, J1455, UL.

The Minigateway is a component of the execution platform adopted for the Automotive use case (UC2). In this context it acts as a bridge between the "sensing" units installed in the car and the cloud platform: the iNemo detects crashes and sends the related information to the Minigateway that, in turns, controls the SecSoc for the photos acquisition/elaboration. Finally, the Minigateway uploads the data on the cloud platform. The first release of the UC2 demonstrator has been based on the first set of EUTH Minigateway prototypes, while the final version is based on the second release of the Minigateway, already introduced in D4.2.1. The following figure illustrates the role of the Minigateway in the UC2 demonstrator.

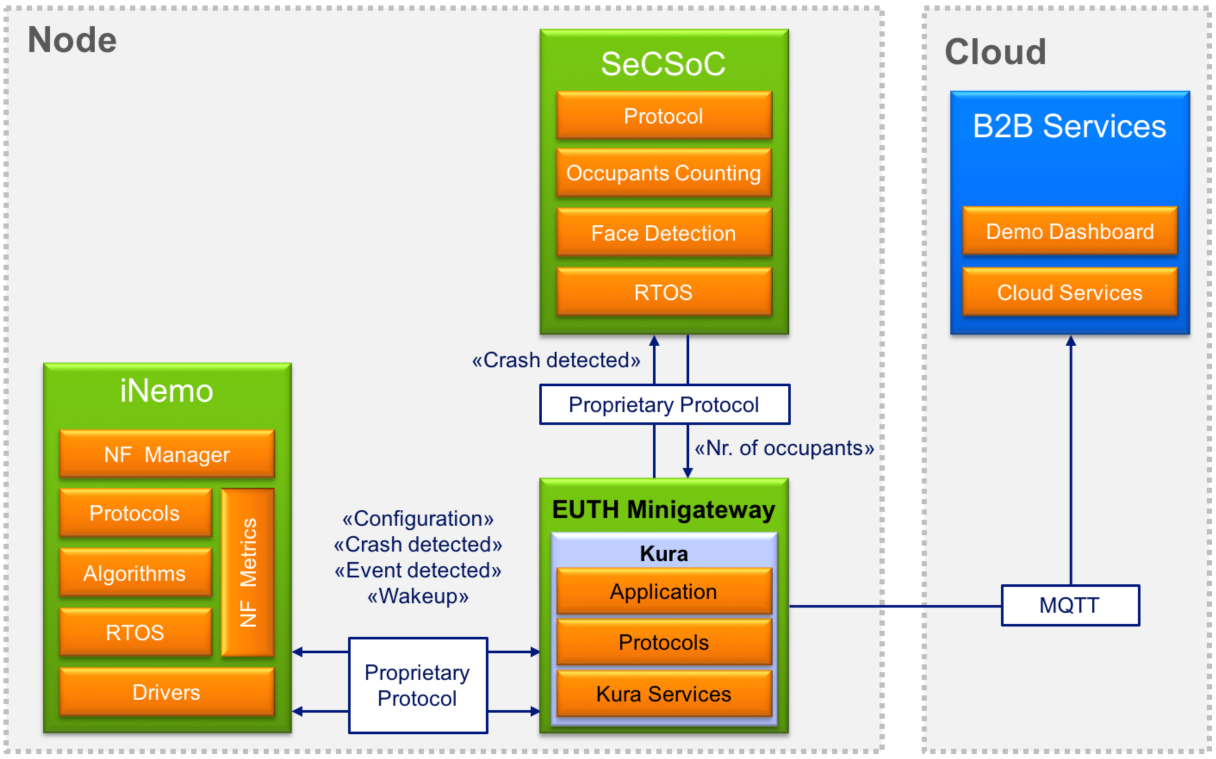


Figure 8: EUTH Minigateway in the UC2 demonstrator.

The EUTH Minigateway is based on a powerful ARM8 microprocessor, high speed DDR3 RAM, Flash Memory and a rich set of communications interfaces. An expansion connector supports additional peripherals such as LCD display, keyboard, A/D interfaces and much more.

The device is supported by a broad range of operating systems, but in the UC2 context the selected operating system is POKY Linux, which is part of the Yocto Project. Yocto is an open source collaboration project that provides templates, tools and methods to simplify the creation of custom Linux-based systems for embedded products, regardless of the hardware architecture. (https://www.yoctoproject.org/).

The following figure shows the board of the ETH Minigateway.

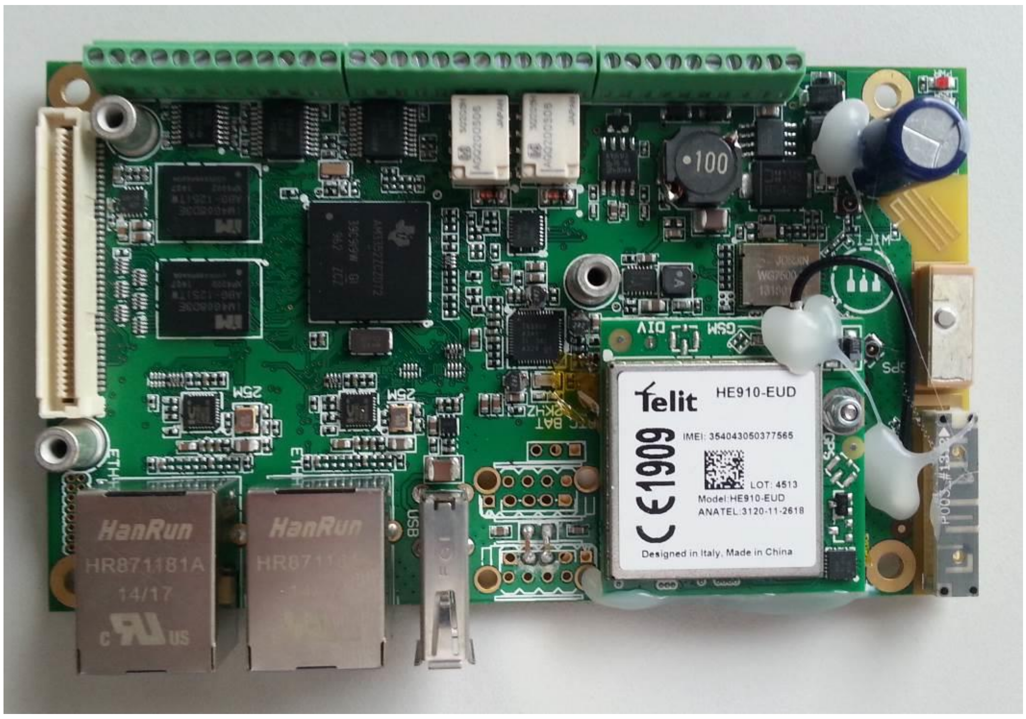


Figure 9: The industrial board of the Minigateway from Eurotech.

The functional architecture of the Minigateway is illustrated in the following block diagram.

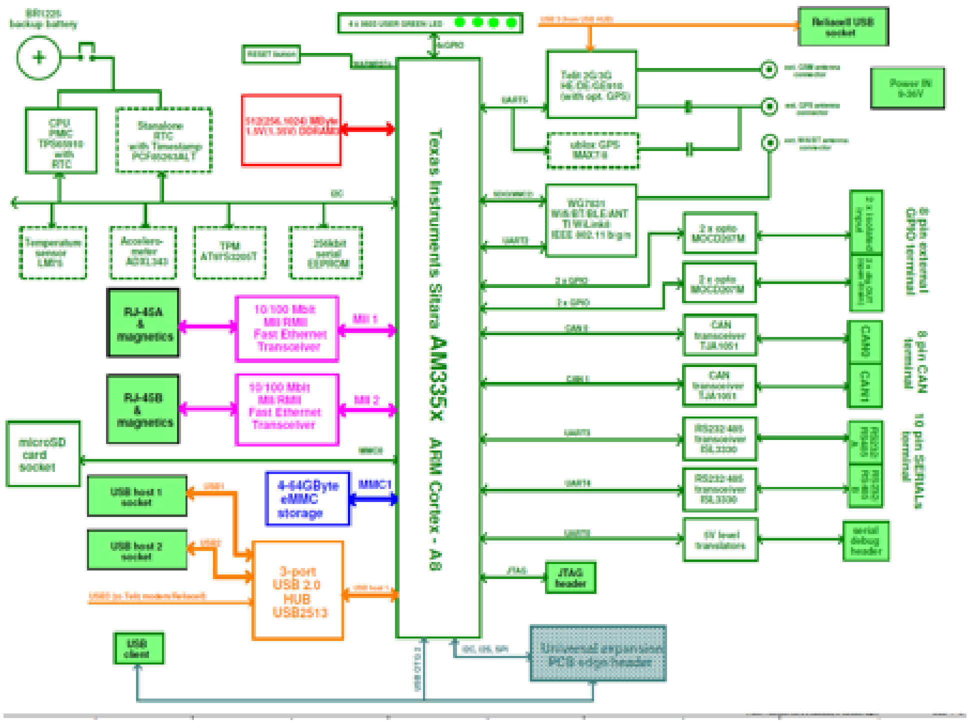


Figure 10: The functional architecture of the Minigateway.

The CPU is an integrated system-on-a-chip microprocessor for high-performance, low-power portable handheld and handset devices, the TI AM335X, an ARM Cortex A8 from Texas Instruments. It incorporates on-the-fly voltage and frequency scaling and sophisticated power management. The characteristics of this system on a chip, its power consumption and thermal profiles make the AM335X a good solution for an Automotive context. I.e., this low power device does not require a heat sink for temperatures up to 70°C (85°C for the industrial variant). The Minigateway uses an internal real time clock (RTC) to store the date and time and provides power management events. The real time clock is mapped into the Linux operating system and can be read or written using standard Linux utilities. An internal watchdog timer, from the AM335x CPU, can be used to protect against erroneous software.

The board supports two types of memories:

- Up to 512MB resident NAND FLASH disk containing:
  - Boot loader: Uboot to boot embedded Linux
  - Embedded Linux
  - Application images.
- Up to 512MB of DDR3 SDRAM for system memory. Additionally, the unit can support a microSD and various USB storage devices.

The AM335x processor contains an integrated LCD display controller, which can be used to control and on-board LCD Display.
Considering the role and functionalities required by UC2 it is important to provide a rich set of interfaces and communication channels. The device must be ideally suited for M2M and IoT (Internet of Things) applications, which means connecting sensors, actuators and devices to the enterprise application and services. For this purpose, several interfaces are provided: USB, CAN Bus, Serial Communication Ports, GPIO. Furthermore, the Minigateway must be a connected device and a powerful combination of interfaces for a broad range of communication connectivity applications has been included: Ethernet, Bluetooth, Wifi, GPRS and 3G.

2.4.1 EUTH Reliagate

EUTH is currently orienting the engineering activities to a new modular approach in industrial and automotive service gateway design that extends the features of the Minigateway to a more modular and flexible system. The version 2 of the Minigateway represented the first step in this direction because, e.g., it is already compatible with external cellular modules that increases its communication potentialities. Starting from the Minigateway design, the result of this engineering activity is Reliagate 10-11, a new industrial and automotive multiservice gateway that has been recently included in EUTH products portfolio. The following figure illustrates the Reliagate 10-11 [10] and the ReliaCell [11] cellular module.

The Reliagate is ARM Powered (TI AM335X CPU) with optimum performance for intensive workloads at just 2W. It is IoT Ready, providing a rich set of connectivity options, field bus interface and on-board sensors. It natively supports Kura and it's commercial version ESF [12]. It is an industrial and automotive grade multiservice gateway, with support for a wide operating temperature, a wide range power supply with transient protection and compliancy with several standards: CE, FCC, E-mark, IC 60950-U, RoHS2, REACH, FCC, PTCRB, IP40, MTBF >40000h or 5 years. The Reliagate is globally deployable, because the ReliaCell is a global cellular module with leading carrier pre-certifications.

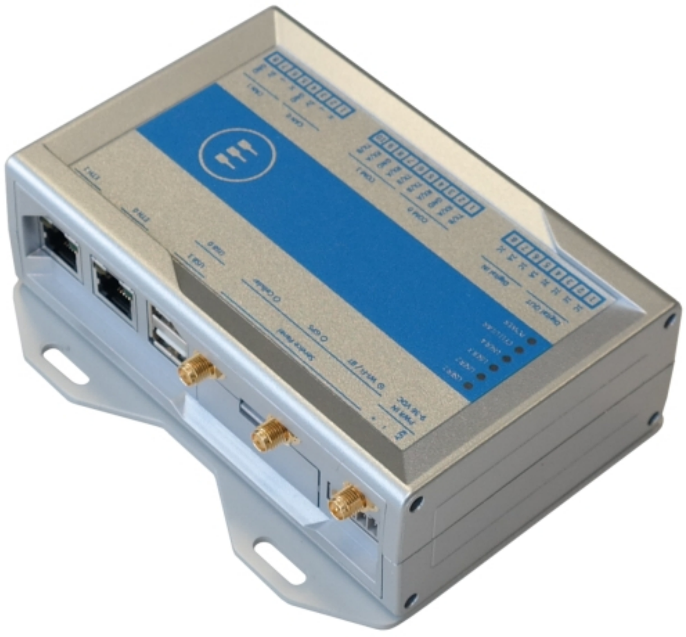


Figure 11: the final version of Reliagate 10-11.

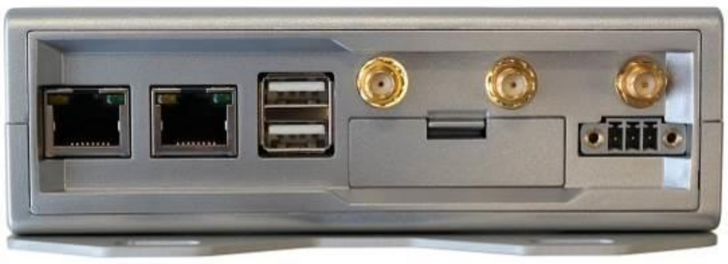


Figure 12: Reliagate 10-11 rear panel view.

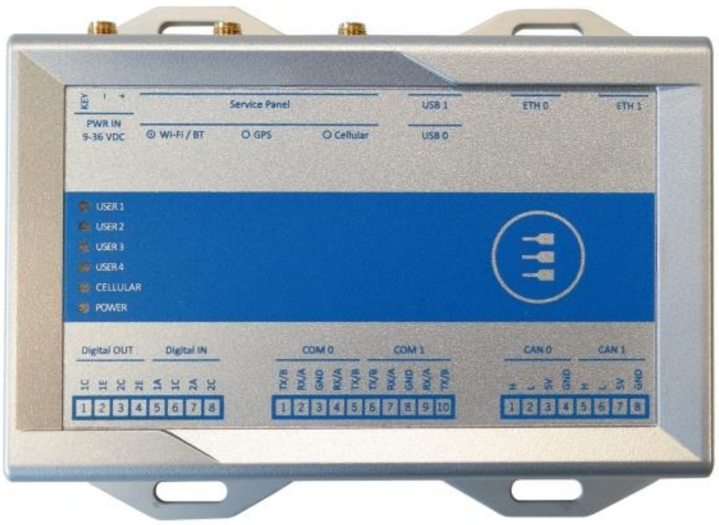


Figure 13: Reliagate 10-11 top view.

Figure 14: EUTH Reliacell cellular module.

2.4.2 EUTH Reliagate Family specification

Following the new modular and flexible design approach, an entire family of multiservice gateways has been introduced in the project portfolio. The following table illustrates the specification of the Reliagate family.

REGATE-10-11-XX

XX=		-31	-32	-33	-35	-36
PROCESSOR	**CPU**	TI AM3352 800MHz, 1 core				
MEMORY	**RAM**	512MB DDR3				
STORAGE	**Embedded**	4GB eMMC				
	Other	1x microSD slot (User Accessible)				
I/O INTERFACES	**Ethernet**	1x 10/100Mbps		2x 10/100Mbps	1x 10/100Mbps	2x 10/100Mbps
	USB	• 2x USB 2.0 (Noise and Surge Protected) • 1x USB 2.0 for ReliaCELL	• 2x USB 2.0 (Noise and Surge Protected)		• 2x USB 2.0 (Noise and Surge Protected) • 1x USB 2.0 for ReliaCELL	
	Serial	• 2x RS-232/RS-485 (Surge protected, RS-485 termination and fail-safe resistor) • 1x Serial console TTL				
	CAN 2.0B	No	2x CANBus 100mA with 5V power out			
	Digital I/O	• 2x Digital Input 5V (TTL), 1KV optoisolated • 2x Digital Output (50VDC), 10mA sink, 1KHz max switching				
RADIO INTERFACES	**Cellular**	Optional with ReliaCELL 10-20	3G global (integrated)		Optional with ReliaCELL 10-20	
	GPS	Optional with ReliaCELL 10-20	Included in 3G module		Optional with ReliaCELL 10-20	
	Wi-Fi / BT	No		802.11b/g/n + 4.0 BLE	No	802.11b/g/n + 4.0 BLE
	Antennas (external)	No	• 1x SMA Cellular • 1x SMA GPS	• 1x SMA Cellular • 1x RP-SMA Wi-Fi/BT • 1x SMA GPS	No	• 1x RP-SMA Wi-Fi/BT
OTHER	**SIM Slot**	No	1x microSIM		No	
	Ext. Watchdog	Yes				
	RTC	Yes, with user accessible battery				
	TPM	Factory option				
	EEPROM	256kbit				
	Sensors	Temperature		Temperature / Accelerometer	Temperature	Temperature / Accelerometer
	LEDs	• 1x Power • 4x Programmable (2x Green, 2x Amber)	• 1x Power • 1x Cellular • 4x Programmable (2x Green, 2x Amber)		• 1x Power • 4x Programmable (2x Green, 2x Amber)	
	Buttons	1x Reset, 1x Programmable				
POWER	**Input**	• 9-36VDC (Nominal 24VDC) with transient protection				

		• Vehicle Ignition Sense
	Consumption	2 W (idle)
ENVIRONMENT	**Operating Temp.**	-20 to +70°C
	Storage Temp.	-40 to +85°C
CERTIFICATIONS	**Wi-Fi/BT Radio**	• CE - EN300 328 (2.4 GHz ISM), EN50371 (EMI), EN301489 (EMC) • FCC - 15.209 (General RF device), 15.247 & 5.249 (2.4GHz ISM)
	Cellular	• PTCRB • FCC PART 22, 24 & 27 and suitable GSM radio certifications
	Ingress	IP 40
MECHANICAL	**Enclosure**	ABS - Color: aluminum
	Dimensions	140x95x45mm (WxDxH)
	Weight	160 g
	Mounting	Mounting Brackets (Wall, DIN Rail, General Purpose Bracket)
SOFTWARE	**OS**	Yocto Linux 1.6 - Kernel 3.14
	SDK	Yocto-based Eclipse Tooling
	IoT Framework	Everyware Software Framework (Java/OSGi)

3 The cloud service abstraction

3.1 Device to cloud

Machine to Machine (M2M) and Internet of Things (IoT) follow a common technological paradigm: intelligent devices, seamlessly connected to the Internet, enable remote services and provide actionable data. The IoT acronym is more adopted in the consumer space while M2M has a stronger industrial connotation. UC2 Automotive scenario can be considered a M2M application.

One of the most important aspects of the "Internet of Things" (IoT) vision is that smart objects communicate effectively with each other and with applications residing in data centers or the cloud.

The concept of the “device to cloud” proposes an end-to-end solution that includes:

- purpose-built hardware, connectivity and embedded device management through a pervasive software framework and a cloud client, running on the devices,
- and a set of machine to machine (M2M) cloud-based services.

The objective of this solution is to deliver actionable data from the field to downstream applications and business processes, dashboards and reports.

The concept of “device to cloud” is fundamental for UC2 scenario and, more in general, for similar contexts because for today’s business it is increasingly important to have constant visibility of assets and processes, anytime and anywhere.

The Kura pervasive framework and the cloud platform, proposed in Contrex project, offers the technical building blocks required to assemble distributed systems of devices and sensors which are effectively connected to IT infrastructures. This solution is based on a combination of hardware, firmware, operating systems, programming frameworks that dramatically accelerate the time to market of M2M / IoT projects and enable future potential customers to layer their added-value components on a reliable read-to-use infrastructure.

Regarding UC2 use case, a decoupled solution based on the device-to-cloud approach has been adopted: the vehicle ECUs are responsible for data acquisition and are physically connected to a Minigateway and thus to the cloud. The Minigateway acts as a bridge between the ECUs and the cloud and hosts the Kura framework. The following figure illustrates the two “device to cloud” solutions that has been adopted in UC2.

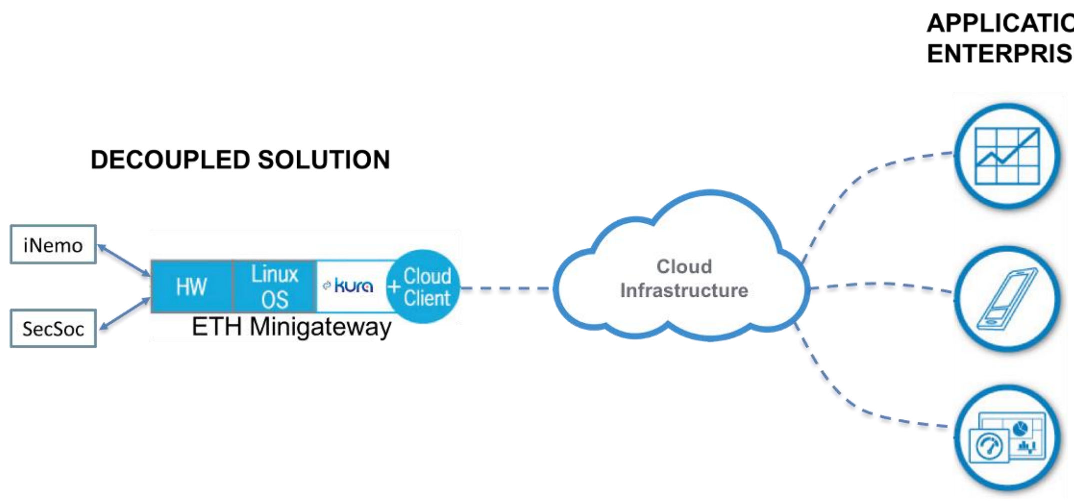


Figure 15: The device to cloud solution adopted in UC2.

3.1.1 Kura software framework

Eurotech's Kura is an inclusive, pervasive and targeted software framework for embedded systems. Kura is a programming environment that wraps the complexity of low level device management with high level constructs. This approach permits simpler and faster programming, with shorter, easier to read code and transparent portability across different hardware platforms.

Kura allows to deliver not only the latest generation pervasive computer hardware platforms, but also all aspects of the required software stack:

- a bootloader/BIOS for hardware platform,
- an operating system,
- a Java Virtual Machine,
- an OSGi application framework,
- and an extensive set of ready-to-use Java Plug-Ins (called bundles) provided for unique platform supplied hardware, network, cellular, and storage applications.

All of these components are "integrated" in Kura and are based on open standards, tools, and implementations. The amount of third party development is greatly reduced, allowing customers to focus on what they do best – writing their business application logic without having to spend man-years developing the software infrastructure and frameworks upon which those applications can run.

3.1.2 The Kura architecture

From a functional point of view, the Kura architecture is illustrated in the following figure:

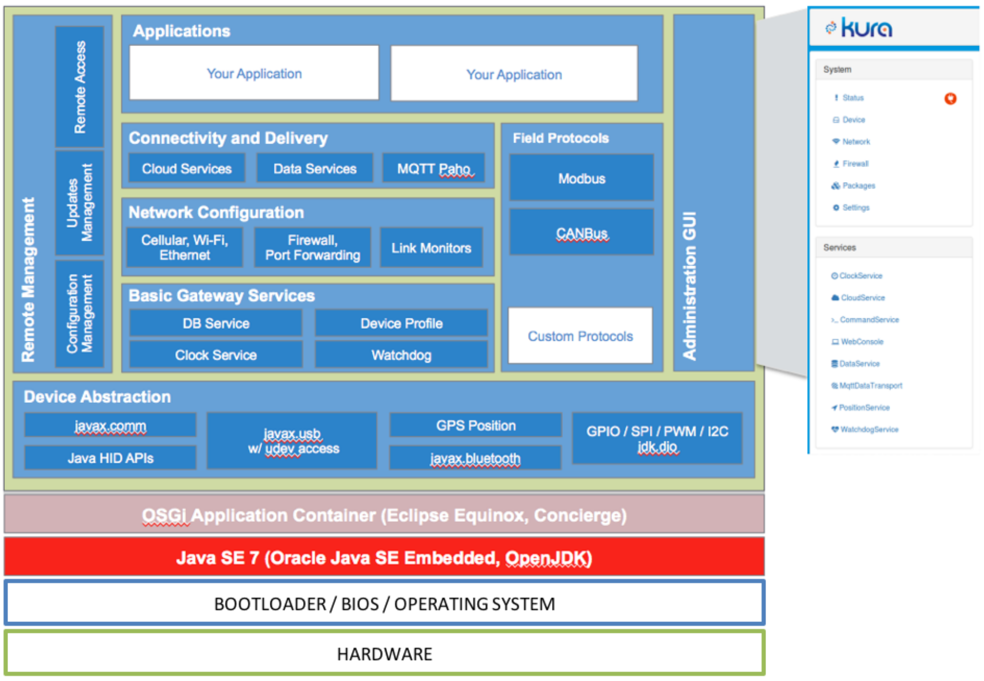


Figure 16: The Kura functional architecture.

Kura has been conceived to support a wide set of hardware devices and the choice of distributing it as an open source project is strategic from this point of view. In the context of the automotive use case (UC2) the adopted hardware is the ETH Minigateway. Many other embedded systems are supported, from industrial embedded systems to popular gateways like the Raspberry Pi or the BeagleBone. The higher-level Kura components are designed to be supported across platforms, independent of board-level differences in the hardware platform. Low-level hardware drivers and components are provided as part of Kura's platform-specific software core.

Bootloader and/or BIOS for each hardware platform are ported and validated by Eurotech as part of the platform-specific software core. The Bootloader/BIOS provides the start-up of the operating system.

With this approach, various operating systems can be ported to the hardware platforms and validated. For the EUTH Minigateway a custom Yocto-based Linux has been selected.

The core technology of Kura is the Java Virtual Machine (JVM). Java runs in a virtual machine runtime environment and thus runs across all operating systems. This allows code reuse between systems and applications following the principle "write once, run anywhere". The Java Native Interface (JNI) provides the hooks for native 'C' code interfaces where custom hardware or performance is required. Kura currently uses the Oracle Java SE Embedded Virtual Machine.

OSGi [13] originally stood for Open Services Gateway initiative and was designed as a services gateway for set-top boxes. Since its introduction ten years ago, it has become broadly adopted by industry as a whole and is now just referred to as "OSGi". OSGi is an Application Framework that provides the environment for running many different applications implemented with a strong component-based SOA oriented model. The OSGi Framework provides a set of

comprehensive and advanced features for installing, starting, stopping, updating, and uninstalling applications dynamically.
Kura runs on OSGi Equinox implementation, from the Eclipse Foundation.

Kura provides the core functionality for getting started with application development using documented APIs. This includes:

- hardware Virtualization (most applications don't care what hardware they are running on. Components are written to operate independently from the hardware platform.).
- Device Configuration Management.
- System Logger (Real-time system logging of debug and runtime information).
- Management of Network Devices (Ethernet, Wi-Fi, Cellular, and Bluetooth).
- IP Networking (DHCP server/client, DNS proxy, Firewall, and Time synchronization).

As project requirements expand beyond the basic Foundation set of bundles, Kura offers targeted vertical market bundles that can provide additional functions and features according to the requirements of the customer. These include a growing set of APIs that are ready to use or to customize for customer applications: location tracking, mobile asset management, Passenger/People Counter, Industrial Protocol (SCADA) Communication (e.g. Modbus, CanBus), MQTT or COAP Publish/Subscribe broker technology, Cloud-based asset and data management infrastructure, etc..

Kura is written around standard Java interface APIs that can be easily incorporated into a specific business application logic. Many standard implementations are available to use as-is, but APIs can be re-implemented according to specific application requirements. It is also possible to decide to add custom sets of bundles that run in addition to Eurotech's growing set of functional bundles.

The Eclipse Integrated Development Environment (IDE) is the dominant application environment in the industry today. The Eclipse IDE provides Kura with all of the tooling required to develop, test, debug, and package application bundles and to deploy these bundles to the Kura environment on every target platform. Eclipse provides a rich set of capabilities for writing applications in Java, C, Javascript/AJAX, and many others. Furthermore, Kura has been released as open source in the context of the Eclipse Development Program.

3.1.3 The Kura Abstraction Layer

The Kura abstraction layer is more than a simple hardware abstraction layer. Kura is a programming environment that wraps the complexity of low-level device management with high level constructs. With this approach the hardware abstraction is just the first level of abstraction of the embedded system: more high-level and application oriented abstraction levels are provided. These levels translate in services that simplify and speed-up the software development: data services, cloud services, configuration services, remote management services, web services, etc.. A rich set of services that abstract the complexity of the embedded system is a key element for the adoption of the "device to cloud" paradigm.

From a development point of view, Kura aims at offering a Java/OSGi-based container for M2M applications running in service gateways. It provides or, when available, aggregates open source implementations for the most common services needed by M2M applications. Kura components are designed as configurable OSGi Declarative Service exposing service API and raising events. While several Kura components are in pure Java, others are invoked through JNI and have a dependency on the Linux operating system.

Kura is currently providing the following initial set of services:

- I/O Services
  - Serial port access through javax.comm 2.0 API or OSGi I/O connection.
  - USB access and events through javax.usb, HID API, custom extensions.
  - Bluetooth access through javax.bluetooth or OSGi I/O connection.
  - Position Service for GPS information from a NMEA stream.
  - Clock Service for the synchronization of the system clock.
  - Kura API for GPIO/PWM/I2C/SPI access.
- Data Services
  - Store and forward functionality for the telemetry data collected by the gateway and published to remote servers.
  - Policy-driven publishing system, which abstracts the application developer from the complexity of the network layer and the publishing protocol used. Eclipse Paho and its MQTT client provide the default messaging library used. In addition, support for the Cobra field protocol has been provided.
- Cloud Services
  - Easy to use API layer for M2M application to communicate with a remote server. In addition to simple publish/subscribe, the Cloud Service API simplifies the implementation of more complex interaction flows like request/response or remote resource management.
  - Allow for a single connection to a remote server to be shared across more than one application in the gateway providing the necessary topic partitioning.
- Configuration Service
  - Leverage the OSGi specifications Configuration Admin and MetaType to provide a snapshot service to import/export the configuration of all registered services in the container.
- Remote Management
  - Allow for remote management of the M2M applications installed in Kura including their deployment, upgrade and configuration management. The Remote Management service relies on the Configuration Service and the Cloud Service.
- Networking

- Provide API for introspects and configure the network interfaces available in the gateway like Ethernet, Wifi, and Cellular modems.

- Watchdog Service
    - Register critical components to the Watchdog Service, which forces a system reset through the hardware watchdog when a problem is detected.
- Web administration interface
    - Offer a web-based management console running within the Kura container to manage the gateway.

3.2 Cloud service abstraction

The cloud platform is a machine to machine integration platform that simplifies device and data management by connecting distributed devices over secure and reliable cloud services. It is an end-to-end platform that provides an easy path to connect cloud-ready devices to IT systems and/or applications. Once devices are deployed, the cloud platform allows users to connect, configure and manage devices through the lifecycle, from deployment, through maintenance, to retirement. In the context of the Automotive use case (UC2), the devices are the embedded systems installed in the car and the data are the functional and extra functional properties monitored by these devices.

The cloud service abstraction is responsible to provide full control over the embedded systems hardware, software and acquired data with a simple service model. The objective is to completely hide the complex details that stand behind the remote management procedures, remote data acquisition and transmission.

3.2.1 The cloud platform architecture

The functional architecture of the cloud platform is described in the following figure:

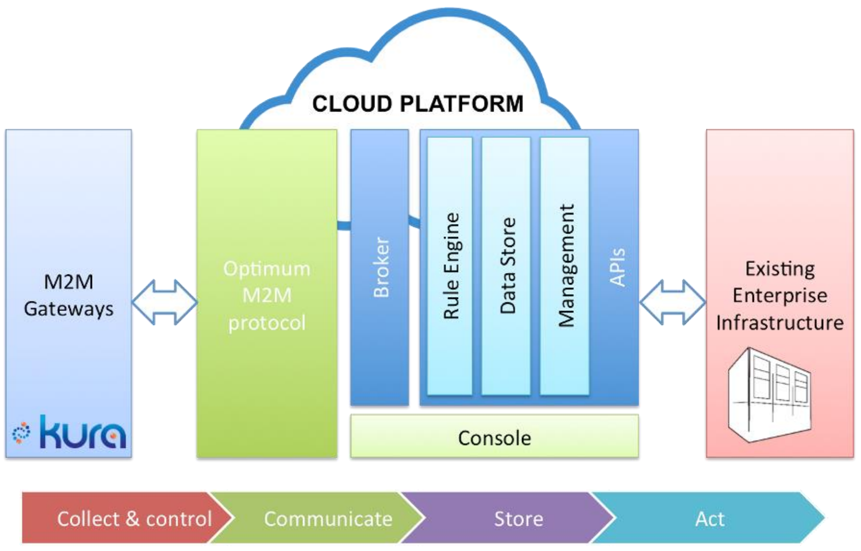


Figure 17: The architecture of the cloud platform.

The M2M protocol is a fundamental element for the cloud infrastructure and the entire device to cloud system. The protocol represents the “glue” that keeps together the distributed elements (sensors, embedded devices/gateways and cloud) that form the pervasive cloud system. For this reason, the cloud platform supports natively a publish-subscribe message brokering protocol known as MQTT. MQTT is an open standard transport layer, which uses a hierarchical topic namespace and a flexible data payload. MQTT is bandwidth efficient and simple to implement, providing both security and reliability of transport and allowing the decoupling between data producers and application consumers, in a one-to-many message distribution. Considering the historical importance of this telemetry protocol for the industrial world, the platform includes MQTT as a standard data communication feature.

The core technology of cloud platform is the message broker. The broker runs in the cloud and provides an MQTT server into which devices publish their data. MQTT clients may also issue subscriptions for certain topic namespace definitions. When a data payload is published into the cloud on a certain topic, the payload is redistributed to connected clients subscribed to that topic. The subscribe/notify model is the basic concept on which the broker, and indirectly the entire cloud infrastructure, is built.

The Rule engine is one of the core components of the cloud platform. The engine is based on SQL and its role is to process incoming published data, allowing immediate message processing and/or notifications to occur based on user-defined criteria. Statistical rules are applied over the data in real-time. The actions generated by a rule can include e-mail, SMS, or Twitter notification, a field protocol publish event, or a REST API call.

The cloud platform utilizes a non-SQL non-relational database for device data storage. The data is redundant and replicated across geographical regions for maximum reliability. Device/user and data plan details are stored in a separate replicated, redundant SQL database. This solution satisfies the typical requirements of an application that has to manage huge amounts of data (Big Data) but, at the same time, has to integrate with the existing enterprise infrastructure.

The management component of the cloud infrastructure has been introduced to provide a rich set of features for data management, cloud infrastructure management and application development. This component includes: a security layer, a device management, a data management unit, an account management layer, a unit for device configuration administration and a layer for application integration support.

REST APIs provide easy, programmatic access to request or store data from the storage databases and to all the other features of the Cloud platform. The APIs allow you to incorporate the data into another system or to develop custom applications within the existing enterprise infrastructure.

The console provides a graphical user interface that shows all aspects of the cloud account’s operation. The administrator account has the responsibility to manage users, devices, data, data plans, and rules definitions through the console. Standard users can be given permissions to view data, as needed.

The cloud platform is designed to provide a “zero configuration” communication architecture, allowing the developer/administrator to focus on developing business application components, without having to worry about device-level programming or complicated data networks. It is a powerful software framework, leveraging the power and redundancy of cloud computing, open-standard protocols (MQTT, TCP/IP, REST), and strong security.

3.2.2 The cloud service abstraction

The cloud service abstraction is the second level of abstraction introduced by the “device to cloud” approach. This abstraction layer introduces a set of services that simplify, from the cloud, the data collection, device monitoring, data management, account management, etc.. The service abstraction cooperates with Kura to completely hide the complexity of hardware, communications and cloud infrastructure.

3.2.2.1 The cloud service data model

The cloud platform data model allows to describe functional and extra-functional properties in the cloud. The data model builds on the MQTT protocol by adding an open structure to the topic namespace and the message payload. This addition enables real-time analysis of the data stream published by the devices (via Kura) and establishes a control channel to remotely manage the devices with a command and control approach.

The platform introduces two types of topics:

- publish topic,
- control topic.

A publishing topic is used by a device (or application) to publish data into the cloud platform. On the contrary, a control topic is used by the application or the platform to send data to a device, such as for real-time control of physical control outputs.

A generic publishing topic can be represented as follows:

```
cloudAccountName/assetId/semanticTopic
```

where:

`cloudAccountName` is the name of the cloud account and is used as a unique identifier that represents a group of devices and users.

`assetId` is a unique identifier within an account that represents a particular asset. It is a good practice to associate the assetId to a single gateway device and to map it to the Client Identifier (Client ID) as defined in the MQTT specifications. For a gateway, the MAC address of its primary network interface is generally used as the Client ID and Asset ID of that gateway.

`semanticTopic` is the section of the topic used to further specify information about the application or sensors from which the data has been gathered, using a hierarchical namespace representation. The `semanticTopic` should be further expanded to provide further levels of topic space that can be subscribed to and queried based on data content. In example it is very useful to identify an application running on the gateway device with an `app_id`. Also the management of resources, like sensors, actuators, local files, or configuration options, can be simplified by the use of a `resource_id`. Applications manage resources by being able to list them, read the latest value, or update them with a new value. Each resource could be identified by a `resource_id` as part of the hierarchical topic.

With this approach, a gateway identified by `assetId` (client ID) and belonging to `accountName` can have one of more applications – “app_id1” and “app_id2” – running. Each application can manage one or more resources identified by distinct `resource_id`(s).

The control topics can be classified in two categories:

- topics that represents a control channel to the specific device,
- topics that represents a control channel to all the devices belonging to an account.

These two topics are defined respectively as follows:

```
$ctrl/accountName/assetId/semanticTopic
```

or

```
$ctrl/accountName/+/semanticTopic
```

where:

`$ctrl` is the default part of the control topic and cannot be changed or altered. This prefix distinguishes control topics from data topics used in unsolicited reports, and it marks the associated control messages as transient, not stored in the historical data archive if one is present.

`accountName` is the name of the account owner.

`assetId` is a unique ID representing a particular asset (either the application or the sensors from which the data has been gathered).

`semanticTopic` is the section of the topic used to further specify information about the device or data, using a hierarchical name space representation.

`+` is an operator that aggregates all the asset IDs belonging to a specific account (essentially a wildcard character used as a middle element of the topic namespace).

An additional control topic may be used in Rules. It is defined as follows:

```
$accountName/rulesAssistant/semanticTopic
```

where:

`$accountName` is the name of the account owner.

`rulesAssistant` is specific to the cloud rules processing

`semanticTopic` is the section of the topic used to further specify information about the device or data, using a hierarchical name space representation.

When creating a new account in the cloud platform, the account holder is given publish and subscribe rights to the Publish and Control topics for that account. By default, when establishing a new cloud connection, the Cloud Client for Java subscribes to the aforementioned control topics listing for messages that may be addressed to that device.

3.2.2.2 The cloud core service layer

This section illustrates the cloud service abstraction layer, as far as concerned the data acquisition services, data publishing services and control services. These three aspects of the abstraction layer are strictly related to the management of the functional and extra-functional properties and, from the cloud perspective, represent the basic building blocks for the implementation of the automotive use case (UC2). These services represent the core of the cloud service abstraction.

The cloud abstraction layer is based on a publish-subscribe-notify paradigm that allows the separation of data producers and data consumers and the creation of one-to-many message distribution.

The cloud service abstraction layer introduces three types of services:

- data acquisition service,
- data publishing service,
- and control service.

Data acquisition services are intended for data acquisition from the field. Each functional and extra functional property must be described following the data model, illustrated in the previous section, and a specific data acquisition service must be defined and implemented. Focusing on a single property, the role of the data acquisition service is to send the data to the publishing service every time the data changes.

The data acquisition service owns the connection to the remote device (via Kura) and has full control on the connection parameters. The service adopts the field protocol available in the cloud platform (MQTT) to communicate with the remote device. Its design include the description of the field protocol both physical - serial vs I2C vs etc.- and logical - payload format used, frequency of information, etc.. Also the physical connection parameters - serial port, baud rate, or I2C address – can be configured.

Data publishing service is responsible for the information stored at cloud level. This service receives data from the data acquisition services and publish them. Once the data publishing service receive data referring to a specific topic, it filters, aggregates and publishes the received data, updating the corresponding topic. Also for this service a large set of parameters can be configured: triggering thresholds, publishing parameters, topics namespace, QoS, etc.. Publishing service design includes the topics namespace and the payload format used (metrics and payloads).

Finally, the control service is responsible for the actuation of commands (retroactions) on the cloud platform itself and on the remote devices (via Kura). In synthesis, this service receives generic command from the cloud and it dispatches them to a specific control executor that, in turns, is in charge to execute the command. Following the subscribe-notify paradigm, this service subscribes to a certain set of command (control topics) on the cloud. When it is notified about one of these commands, it converts the received control topic in a proper command and sends it to an executor. The control executor, in turns, is a software capable to manage and execute this specific command on the hardware of the remote device (i.e. via Kura the executor is capable to convert a command in a set of electrical signals for the I/O of the remote board). Control executors are bundles that run in the OSGi framework. As for the other core cloud services, the control service can be configured regarding the enabled commands, the default arguments, the control executor, etc..

The cloud platform adopted in the automotive scenario (UC2) has been implemented following this design pattern. The following figure illustrates a possible implementation for extra-functional properties like the power consumption and the thermal profile of the remote device. Every further domain service follows the same design pattern.

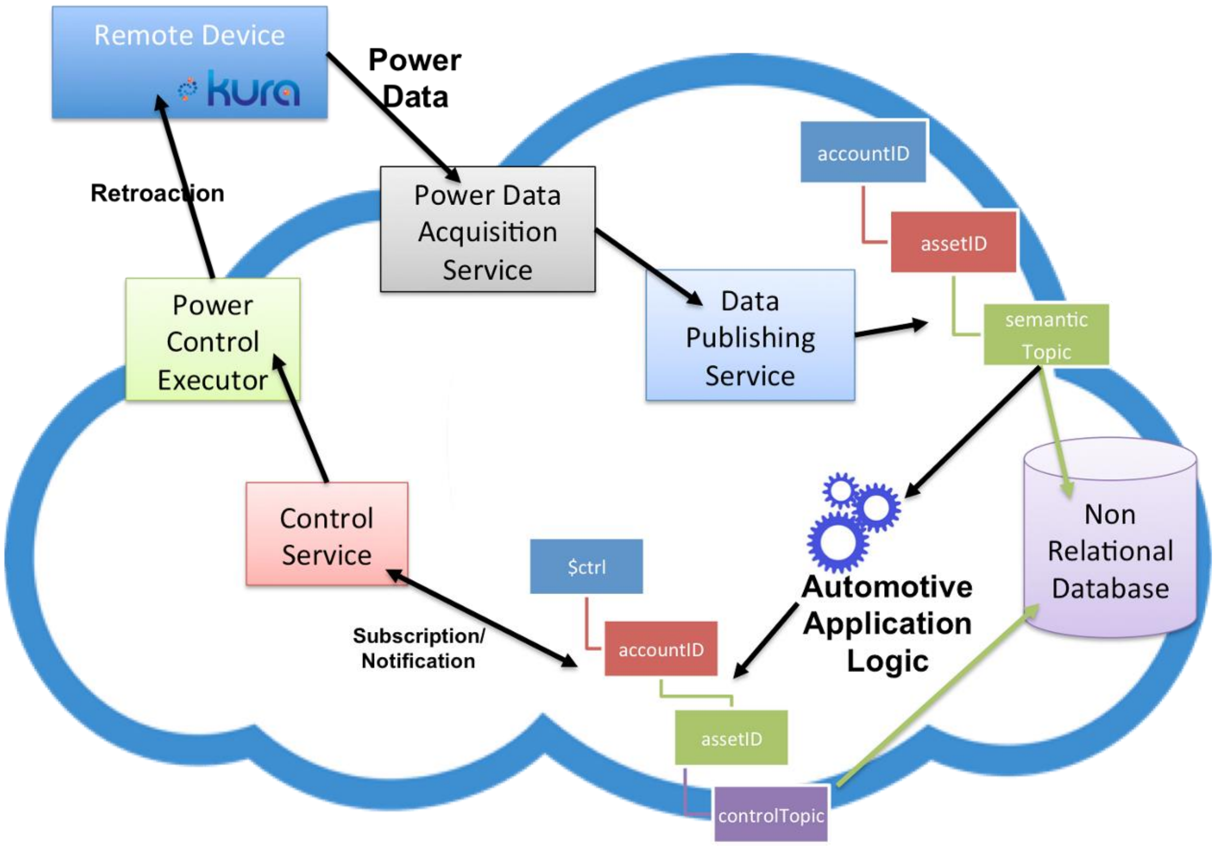


Figure 18: An example of the cloud service abstraction.

The power and thermal functional properties are described in the cloud with a set of specific topics, which represent the reference to the power and thermal information stored in the cloud.

Two specific data acquisition services are introduced, in order to collect the data from the remote devices installed in the car: the power and thermal data acquisition services. Every time these services receive a power or thermal data, they send it to the data publishing service that, according with the application logic, elaborate it, aggregate it and finally publishes the data to the corresponding publishing topic.

A set of control executors has been developed, in order to perform the retroaction activities required by the logic of the automotive application. Every control executor subscribes to specific control topics and, when the cloud notifies it, it executes (via Kura) the code required to manage that particular event. In example, if the thermal status of the remote device is critical, the executor responsible to manage this situation can activate and reduce the frequency of the CPU, shut down certain on-board peripherals, etc..

4 Services definition and characterization

4.1 Open Virtual Platform used in use-cases 1 and 3

Open Virtual Platform™ (OVP™) [6][5] by Imperas™ provides a complete simulation infrastructure for virtual platforms of embedded systems. OVP™ consists of three main components: OVP APIs to build own models and to create extensions for the simulator, a library that contains many open source processor and peripheral models that are free available and OVPsim™ which is the simulation kernel to execute these models. The partners of use-cases 1 and 3 use OVP™ to create and simulate virtual platforms of their execution platforms. Goal is to extend OVP™ by extra-functional models to be able to get timing and power information while simulating the execution platforms. Since OVP™ is only a functional simulator it is able to simulate the virtual platforms very fast, but has no accurate timing models for the provided models of processing elements. In that way, also no power models are available for them. However, with the offered APIs it is possible to get basic information about the simulated platforms to implement such models during simulation. Especially the *Innovative Cpu Manager Interface* (ICM) API gives access to the internal state of the simulated virtual platform. As a first step, OFFIS developed a timing model for the model of the Xilinx MicroBlaze, which is available in the OVP™ library, which uses the ICM API. To create an accurate timing model the mainly needed information needed are the executed instructions of the particular processing element. For this, the ICM API offers methods to register callback functions for specific events or memory address areas. One of these callbacks is the fetch callback.

After registering this callback, the committed function pointer “writeCB” is called for every

```
void icmAddFetchCallback(  icmProcessorP processor
                         , Addr lowAddr
                         , Addr highAddr
                         , icmMemWatchFn writeCB
                         , void *userData);
```

fetched instruction. Next to the callback function, a pointer to the specific processing element is needed as well as the low and high addresses of the instruction memory. Since the callback function needs to be static, the user is not able to access any non-static elements of its class. To overcome this issue the “icmAddFetchCallback” function has the parameter “userData” to commit any kind of data or class pointer (e.g. “this”). To get the pointer of the needed processing element it can be searched by its plaintext name by another ICM API function.

```
icmProcessorP icmFindProcessorByName(const char *name);

Example:
icmProcessorP cpu = icmFindProcessorByName(“top.cpu1”);
```

The function is called with a hierarchical name as a string with ‘.’ as separator and returns a pointer to the named processing element. In the example, the whole platform is named “top” and the contained processing element “cpu1”. The signature of the callback function is defined by the type “icmMemWatchFn” and is like follows.

```
static void fetchCallback(icmProcessorP processor
                        , Addr address
                        , Uns32 bytes
                        , const void *value
                        , void *userData
                        , Addr VA);
```

By using the address of the virtual instruction memory in the host machine and adding the given address offset the fetched operation code of the current instruction can be accessed. This operation code is given as parameter to the MicroBlaze timing model that returns the present system time in processed cycles. For getting the time in seconds, it has to be divided by the configured processor's frequency. For updating the simulation time, the ICM API has a function as well.

```
Bool icmAdvanceTime(icmTime time);

(With: typedef long double icmTime; // Current simulator time
                                          in seconds)
```

The simulation time is updated continuously by giving the computed time of the timing model to this function in seconds as a double value after each instruction fetch and timing model call. Because of that, OVP™ gets a quasi-cycle accurate behaviour. More information about the timing model is provided in [5].

Next to the used functions for the timing model, the ICM API provides more helpful ones to analyse the current state of the virtual platform. With the following function, it is also possible to get the current simulation time of the virtual platform.

```
icmTime icmGetCurrentTime(void);

(With: typedef long double icmTime; // Current simulator time
                                          in seconds)
```

As well as the current counter for executed instructions of each processor model:

```
uint64_t icmGetProcessorICount(icmProcessorP processor);
```

For accessing the processor's registers to get further information about the present state the ICM API has special read and write functions.

```
Bool icmReadReg(icmProcessorP processor
              , const char *name
              , void *buffer);

Bool icmWriteReg(icmProcessorP processor
               , const char *name
               , const void *buffer);
```

In addition, these functions need a pointer to the processing element the register belongs too. The register names can be found in the documentations of the particular virtual processor. The

```
Addr icmGetPC(icmProcessorP processor);

void icmSetPC(icmProcessorP processor
          , Addr simAddr);
```

character buffers are used for getting and setting the register value. Further functions for setting and getting the present program counter of the processing element are available.

If the next instruction is needed the processor pointer and the present program counter can be used to get its address as return value of the following function.

```
Addr icmGetNextInstructionAddress(icmProcessorP processor
                              , Addr thisPC);
```

Like the mentioned fetch callback also callbacks for read and write accesses of memory areas and busses are available.

```
void icmAddReadCallback(icmProcessorP processor
                    , Addr lowAddr
                    , Addr highAddr
                    , icmMemWatchFn writeCB
                    , void *userData);

void icmAddWriteReadCallback(icmProcessorP processor
                         , Addr lowAddr
                         , Addr highAddr
                         , icmMemWatchFn writeCB
                         , void *userData);

void icmAddBusReadCallback(icmBusP bus
                         , icmProcessorP scope
                         , Addr lowAddr
                         , Addr highAddr
                         , icmMemWatchFn writeCB
                         , void *userData);

void icmAddBusWriteCallback(icmBusP bus
                         , icmProcessorP scope
                         , Addr lowAddr
                         , Addr highAddr
                         , icmMemWatchFn writeCB
                         , void *userData);
```

The usage of these callbacks is in basic the same as of the fetch callback. The observed address range has to be determined and the callback function has to be announced and implemented. In additional the bus callbacks need a reference (as pointer) to the observed bus.

4.2 Extension of OVP scenario with SystemC models

The OVP virtual platform is useful until standard components are used or new components (e.g., peripherals) are implemented by using OVP native API. Unfortunately, most of behavioural models found in the industrial context are written in SystemC and their translation could be time-consuming and error-prone thus limiting the adoption of OVP. OVP provides a mechanism to wrap the CPU model and OVP peripherals into a SystemC design. However, in this project we investigated also another approach.

In this project, the HW/SW co-simulation architecture depicted in Figure 19 is proposed by EDALab [7]. Similarly to the approaches proposed in [8][9], it reflects the traditional operating system based stack where software applications interact with hardware peripherals through device drivers. The target platform, where software applications run, is emulated by using OVP and it is connected to a SystemC hardware peripheral through a virtual device, a SystemC bridge and a device driver for the target operating system. The virtual device, connected to the bus of the virtual platform, acts as an interface between the SystemC simulator and the OVP virtual platforms. The virtual device code depends on the APIs exported by OVP for modelling new devices. The SystemC bridge consists of a set of functions that allow the communication with SystemC. The bridge is compiled as a C library linked to the implementation code of each virtual device, thus it is independent on the selected virtual platform (e.g., it can be the same for both a QEMU-based and an OVP-based architecture). Finally, a device driver must be developed for the target operating system to use the hardware peripheral. Its code is clearly independent on the selected virtual platform and it does not need to be changed when it is moved to the actual platform. This approach allows a rapid interchange from an OVP-based SystemC co-simulation and vice versa, since only the virtual device must be re-coded moving from an OVP virtual platform. Further details about the virtual device and the SystemC bridge are reported in the following. The implementation of the device driver is not reported since it does not depend on the proposed co-simulation architecture, but only on the operating system.

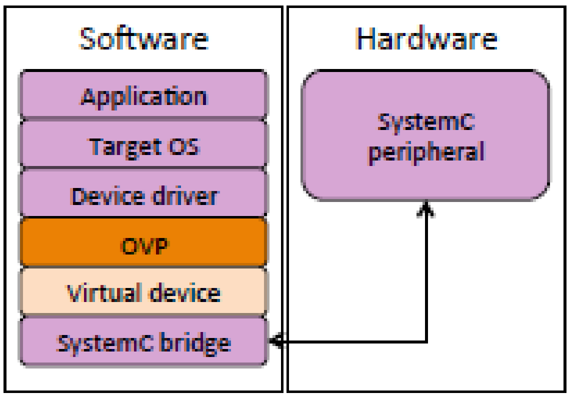


Figure 19 Co-simulation architecture.

The core of SystemC bridge is a C++ class implementing a singleton design pattern that exposes a set of APIs towards the virtual device for interfacing with SystemC. Moreover, it manages the communication protocol and the synchronization mechanism between OVP and SystemC. Since OVP is written in C, the bridge wraps the SystemC APIs through a set of C functions included in a library which is statically linked to the SystemC runtime library. The OVP virtual platform initially calls a function implemented in the bridge to start the SystemC simulator. Until the SystemC runtime is operative, the virtual platform is blocked to prevent its premature requests to the hardware device. Differently from [8], the SystemC simulator is executed as a separate thread inside the same process where the OVP simulator is executed (Figure 20), such that the communication between the two worlds is based on shared memory and thread synchronization primitives. This prevents the use of expensive interprocess communication mechanisms (like sockets). Then, the starting routine of the thread launches the SystemC sc_main() function where the following steps are executed:

- instantiation and initialization of the SystemC device to be connected to the bridge; in particular, input and output ports of the device are registered in the bridge;
- unlocking of the semaphore that is blocking the virtual platform;
- starting of the SystemC simulator.

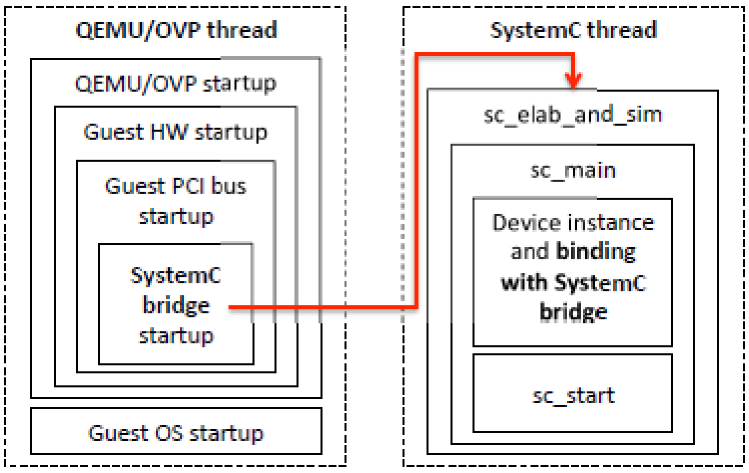


Figure 20 Start-up of the co-simulation between OVP and SystemC.

A device can be made visible to the OVP virtual platforms by implementing a virtual device. Its implementation depends on the kind of adopted bus. The creation of a virtual device is tightly coupled to the API provided by the virtual platform. The implementation of the virtual device in OVP is split in two parts: the device and the intercepted functions. The device consists of a C application with a standard main function. It first executes a set of initialization activities (SystemC simulation, bus configuration header, bus memory regions); then it connects the bus configuration port (necessary to read the bus configuration header) and the master bus. Finally, it registers some call back functions that are triggered at each read/write operation from/to the bus I/O port regions. The interaction between the SystemC bridge and the virtual device is performed by means of a set of intercepted functions. The Peripheral Simulation Engine intercepts such functions through the Application Binary Interface (ABI), which specifies size, layout and alignment of data types, how an application should make a system call to the operating system, and the calling conventions (how arguments of a function are passed, and how the return value is retrieved). In particular, there is an intercepted function for reading from the SystemC device and one for writing to the SystemC device. Their role consists in calling the corresponding sc_ioport_read() and sc_ioport_write() of the SystemC bridge.

4.3 Use-case 1: Unmanned Aerial Vehicle

Services in the Unmanned Aerial Vehicle use-case are provided by the Xilinx Zynq platform hardware components, its Board Support Package (BSP) or Hardware Abstraction Layer (HAL), its Operating System Layer, the corresponding virtual platform for simulation, and tools for trace processing to extract and control properties. In CONTREX, these services are used for

1. Mapping and implementation of the Mixed-Critical applications on the provided target platform (Xilinx Zynq)
2. Introspection and reference for extra-functional traces (e.g. power and temperature over time)
3. Control of extra-functional properties (e.g. adaptive QoS, power and temperature management)

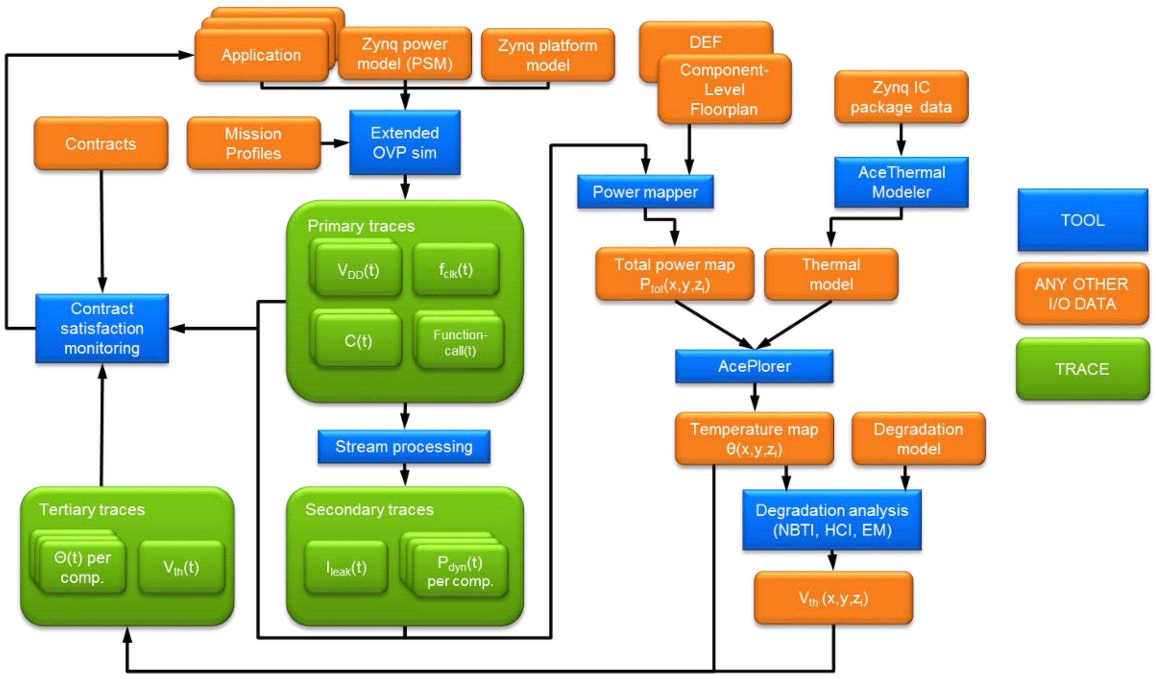


Figure 21: Overall CONTREX power, temperature and degradation analysis flow.

In use-case 1 we want to analyse the extra-functional properties power, timing and temperature of a safety-critical flight control algorithm and a mission-critical payload processing (i.e. high performance image processing) running together on the Xilinx Zynq platform, as described in s section 2.3. As described in D3.1.1, the overall CONTREX power, temperature and degradation analysis flow, as shown in Figure 21, enables tracing extra-functional properties like supply voltage, clock frequency or switched capacity per hardware platform component (processor, memory, bus, peripheral) over time. For establishing a cause and effect relationship, identification of the Application, Task, Function state and its associated context over time need to be traced and associated with the observed extra-functional properties.

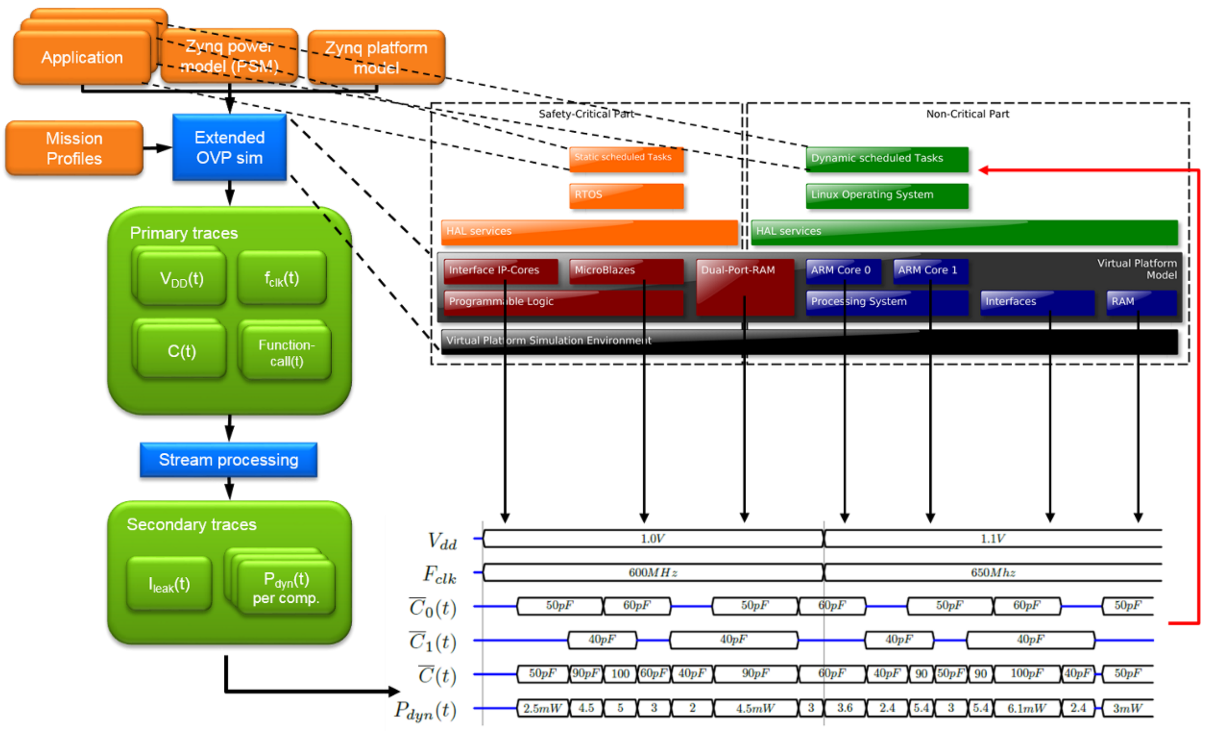


Figure 22: Establishing a link and traceability between applications and platform hardware component's extra-functional traces through services.

Figure 22 shows the goal of establishing a link and traceability between applications and platform hardware component's extra functional traces. For enabling a correlation of application activity and status of the platform components with the power over time trace per component, a traceable service model is proposed.

In a first instance, we analysed extra-functional properties on a task level granularity. This can be refined to a function level, if required.

In our service model, an application is represented by a set of periodic and sporadic tasks. Tasks can communicate and synchronize through FIFO and handshake communication channels, called Shared Objects as generalization in the following. We define a finite set of Applications A_i with

- criticality level Li
- set of Tasks Ti
- set of Shared Objects Si

Each Task $t_j$in T_i is defined by (P_j, D_j, SI_j, L_j) with

- period (minimum arrival time) P if periodic task and P =
- deadline D
- ports to Shared Object Interfaces SI in Si.I
- criticality level L

Each Shared Object S_i consists of

- a set of Interfaces Ii with methods mj in ik in Ii(let Mi be the union of all methods in Ii)
- a set of side effect free Guards Gi
- a set of guarded methods GMi in Mi x Gi
- implementing all interfaces methods Mi
- a shared resource access arbitration policy

As shown in Figure 23, each task can be in the state inactive (not initially assigned to a processing element), ready (waiting to be executed on a statically or dynamically assigned processing element), running (currently executing on its assigned processing element), waiting (preempted by the task scheduler), blocked (waiting for shared resource access), and resource access (accessing a shared resource). When no application or operating system task is running, an idle task is running on each processing element.

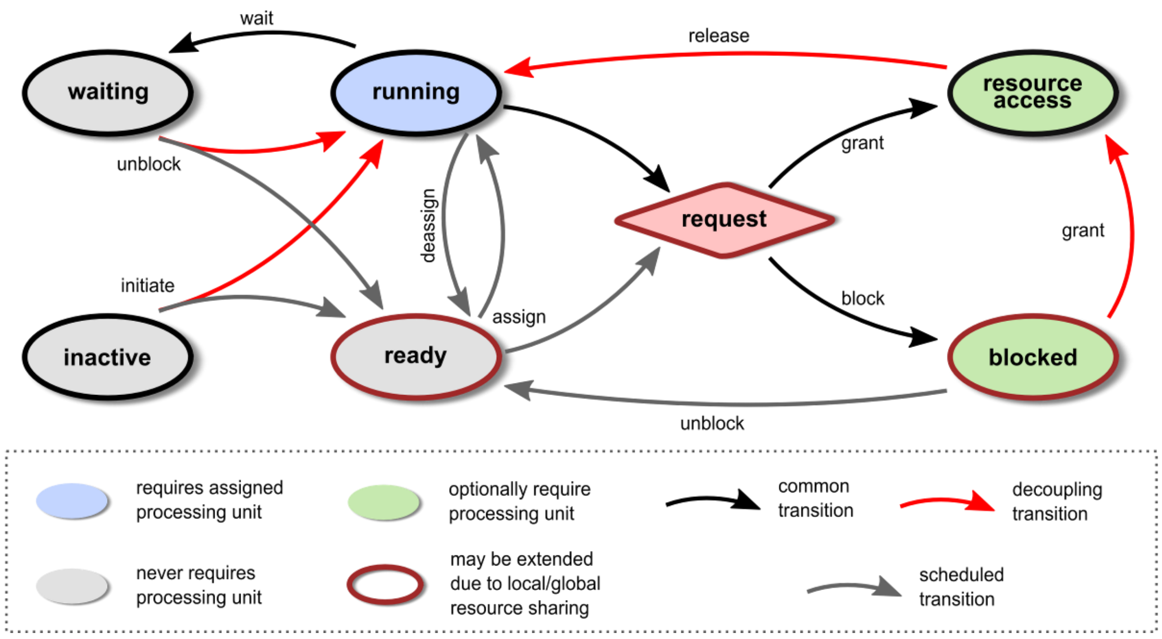


Figure 23: Example of a task state machine.

In our use-case, a task consists of at least one function and/or communication of a task with other tasks or the environment of the system. To align and combine the extra-functional properties per processing unit over time in line with tasks we needed to represent state of each task together with the system time. The overall system time has been provided by the virtual platform simulator's reference time while the current state of the tasks has been represented by an event-system denoting the transition or transition events of the described task state machine.

Figure 24 and Figure 25 show the structure of the different layers of the real hardware prototype system and the virtual platform of the Xilinx Zynq. All service layers above the hardware layer are similar (these are the HAL-, OS-/Scheduler- and Task-Layers). The main differences are in the base layers. In the real hardware prototype system we have the actual Xilinx Zynq MPSoC with the Hardware processing elements, memories, communication elements and peripherals either implemented in the ASIC or the FPGA partition. In the virtual platform model, these layers are represented by corresponding OVP simulation models.

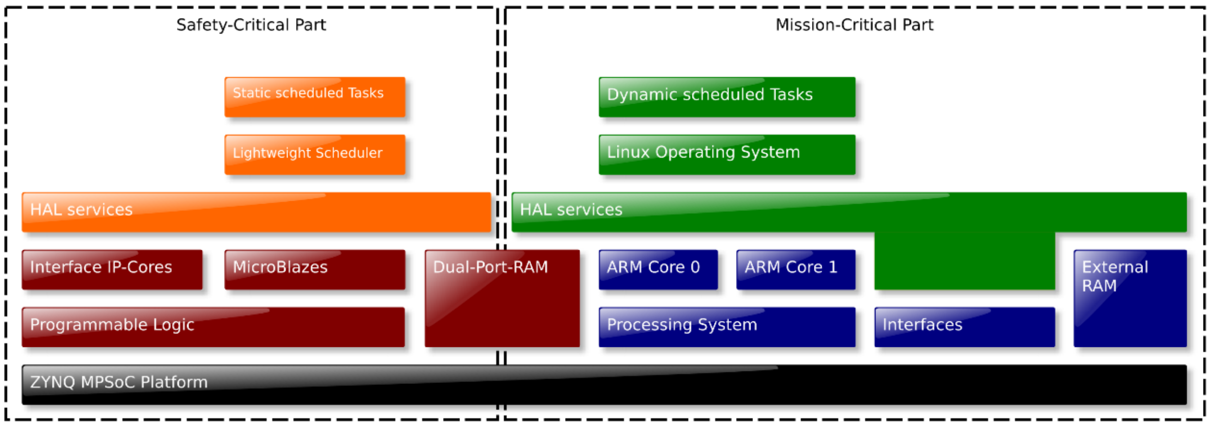


Figure 24: Overview of Layers in real system of use-case 1.

Both, the hardware and the virtual platform can be divided into a safety-critical (left side in Figure 24 and Figure 25) and a mission-critical part (right side in Figure 24 and Figure 25). While the safety-critical part computes the flight algorithms, the mission-critical part performs the payload processing, e.g. video processing. As shown in the figures the safety-critical part uses the programmable logic of the MPSoC and the mission-critical part uses the ARM processing system. While we analysed the whole MPSoC we needed the availability of providing services in both parts of the system.

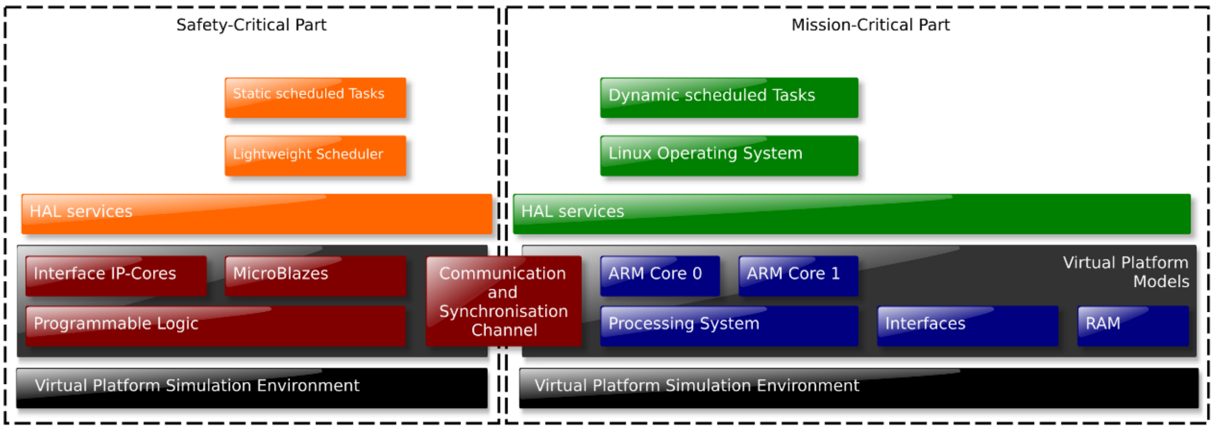


Figure 25: Overview of layers in virtual platform of use-case 1.

In our first approach for analysing the extra-functional properties on task level we need for each task: start time, end time and also the times of disruption through other tasks or communication and shared resource accesses. Like mentioned before, timers of the processing elements have be used via the HAL services to get the present system time. For getting the present state of a task, a task state machine can be used like shown in Figure 23. The emitted events of the state-machine together with the time can be used to synchronize the tasks and the measurements of the extra-functional properties of the real system. As follows these services are located in the top layer of Figure 24 and Figure 25, where the tasks are located.

After a characterization of the real platform, the extracted information about the power-states of the hardware can be used together with the event emitting system in the virtual platform to get a simulation of the extra-functional properties.

To simulate extra-functional properties assertions are extracted from traces by using a model abstraction and mining strategies. Assertion definition is generally a tedious and error-prone manual activity, which requires high expertise and long verification time. Too many assertions can cause too time expensive verification sessions, while too few assertions can lead inadequate coverage of design under verification (DUV) behaviours. In order to make easier the definition and refinement of assertions, automatic specification mining has been proposed, whose goal is to extract significant assertions from the actual implemented design.

Specification mining approaches can be classified as static [Ammons2002] or dynamic [Bonato2012]. Static specification mining infers assertions by formally exploring design models or implementations. It is effective and accurate, but it does not scale sufficiently well with the design size. Dynamic specification mining is intended to infer assertions from the execution traces of the DUV. It can guarantee a better scalability, but the quality of mined assertions depends on the quality of the test suite used to simulate the design.

Despite of static or dynamic approaches, mined assertions are, generally, not intended as an alternative for their (manual) definition; their primary goal is to improve the verification and documentation process by helping verification engineers with a way for evaluating and extending the basic manually-defined set of assertions.

While specification mining techniques are gaining popularity for verification of the DUV functional behaviours there is a lack of studies concerning their application for extra-functional properties. However, due to the ever increasing importance of monitoring and evaluating extra functional properties, it seems a promising research direction to extend specification mining approaches to the analysis of timing, power, thermal and reliability characteristics of a DUV.

In the context of the CONTREX project, techniques for automatic generation of assertions for extra-functional properties has be defined and integrated with the traditional contract-based design approach targeting functional behaviours.

The proposed approach is reported in Figure 26. The first issue that we are going to face is a coding problem. To properly integrate the functional and the extra functional aspects of a design, a common formal language, i.e., the Property Specification Language (PSL), has be used. PSL assertions mixing functional and extra-functional propositions allowed us to describe how the DUV changes its functionality/state according with the not-functional aspects of the entire system such as power and temperature. Then, specification mining approaches have been integrated with formalisms targeting extraction of non-functional aspects like for example Power State Machines (PSMs) leading to the definition of a mixed functional/extra-functional miner.

If necessary, the granularity of the service abstraction level can be further refined, so that the extra-functional properties can be linked to the function calls and the communication.

While a Linux operating system has been used at the ARM processing system (executing the mission and non-critical applications) the runtime manager BBQ, developed in Task 3.4 and presented in Deliverable D3.3.1 can be used with its provided services.

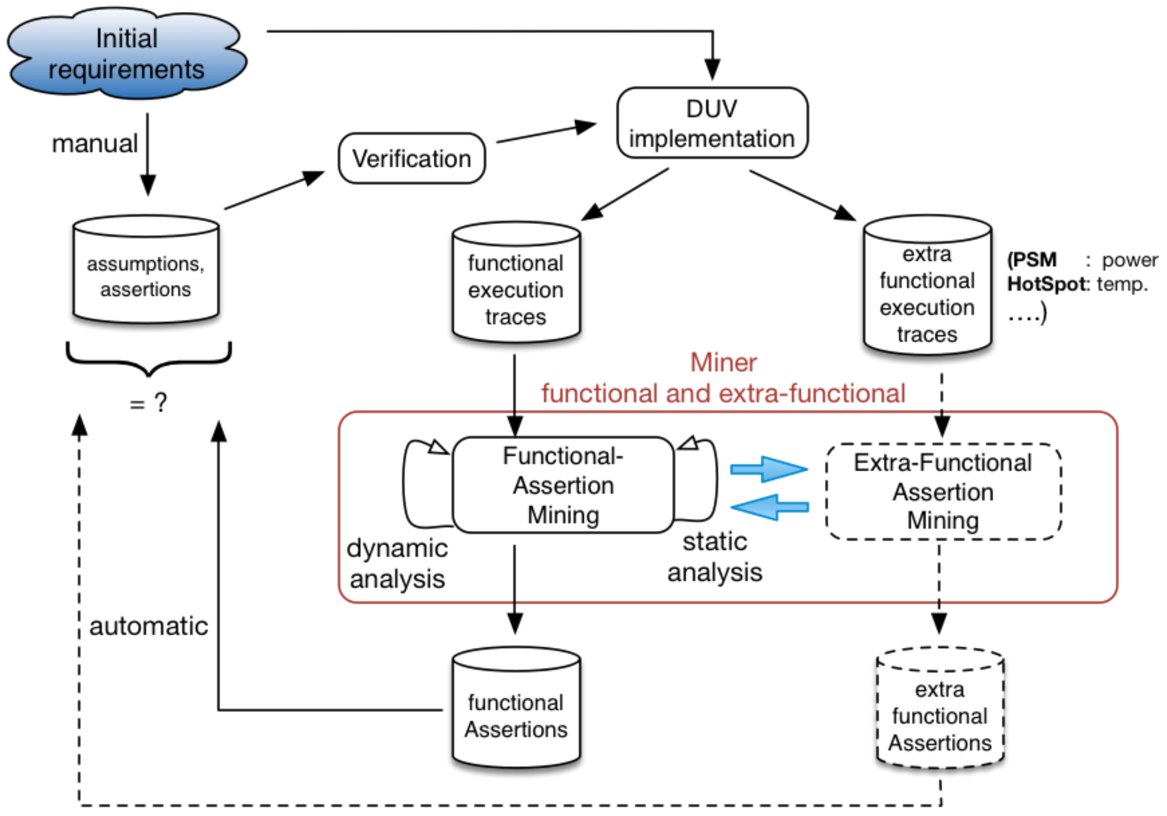

Initial requirements
manual
assumptions, assertions
= ?
Verification
DUV implementation
functional execution traces
extra functional execution traces
(PSM : power
HotSpot: temp.
….)
Miner
functional and extra-functional
Functional-Assertion Mining
Extra-Functional Assertion Mining
dynamic analysis
static analysis
automatic
functional Assertions
extra functional Assertions

Figure 26: Overview of the proposed approach for extra-functional assertion mining.

4.4 Use-case 2: Automotive Telematics

4.4.1 iNEMO node level application services

iNEMO-M1 SOB is provided together with an evaluation board, known as iNEMO-M1 discovery board and an SDK (Software Development Kit) that supports user application development and facilitate system parameter configuration and operation by connecting it to a PC (see Figure 27).

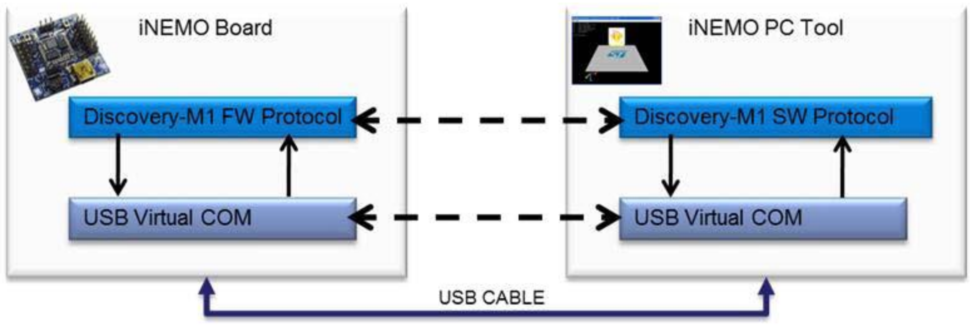


Figure 27: iNEMO-M1 board and PC GUI interaction

4.4.1.1 iNEMO-M1 SDK as node-level service provider

iNEMO-M1 discovery system is a Real Time system that allows getting data from its embedded sensors (accelerometer, magnetometer, gyroscope, pressure) according to the configured time limits and data rates. Simple data sampling and two estimation algorithms are provided too. Data sampling mainly consists of a reading of the output registers of the sensors and, eventually, in their conversion into a measurement unit. Indeed, according to the user choice, data can be represented both in raw mode (values as they are represented into the sensor output registers) or using a measurement unit (*mg* for acceleration, *degree/sec* for angular speeds, *mgauss* for magnetic field and *mbar* for pressure). Moreover, it is possible to set offset and gains in order to make possible a calibration of the output according to the application needs. The two estimation algorithms mentioned above can be enabled or disabled by the user. They are:

- An **AHRS** (Attitude and heading reference system) algorithm. It uses a Kalman filter to estimate the attitude of the board from the measurements of the gyroscope, accelerometer and magnetometer. The output of this algorithm is the four quaternions representing the attitude and the corresponding roll, pitch and yaw angles expressed in degrees.
- A **tilted compass** algorithm. It is used to compute the heading angle (the angle between the X axis of the board system and the magnetic north on the horizontal plane measured in a clockwise direction) also when the board is tilted. This estimation makes use of the accelerometer and magnetometer outputs
- Moreover, a **HIC** (hard iron calibration) automatic procedure for magnetometer has been included essentially with the intent to improve the heading measurements. It tries to find the magnetometer gains and offsets minimizing an error while asking the user to explore all the possible directions.

There are mainly two ways to gather data:

- Using a timer of the microcontroller and get data once it raises an IRQ.
- Synchronize the sensor output reading to the *DATA READY* signal of one sensor.

By default, each time one of these events occurs, data are sampled and sent to the host. This operating mode ensures that every sampling operation corresponds to one sent message. However, an asynchronous mode for providing data that is called ASK_MODE is available. This one consists into the separation between the sampling (and eventually processing operation) and the sending one as such the user has to ask the previously acquired data (that are buffered) in order to obtain them. Using FreeRTOS, it is possible to obtain a logical partition of tasks directly mapped on the code.

Three FreeRTOS tasks are implemented here:

- A **command task** manages the command frames coming from the host. It decodes the command and executes it, configuring sensors, acquisition modes, algorithms to execute and so on.
- A **data process** task, woken up when an event occurs (a timer interrupt or a DATA READY IRQ). It processes the data from sensors and sends them to the host.
- A **gatekeeper** task that implements the communication front end. It manages the physical layer (USB, USART or wireless and so on) at low level and it is the only module that understands the details of the communications (here it is implemented as a

USB-VirtualCom manager). This approach gives flexibility to the entire system as such it can be used for different applications by changing only the implementation of this task still using the same processing core. The gatekeeper provides the input data into a FreeRTOS list on which the command tasks is stopped and sends the output data picking them from a list in which the other two tasks write.

The following figure shows the C modules used to drive specific parts of the system.

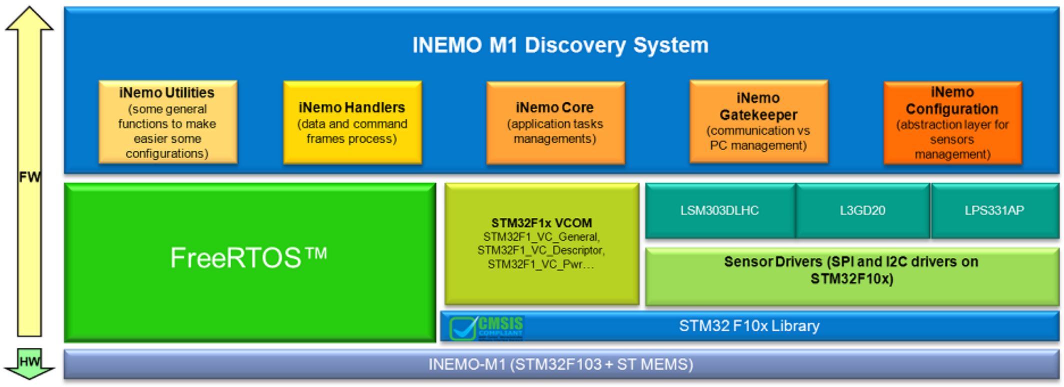

INEMO M1 Discovery System
iNemo Utilities (some general functions to make easier some configurations)
iNemo Handlers (data and command frames process)
iNemo Core (application tasks managements)
iNemo Gatekeeper (communication vs PC management)
iNemo Configuration (abstraction layer for sensors management)
FW
FreeRTOS™
STM32F1x VCOM STM32F1_VC_General, STM32F1_VC_Descriptor, STM32F1_VC_Pwr…
LSM303DLHC
L3GD20
LPS331AP
Sensor Drivers (SPI and I2C drivers on STM32F10x)
CMSIS COMPLIANT
STM32 F10x Library
HW
INEMO-M1 (STM32F103 + ST MEMS)

Figure 28: iNEMO-M1 Discovery system software architecture

4.4.1.2 iNEMO M1 Discovery System – Interface block description

The modules shown in the top level area of Figure 28 constitutes the interface of iNEMO system towards the host unit (a PC, an application system execution platform, etc.) and allow to process input commands and provide in output the sensor data through a proprietary communication protocol, described in the next section. Here follow the main functionalities provided by these modules.

iNemo Handlers:
This module includes all the features to manage the protocol frames. The frames are exchanged with the host through the gatekeeper which accesses the communication peripheral.

iNemo Core:
This is the main iNemo discovery system module. It contains the Command and Data task implementations.

iNemo Gatekeeper:
This module is responsible for the physical host communication. The gatekeeper is a task used to avoid race condition on the communication peripheral (USB, RS232, RF,...) since it is the only one that accesses it.

iNEMO Configuration:
This module is used to enable or disable sensors and other peripherals

4.4.1.3 iNEMO-M1 Firmware Main module

Once the iNEMO-M1 discovery board is powered and connected to a host unit via an USB cable, it is ready to receive commands for configuring it or to ask for data sensed by its embedded sensors.
At runtime, the program execution is managed inside the main.c module, where after initialization procedures, just two tasks are involved for implementing the application services towards the host unit requests.
The source code reported below summarizes the main parts and procedure calls contained in the main.c module:

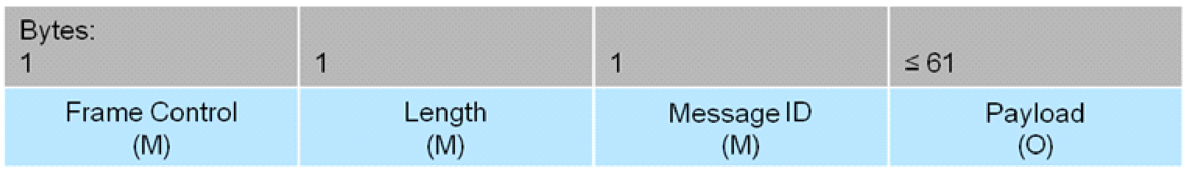
```
…

/* Real time scheduler includes */
#include "FreeRTOS.h"
#include "task.h"

/* STM32 Library includes */
#include "stm32f10x.h"
#include "stm32f10x_it.h"

/* Application includes */
#include "iNemo.h"

…

  /* iNemo Task init */
  iNemoInitTask(NULL);

…

/**
 * Main program. Sets up the hardware, init the tasks and starts the scheduler.
 * .
 */
int main( void )
{
  /* Configure all the hardware peripherials */
  prvSetupHardware();

  /* Start all iNemo Tasks. */
  iNemoStartTask(NULL);

  /* Start the scheduler. */
  vTaskStartScheduler(NULL);

/* Define here any additional user task */

UserStartTask(NULL);

}

…
```

4.4.1.4 iNEMO M1 Communication protocol

The communication between the board and the host happens in form of data frames complied with a protocol that encodes some kind of commands. They can be managed by the iNemo GUI running on a PC or, more in general, by any user Host Application.

This section describes the format of the frames used in the STEVAL-MKI121V1 communication protocol. Because, the STEVAL-MKI121V1 exchanges data and commands with the Host Application (or a PC GUI) through a physical communication channel based on a USB Virtual COM, each frame, described below, represents the payload of a USB frame.

The frame format is composed of a header and an optional payload. The general frame shall be formatted as illustrated in Figure. The header is composed of three mandatory (M) fields, each of which is 1 byte in length, while the payload is an optional field whose maximum length is 61 bytes.



Bytes: 1	1	1	≤ 61
Frame Control (M)	Length (M)	Message ID (M)	Payload (O)

Figure 29: Generic frame fields

There are two types of transactions: acknowledgment or non-acknowledgment of a DATA or CONTROL frame.

A DATA or CONTROL frame transmitted with the Ack subfield of its frame control field set to one shall be acknowledged by the recipient. If the intended recipient correctly receives the frame, it shall generate and send an ACK frame containing the same message ID from the DATA or CONTROL frame that is being acknowledged. It is possible also to include a payload in the ACK frame to transfer useful data from the recipient to the originator. The message sequence chart in Figure the shows scenario for transmitting a single DATA or CONTROL frame from an originator to a recipient with an acknowledgment and without it.

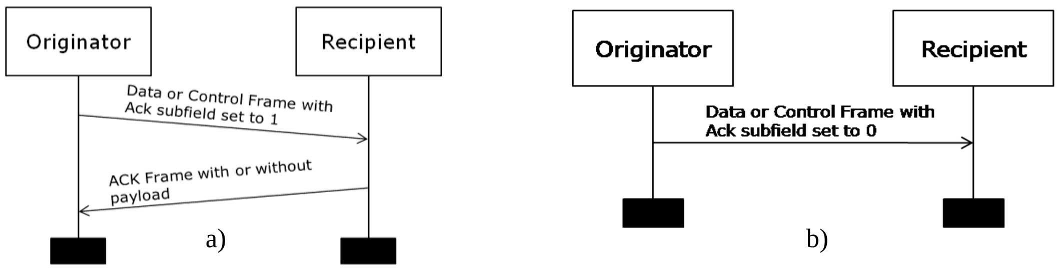

Originator
Recipient
Data or Control Frame with Ack subfield set to 1
ACK Frame with or without payload
a)
Originator
Recipient
Data or Control Frame with Ack subfield set to 0
b)

Figure 30: Message sequences with ACK (a) and without ACK frame (b)

The frames used in the STEVAL-MKI121V1 are classified in four types:

1. Communication control frames
2. Board information frames
3. Sensor setting frames
4. Acquisition sensor data frames

Communication control frames are originated by the Host unit (a PC SDK or a PC GUI for instance) and used to send specific commands to the Discovery-M1 board. All the communication control frames are listed in Table 1:

Commands	*Description*
iNEMO _Connect	It shall be the first command sent from Host unit. Any other command sent before the iNEMO_Connect is not processed by Discovery-M1
iNEMO _Disconnect	The iNEMO_Disconnect command closes the communication between the PC and theDiscovery-M1 board.
iNEMO_Reset_Board	The iNEMO_Reset command implies a software reset of the Discovery-M1 board.
iNEMO_Trace	The iNEMO_Trace command allows the user to enable or disable "trace data".

Table 1: List of communication control frames

Board information frames are originated by the Host unit and used to retrieve information about firmware and hardware features of the Discovery-M1, as described in Table 2:

Commands	*Description*
iNEMO_Get_MCU_ID	The iNEMO_Get_MCU_ID command allows retrieving from the Discovery-M1 board the 96-bit unique device identifier of the STM32F103 microcontroller.
iNEMO_Get_FW_Version	The iNEMO_Get_FW_Version command allows retrieving the board firmware version.
iNEMO_Get_HW_Version	The iNEMO_Get_HW_Version command allows retrieving the board hardware version. The iNEMO_Get_AHRS_Library command allows knowing the version of the Discovery-M1 firmware Attitude Heading Reference System (AHRS) algorithm.
iNEMO_Get_AHRS_Library	The iNEMO_Get_AHRS_Library command allows knowing the version of the Discovery-M1 firmware Attitude Heading Reference System (AHRS) algorithm.
iNEMO_Get_Libraries	The iNEMO_Get_Libraries command allows knowing which specific libraries are supportedby the Discovery-M1 firmware.
iNEMO_Get_Available_Sensors	The iNEMO_Get_Available_Sensors command allows knowing which specific sensors aresupported by the Discovery-M1 firmware.

Table 2: List of board information frames

Sensor setting frames are originated by the Host unit and used to set sensor parameters or to retrieve information about them. All the sensor setting frames are listed in Table 3:

Commands	*Description*
iNEMO_Set_Sensor_Parameter	The iNEMO_Set_Sensor_Parameter command allows setting a specific sensor parameter.
iNEMO_Get_Sensor_Parameter	The iNEMO_Get_Sensor_Parameter command allows retrieving from the Discovery-M1 aspecific sensor parameter.

iNEMO_Restore_Default_ Parameter	The iNEMO_Restore_Default_Parameter command allows restoring a default, specificsensor parameter.
iNEMO_Save_to_Flash	The iNEMO_Save_to_Flash command allows storing the settings of the sensor parametersin Discovery-M1 flash.
iNEMO_Load_from_Flash	The iNEMO_Load_from_Flash command allows loading from Discovery-M1 flash thesensors parameters stored in it.

Table 3: Sensor setting frames

Acquisition sensor data frames are frames originated by the Host unit to set how to retrieve sensor data from Discovery-M1. Sensor data frames can be also originated by Discovery-M1 to send acquired sensor data towards the host unit. Acquisition sensor data frames are listed in Table 4:

Commands	*Description*
iNEMO_Set_Output_Mode	The iNEMO_Set_Output_Mode command allows setting which sensors shall be enabled, inwhich format the data sensor shall be sent from Discovery-M1 to SDK, and other parameters.
iNEMO_Get_Output_Mode	The iNEMO_Get_Output_Mode command allows retrieving information from Discovery-M1 about its acquisition settings.
iNEMO_Start_Acquisition & iNEMO_Acquisition_Data	The iNEMO_Start_Acquisition command allows starting the acquisition of sensor dataaccording to the output settings. The acquired data are contained inside frame iNEMo_Acquisition_Data.
iNEMO_Stop_Acquisition	The iNEMO_Stop_Acquisition command stops the acquisition and data transmission
iNEMO_Get_Acq_Data	The iNEMO_Get_Acquired_Data command is used to send the acquired data.

Table 4 Acquisition sensor data frames

Here below is shown in more details all the fields constituting the frame ‘iNEMO_Acquisition_data’ sent from iNEMO-M1 discovery board towards host unit:

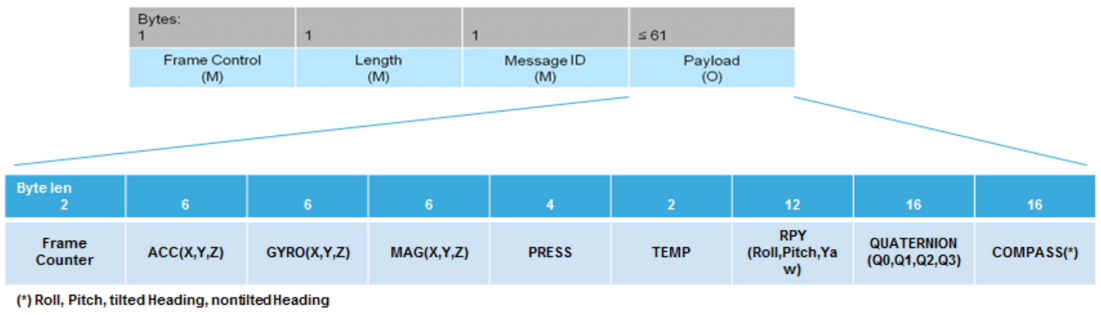


Figure 31: Fields of Acquisition Data frame

For detailed description of communication protocol specifications and frame formats, please refers to UM1744 from STMicrolectronics.

4.4.2 Power characterization of iNEMO module

The system developed for measure power consumption is described in the below state machine model.

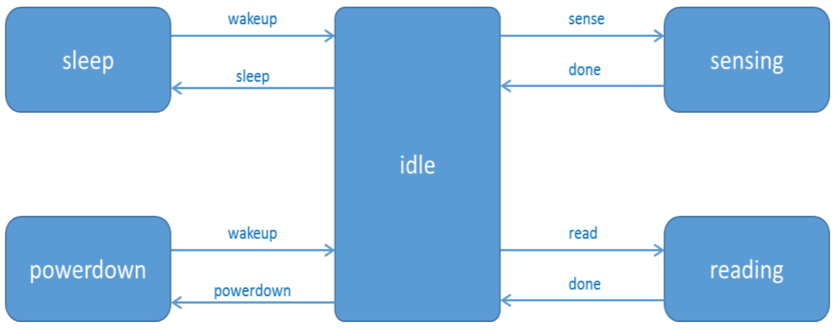


Figure 32: State machine power model

In this scheme, in term of power consumption, the device can assume 5 different states:

- **Power down**: this state represents the condition of minimum power dissipation (device is power-supplied);
- **Sleep**: some minimum functionalities are provided (for instance, internal registers memory retention);
- **Idle:** system turned on without performing any operation;
- **Sensing**: this state represents the condition on which the system is sensing actual data generated by external stimuli. Generally, this state incorporates A/D conversion to make data available in digital format. In this state, converted data set are retained internally into sensing device registers;
- **Reading**: this state represents the condition on which the device is asked to send externally data stored in the internal registers.

4.4.2.1 Measurement methodology

The iNEMO module characterization consisted to measure the power consumption of its two sensor devices (accelerometer and gyroscope). In fact, the information provided in their respective datasheets does not go in detail about the power figures of their operating modes.
For the microcontroller and voltage regulator it could be possible to use the power characterization data reported in the technical documentation (device datasheets).

4.4.2.1.1 Workbench hardware architecture

For performing the actual measurement operation, we decided not to use the iNEMO-M1 module, as it is very small and miniaturized module. It makes difficult to connect equipment probes (for instance, from the scope) and to add any additional measurement circuitry. It was decided to use another hardware platform for our purposes.
The picture below shows a scheme of used measuring bench:

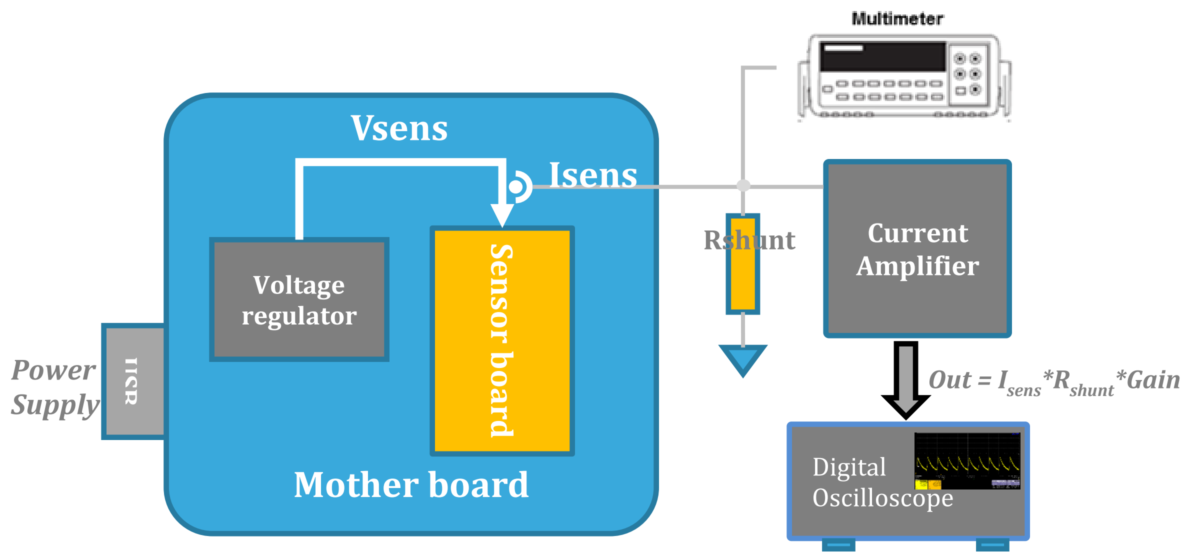


Figure 33: Scheme of the measurement bench

During our test sessions, sensor board is biased with a constant voltage value (3V from the voltage regulator), while current in the sensor depends on the sensor operational state: power characterization will consist to measure this current.
Since current values can be in the order of few µA, for getting a proper amplitude level and see its shape on the scope, it has been necessary to use a signal amplifier that amplifies the sensor current; its output is a voltage proportional to the sensor current.
These voltage waveforms are viewed with a digital oscilloscope, while a digital multimeter, configured to operate as an amperometer, has been used to measure the current rms value.

The *mother board* is an evaluation board from STMicroelectronics (STEVAL-MKI109V2) able to host the sensor device to be characterized (Figure 34).
STEVAL-MKI109V2 is an electronic board designed to provide users with a complete, ready to use platform for the evaluation of STMicroelectronics' MEMS products. It features a high-performance 32-bit microcontroller, which allows configuring the operating modes of the sensor under testing.

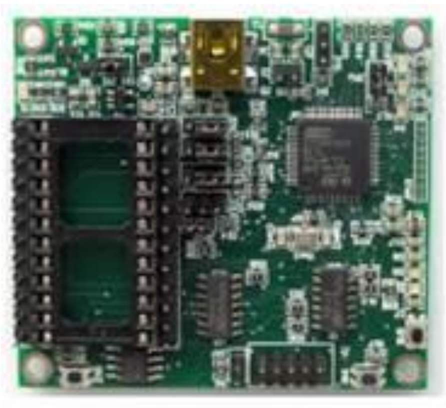


Figure 34: STEVAL-MKI109V2 evaluation board

It features a 24 pins dual in line socket for plugging chosen sensor daughter board. The latter, in turns, hosts the MEMS sensor device to be evaluated.

In Figure 35 the *Sensor board* is a daughter board containing the sensor IC device under testing and it can be easily plugged into the motherboard.
For our characterization the STEVAL-MKI106V1 (for the LSM303DLHC) and STEVAL-MKI107V2 (for the L3GD20) daughter boards have been used.

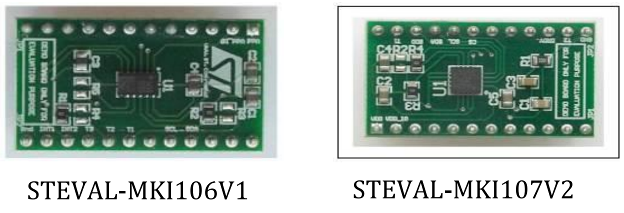


Figure 35: The two sensor daughter boards

As the amplitude of current flowing in the sensor device under test are very low (in the order of micro-ampere), the sensing circuit converts and amplifies it. By means of a shunt resistor, the current signal is converted into a voltage signal with levels that can be appreciated by a digital oscilloscope.
Here below the schematic of the current sensing circuit is shown:

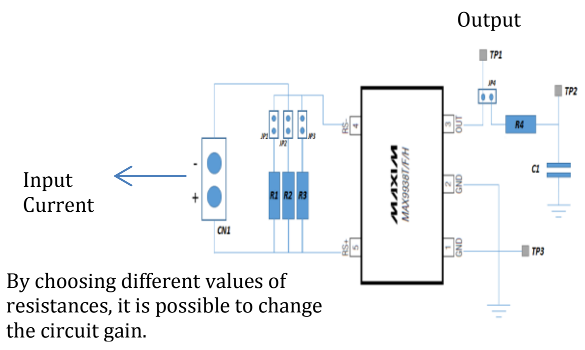

Output
TP1
TP2
TP3
JP1
JP2
JP3
JP4
R1
R2
R3
R4
C1
CN1
Input Current
MAX9938T/F/H
OUT
GND
RS-
RS+
By choosing different values of resistances, it is possible to change the circuit gain.

Figure 36: Current amplifier circuit schematic

Jumper JP_x (x=1,2,3) allows to choose the appropriate shunt resistance (according to the magnitude range of input signal).
Sensor current I_{sens} flows through the shunt resistor R_x and produces a voltage signal, at the input of OpAmp (MAXIM MAX9938), with a value equal to:

$$V_{sens} = I_{sens} * R_x$$

The amplifier gain is equal to 50, so the output voltage signal (V_{sens}) at *TP1* test point will be:

$$V_{out} = V_{sens} * 50$$

Oscilloscope probe can be connected on *TP1*.

4.4.2.1.2 Firmware general description

The STEVAL-MKI109V2 is supported with a Microsoft Windows™-based SW application called UNICO. This tool may be used as a simple real-time sensor device demonstrator or to test device behaviour and performance.
It allows easy monitoring of the sensor internal register status and allows making changes to them according to the intended use.
The revision fully supporting the STEVAL-MKI106V1 and STEVAL-MKI107V1 are:

- eMotion V3.0.6 : Firmware library for programming the sensor device (needs an IDE tool for this)
- Unico Rev. 3.0.1.0 beta : GUI for configuring the sensor device by PC

Firmware has been written using the IAR embedded workbench IDE that provides functionalities for programming the board and for its debugging.
Some of the files included in the eMotion firmware have been modified for the aim of the power characterization. Next sections detail how and which files have been customized.

4.4.2.2 LSM3003DLH Accelerometer and Magnetometer Sensor

In this paragraph, the LSM303DLHC IC main features (integrating both one accelerometer and one magnetometer sensors) will be shown. The steps necessary to configure the sensor operating modes and a description of the measuring procedure for power consumption estimation will be also shown.
The firmware code for setting the sensor operating modes is provided too.

4.4.2.2.1 LSM303DLHC general description

The LSM303DLHC is a system-in-package featuring a 3D digital linear acceleration sensor and a 3D digital magnetic sensor.
It includes an I2C serial bus interface that supports standard and fast mode (respectively, 100 kHz and 400 kHz).
The device can be configured to generate interrupt signals by inertial wake-up/free-fall events as well as by the position of the device itself. Thresholds and timing of interrupt generators are programmable by the end user.
Magnetic and accelerometer blocks can be enabled or put into low power mode separately.

4.4.2.2.2 Accelerometer conversion settings

The accelerometer sensor, during its operating modes can transit through all the states listed below.
When the sensor is powered on, the device automatically enters in Power-down (or Sleep mode).

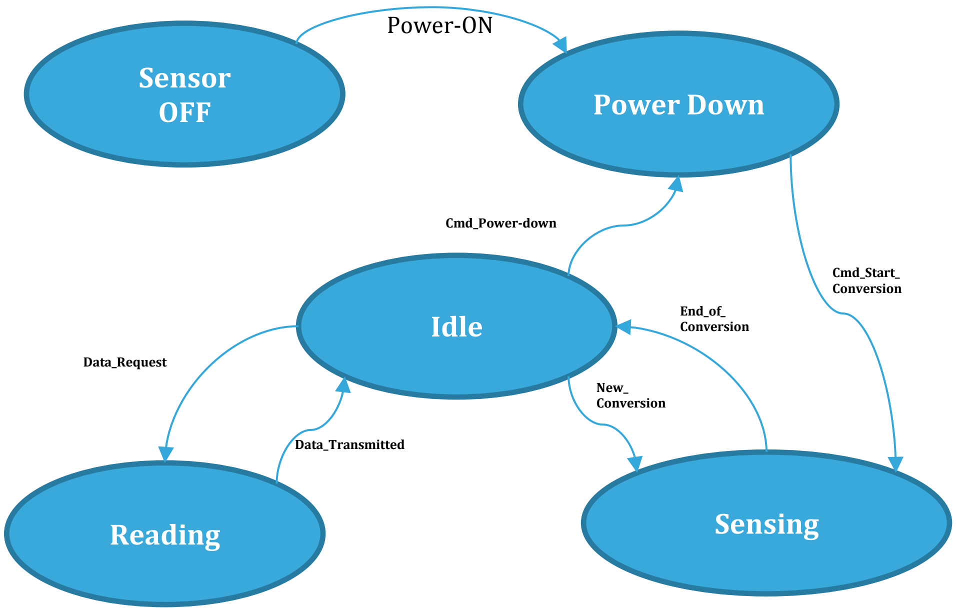


Figure 37: Accelerometer sensor state machine

Before entering the sensing state, the following parameters must be set:

- *Output data rate* (ODR): each conversion process consists to transform the acceleration stress, in input, into a correspondent voltage level and then convert it to digital information. The rate at which the input signal is converted depends on the output data rate selected. The following table shows, the possible ODRs according to the selected resolution:

ODR (Hz)	1	10	25	50	100	200	400	1344	1620	5376
Normal	•	•	•	•	•	•	•	•		
Low power	•	•	•	•	•	•	•		•	•

- *Number of axes* to enable: the accelerometer allows to acquire acceleration data in the 3 directions (3D). Anyway, just one or two axes can be enabled to save energy.
- *Conversion mode*: according to the preferred resolution of converted data provided at the output of the sensor, two different modes can be selected:
  - Normal mode (12-bits resolution)
  - Low-power mode (10-bits resolution).

After these settings have been completed the sensor enters in ‘Single Continuous’ conversion mode: a data set is provided at the selected ODR.
Sensor can be configured to operate also in ‘Multi Continuous’ conversion mode (see next section).
Here below some more details are given to describe the state machine and the different operating modes of the accelerometer device.

4.4.2.2.3 Matching of sensor operative modes with the power state representation

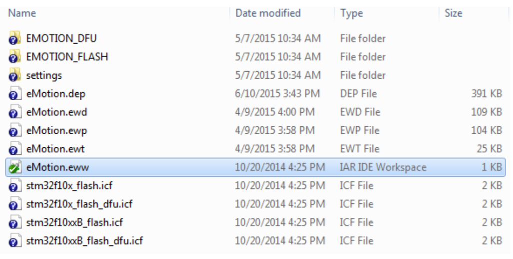
The actual sensor operative modes of Figure 37 can be associated to the ones listed in more generic power state machine shown in Figure 32:

- *Idle mode*: device is powered-on but not performing any operation. This state is assumed when a sense & convert operation has been just completed;
- *Sensing mode*: the actual sense and convert operations. We have two modes for getting and storing data after conversion:
    - Continue conversion: the converted data is stored in one internal register produced at programmed ODR frequency; if no reading occurs, a new data will overwrite the previous one;
    - Multi conversion: at established ODR rate, multiple conversions are done and data are stored in an internal FIFO stack. Multi conversion support two further different modes: 1) FIFO mode: a number of FIFO slots are chosen for storing converted data. When this level is reached, an interrupt (watermark int) can be generated to inform a host unit (MCU) about data availability. Until FIFO stack is not read, data remain in the FIFO and won't be overwritten. 2) STREAM mode: Each converted value is placed inside the FIFO. Unlike the FIFO mode, when the FIFO is full, subsequent conversions overwrite the older data.
- *Reading mode:* the receipt of a reading command from the host microcontroller let the sensor to switch in this state. Generally, the host asks for a reading operation after a conversion 'completed' trigger is raised from the sensor itself.
- *Power-down (or Sleep) mode:* almost all internal blocks of the device are switched off to minimize power consumption. Only the host communication interfaces are active.

4.4.2.2.4 Description of the firmware settings

To perform a measurement session, the workbench must be properly arranged and the STEVAL-MKI109V2 mother board has to be connected to a PC through its USB connector. USB port also provides the power source for supplying the workbench. Then, the mother board has to be programmed for configuring the sensor operations. At this aim:

- Inside IAR software IDE, open the emotion.eww project placed inside the INEMOM1_POWERCAR.

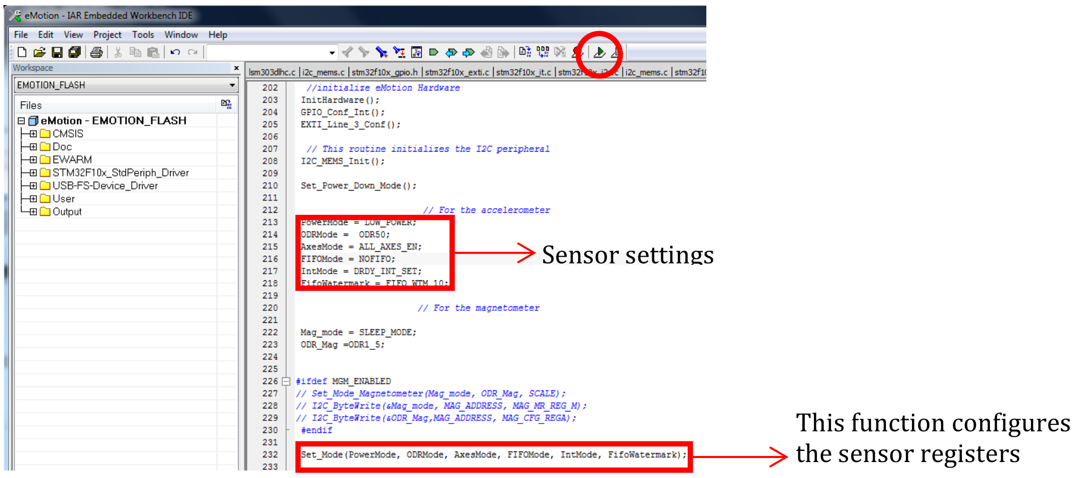


Name	Date modified	Type	Size
EMOTION_DFU	5/7/2015 10:34 AM	File folder	
EMOTION_FLASH	5/7/2015 10:34 AM	File folder	
settings	5/7/2015 10:34 AM	File folder	
eMotion.dep	6/10/2015 3:43 PM	DEP File	391 KB
eMotion.ewd	4/9/2015 4:00 PM	EWD File	109 KB
eMotion.ewp	4/9/2015 3:58 PM	EWP File	104 KB
eMotion.ewt	4/9/2015 3:58 PM	EWT File	25 KB
eMotion.eww	10/20/2014 4:25 PM	IAR IDE Workspace	1 KB
stm32f10x_flash.icf	10/20/2014 4:25 PM	ICF File	2 KB
stm32f10x_flash_dfu.icf	10/20/2014 4:25 PM	ICF File	2 KB
stm32f10xxB_flash.icf	10/20/2014 4:25 PM	ICF File	2 KB
stm32f10xxB_flash_dfu.icf	10/20/2014 4:25 PM	ICF File	2 KB

Figure 38: The eMotion project

- In file 'main.c' there some parameters and API functions for configuring the sensor registers (see highlighted parts in Figure 39);

- After the parameter settings are completed, the firmware library has to be compiled and the binary code downloaded into MCU by pressing the ‘make’ button (red circle highlighted in Figure 39).


eMotion - IAR Embedded Workbench IDE
File Edit View Project Tools Window Help
Workspace
EMOTION_FLASH
Files
eMotion - EMOTION_FLASH
CMSIS
Doc
EWARM
STM32F10x_StdPeriph_Driver
USB-FS-Device_Driver
User
Output
lsm303dlhc.c | i2c_mems.c | stm32f10x_gpio.h | stm32f10x_exti.c | stm32f10x_it.c | i2c_mems.c
Sensor settings
This function configures the sensor registers

```
 //initialize eMotion Hardware
InitHardware();
GPIO_Conf_Int();
EXTI_Line_3_Conf();

 // This routine initializes the I2C peripheral
I2C_MEMS_Init();

Set_Power_Down_Mode();

                        // For the accelerometer
PowerMode = LOW_POWER;
ODRMode =  ODR50;
AxesMode = ALL_AXES_EN;
FIFOMode = NOFIFO;
IntMode = DRDY_INT_SET;
FifoWatermark = FIFO_WTM_10;

                       // For the magnetometer

Mag_mode = SLEEP_MODE;
ODR_Mag =ODR1_5;


#ifdef MGM_ENABLED
// Set_Mode_Magnetometer(Mag_mode, ODR_Mag, SCALE);
// I2C_ByteWrite(&Mag_mode, MAG_ADDRESS, MAG_MR_REG_M);
// I2C_ByteWrite(&ODR_Mag,MAG_ADDRESS, MAG_CFG_REGA);
 #endif

Set_Mode(PowerMode, ODRMode, AxesMode, FIFOMode, IntMode, FifoWatermark);
```

Figure 39: The main.c module with highligted the parameters to configure

4.4.2.3 Accelerometer sensor consumption measurements

This paragraph enters in detail about the measurement procedure for power consumption estimation and, for some operating modes; it provides the actual power/energy level consumed in each state assumed by the sensor.

4.4.2.3.1 Power-down state

When the sensor is powered on, it goes in the *Power-down or Sleep* state [1]: its power consumption is the minimum possible.
The typical current RMS value reported in the datasheet is 1uA. This led to set a value of the Rsense resistance for the current amplifier circuit equal to 4.6k.
In Figure 40 it is shown the current waveform in Power-down mode; as it can be easily seen, the peak value is 2V, that in turn corresponds to a current of 8.7 uA (maximum peak).
The rms current value measured by the multimeter varies between 1.2 and 1.4uA.

[1] For the accelerometer of in LSM303DLHC, these two states are the same.

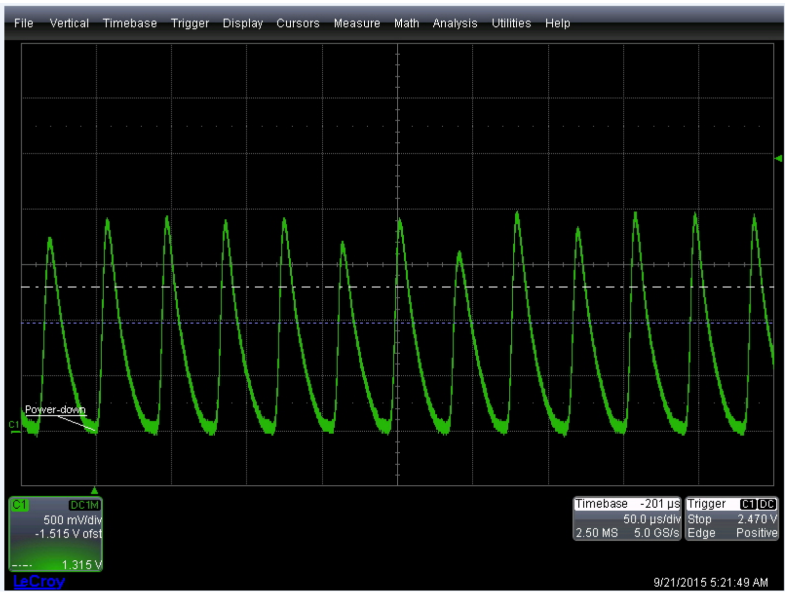


Figure 40: Power down current waveform

Power consumed in Power-down is get multiplying the Vcc voltage (2.986V) for the Irms value:

Irms	Power µW
1.4	4.18

4.4.2.3.2 Sensing state

Regardless the ODR parameter, a basic energy measurement can be evaluated for each conversion cycle: it includes the actual sensing and convert phase, the data storing and the consequent idle state. Then, a new conversion is triggered again automatically by internal logic circuitry.
Let assume a fixed *ODR* as output data rate. In this case each conversion-idle state cycle will last 1/ODR s. In this time interval, the sensor performs the data sampling, the A/D conversion and data storing on its internal register.
The following time diagram shows the whole cycle sequence:

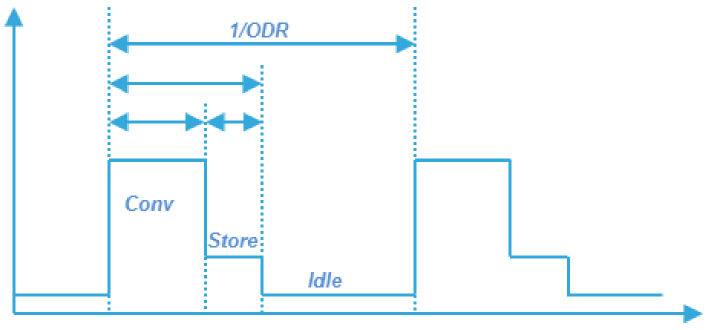


Figure 41: Typical sense, conversion, idle cycle

For each cycle and a set of sensor settings, the energy amount has been measured. In other words, it will be given the energy necessary to perform a complete sensing phase, including the idle phase, where the sensor performs no operation.
Of course, changing the sensor settings, the energy quantity per cycle will vary consequently.

Single continuous conversion

When in Power-down state, once the preferred sensor operating modes has been set, by just configuring ‘low power’ or ‘normal’ mode the sensing phase begins.
In continuous conversion mode the sensor internal logic performs a conversion operation at ODR frequency. Each data is then stored in an internal register. Next conversion occurrence will overwrite this data, if an external host (like an MCU) has not read it.

The formula used for the energy calculation is:

$$E = I_{RMS} * V_{BIAS} * \frac{1}{ODR} \quad (1)$$

where:

I_{RMS} : the RMS value of the current measured with the multimeter at the fixed ODR frequency.
V_{BIAS} : the daughter board voltage biasing equal to 2.986 V

At various sensor settings, the following tables show the energies amount needed for doing a complete conversion.
Here below the measurements results for the normal mode and low power mode, at different ODR and enabling 3 and 1 axis are shown:

ODR	10	Hz	50	Hz	200	Hz	400	Hz	1344	Hz
Current/Energy	**Irms uA**	**Energy uJ**	**Irms uA**	**Energy uJ**	**Irms uA**	**Energy uJ**	**Irms uA**	**Energy uJ**	**Irms uA**	**Energy uJ**
3 axes	5/5.2	1.49/1.55	13.0/13.3	0.776/0.794	43.0/43.5	0.642/0.649	81.7/82.1	0.610/0.613	199.5/199.	0.443/0.445
1 axis	5/5.1	1.49/1.52	12.7/13.2	0.758/0.788	42.6/43.4	0.636/0.648	81.6/81.9	0.609/0.611	198.5/198.	0.441/0.441

Figure 42: Normal mode

ODR	10	Hz	50	Hz	200	Hz	400	Hz	1620	Hz
Current/Energy	**Irms uA**	**Energy uJ**	**Irms uA**	**Energy uJ**	**Irms uA**	**Energy uJ**	**Irms uA**	**Energy uJ**	**Irms uA**	**Energy uJ**
3 axes	3.8/4.1	1.13/1.22	7.6/7.8	0.454/0.466	21.0/21.4	0.314/0.320	38.9/39.3	0.290/0.293	110.0/112.	0.203/0.205
1 axis	3.8/4.0	1.13/1.19	7.4/7.7	0.442/0.460	20.8/21.4	0.311/0.320	38.8/39.2	0.290/0.293	109.9/110.	0.203/0.203

Figure 43: Low Power mode

The energy per cycle decreases as ODR increases, as the t_{Idle} time is shorter compared to the other times (t_{Conv} and t_{Store}).
The two pictures below show the current waveforms recorded fixing an ODR of 10 Hz.
The first one on the left shows one conversion cycle, while the second one on the right shows four conversion cycles:

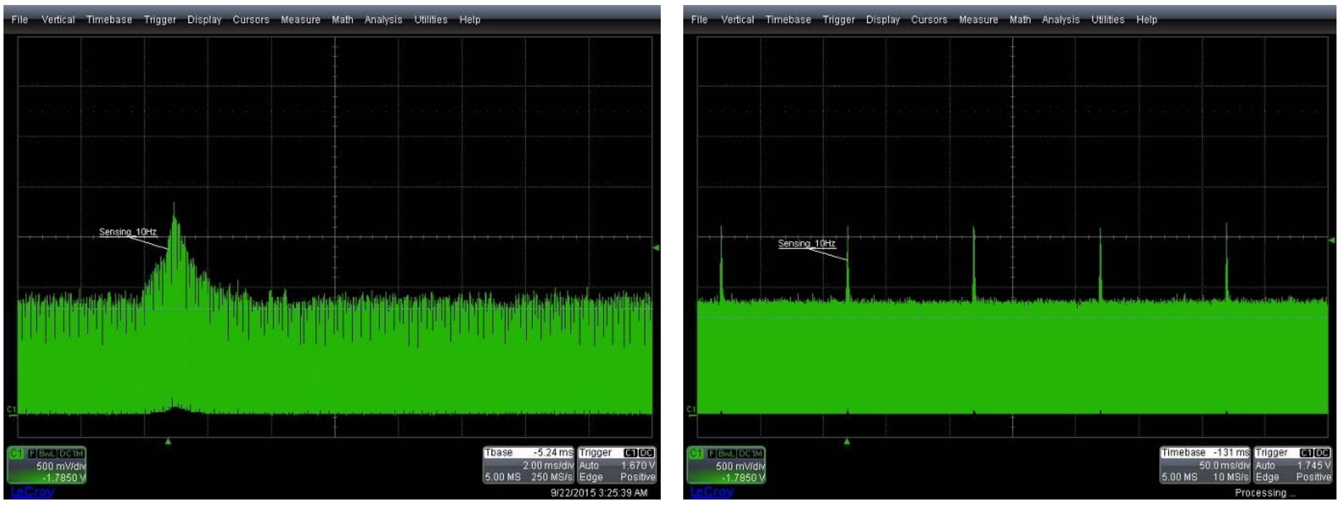


Figure 44: Current waveforms during conversion: one and four conversion cycles

Multiple Continuous Conversion

In multiple conversions mode, converted data are stored into a FIFO buffer. The buffer consents to collect up to 32 data.
It is possible to configure a watermark level that indicates the number of slots to be filled. The sensor logic can be programmed to generate a logical signal that indicates the attainment of the watermark level. The latter can be connected to a host MCU and treated as an asynchronous signal (external interrupt) for indicating that the data are available to be read.
Firmware APIs allow to set the watermark level, the multiple conversion mode (FIFO) and to enable the generation of the logical signal.

Reading state

After a sense operation is completed (single or multiple), the sensor can generate an asynchronous signal (DATA_READY) for starting a reading operation from an host device (generally a MCU). This cause the beginning of Reading phase (referred to the sensor).
During reading, an I2C communication is initiated between the MCU and the sensor. Total power consumption at this stage must be considered as the sum of the single power consumption of each device involved in data transfer: the sensor, the two pull-up resistors of I2C bus and the MCU (

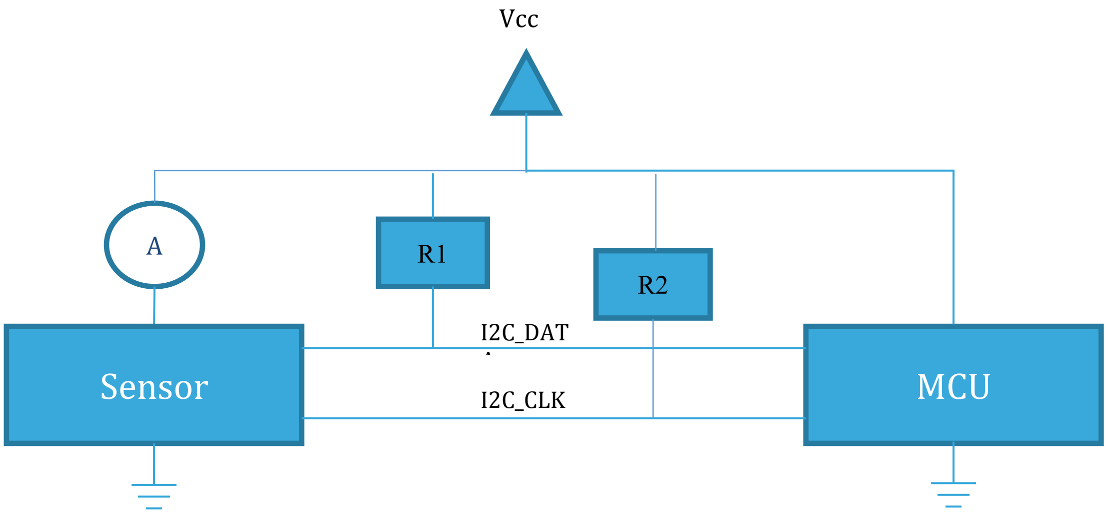


Figure 45).

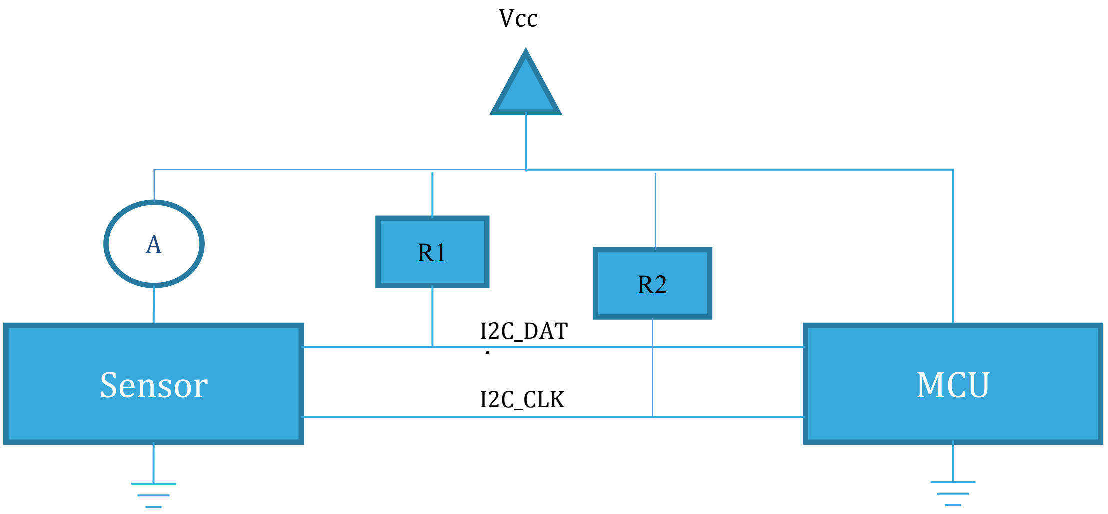


Figure 45: Devices and components involved during reading phase

According to what stated above, we have:

$$P_{SYS} = P_{ACC} + P_{R1} + P_{R2} + P_{MCU}$$

where:
P_{SYS}: total power consumption of the system during reading phase;
P_{ACC} : power consumed by the accelerometer to send the data;
P_{R1} and P_{R2} : power consumed by the pull up resistances used during an I2C communication;
P_{MCU} : power consumed by the I2C MCU peripheral;

Let us see how each contribute has been computed.

P_{ACC}

If the system is properly configured, a data transfer from the accelerometer to the microcontroller starts immediately after the 'Sensing' state ends, when a reading command is issued by the MCU.
Figure 46 highlights the instants when data transmission starts and ends (red pulse) referred to a single data set transfer. Duration is equal to 192µs and does not depend on the sensor settings.

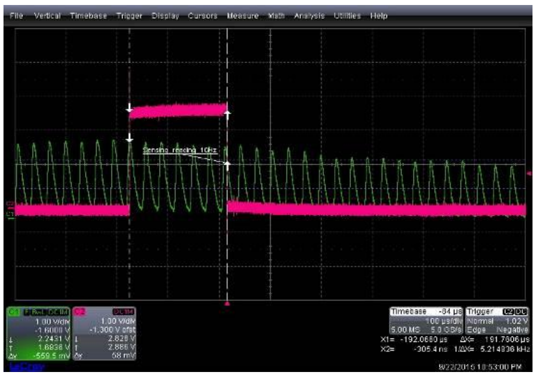

File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
Sensing reading 10Hz

Figure 46: The duration of the reading phase

The methodology used to measure P_{ACC}, due to the reading operation, consists to subtract the power consumption of a purely 'sense' operation (discussed on section 0) from the consumption of a sense and read operation. The latter can be easy measured with the multimeter.
The below figure shows the results of the measurements for the 'Sensing' +'Reading' (for a cycle):

ODR	10 Hz		50 Hz		200 Hz		400 Hz		1344 Hz	
Current/Energy	Irms uA	Energy uJ	Irms uA	Energy uJ	Irms uA	Energy uJ	Irms uA	Energy uJ	Irms uA	Energy uJ
3 axes	5.4	1.61	13.7	0.818	45.3/45.7	0.676/0.682	85.6/85.9	0.639/0.642	212.4/212.6	0.471/0.472

Figure 47: Normal mode 'Sensing' + 'Reading'

The difference between the correspondent measurements listed in Figure 47 gives:

ODR	10 Hz		50 Hz		200 Hz		400 Hz		1344 Hz	
Delta Curr/En	Delta Irms uA	Delta Energy	Delta Irms	Delta Energy	Delta Irms	Delta Energy	Delta Irms	Delta Energy	Delta Irms	Delta Energy
Values	0	0	0.4	0.03	2.2	0.032	3.7	0.029	13	0.029

Figure 48: The energy consumed by the accelerometer during its 'Reading' phase

From the above table, the energy for reading operation used by accelerometer is:

$$E_{acc} \cong 30\ nJ$$

Note: The energy amount for reading operation is independent from ODR value and the above table does confirm it.

P_{R1} and P_{R2} pull-up resistances

An I2C communication bus consists of two serial lines: a clock line (ranging from 100KHz up to 400KHz) and data line for data transfer.
A master device is used to drive the clock line, while the master or the slave drive the data line according to the direction of data transfer.
As shown in the Figure 45, two resistances of pull up of 4.7 KOhm connect the Vcc to the DATA and CLOCK lines.
When a '0' logic level in the clock line has to be set from the master, its internal logic generates a path versus ground for the pull up resistor. The pull up resistor is in open drain when a '1' level has to be set. The same considerations are valid for the data line.

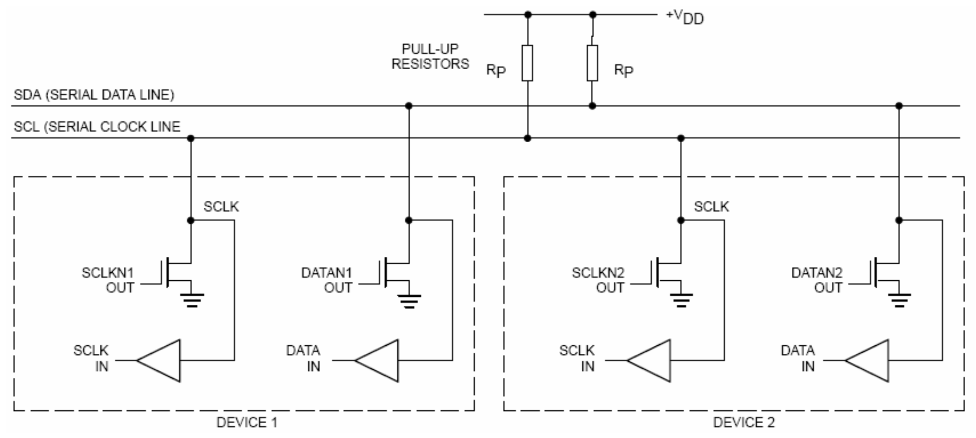


Figure 49. I2C internal logic

During a ‘0’ logic level, a current will flow through the interested pull-up resistor. Estimated power consumption for the clock line can be evaluated considering the signal clock has a 50% of duty cycle.
When the clock signal is ‘0’, current that flows through the pull-up resistor can be obtained using the below formula:

$$I_{R2} = V\_dd/R2$$

Where V_dd is about 3V.

Considering the 50% duty cycle, the RMS of a square waveform is:

$$I_{R2\_rms} = \frac{I_{R2}}{\sqrt{2}} = \frac{Vdd}{R2*\sqrt{2}}$$

Power consumed by R2 during a reading is:

$$P_{R2} = V_{dd} * I_{R2\_rms} = Vdd^2/(R2 * \sqrt{2})$$

Energy consumed in a reading by the R2 resistance is:

$$\boldsymbol{E_{R2} = P_{R2} * T_{read} = \frac{Vdd^2}{R2 * \sqrt{2}} * T_{read}}$$

The same considerations are valid for the energy consumed by R1:

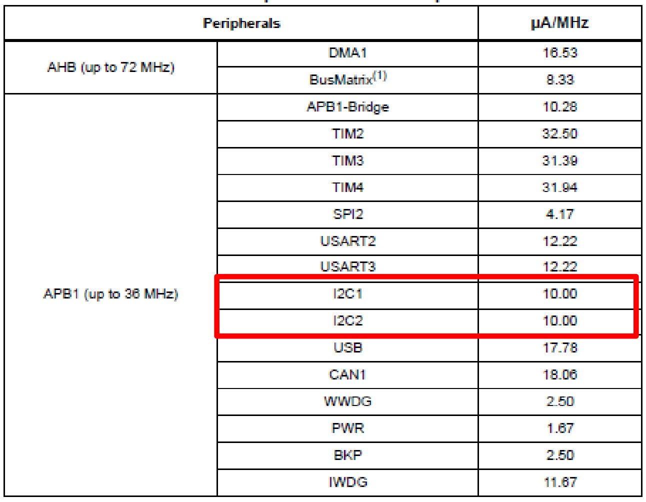
$$\boldsymbol{E_{R1} = P_{R1} * T_{read} = \frac{Vdd^2}{R1 * \sqrt{2}} * T_{read}}$$

From numerical point of view, we have:

$$\boldsymbol{E_{R2} = \frac{Vdd^2}{R2 * \sqrt{2}} * T_{read} = \frac{(2.986)^2}{4.7 * 10^3 * \sqrt{2}} * 192 * 10^{-6} = 257nJ}$$

P_{MCU} - I2C peripheral power consumption

Power consumed by the MCU I2C peripheral can be obtained using the below picture:



Peripherals		µA/MHz
AHB (up to 72 MHz)	DMA1	16.53
	BusMatrix(1)	8.33
APB1 (up to 36 MHz)	APB1-Bridge	10.28
	TIM2	32.50
	TIM3	31.39
	TIM4	31.94
	SPI2	4.17
	USART2	12.22
	USART3	12.22
	I2C1	10.00
	I2C2	10.00
	USB	17.78
	CAN1	18.06
	WWDG	2.50
	PWR	1.67
	BKP	2.50
	IWDG	11.67

Figure 50: STM32 peripherals power consumption

The I2C peripheral input clock frequency must be at least 2MHz (Standard mode – 100KHz) and 4MHz (Fast Mode - 400 KHz). According to Figure 50, as the system operates in fast mode, the current consumed by the I2C peripheral is:

$$I_{I2C} = 40\mu A$$

Power consumed by the I2C peripheral is:

$$P_{I2C\_MCU} = V_{dd} * I_{I2C}$$

and the energy for a complete data transfer is:

$$E_{I2C\_MCU} = P_{I2C\_MCU} * T_{read} = V_{dd} * I_{I2C} * T_{read} \cong 23nJ$$

Summarizing, the energy consumption in a cycle related to a reading operation for the whole system depicted in Figure 45 is the sum of the various energy contribution.

$$\boldsymbol{E_{read} = E_{acc} + 2 * E_{R2} + E_{I2C_{MCU}} = [30 + 2 * (257) + 23]nJ = 567n}$$

4.4.2.3.3 FIFO conversion mode: two cases of study

This paragraph will introduce the measurement methodology used to evaluate the energy consumption for two cases of study: FIFO mode at two different ODRs. The table below shows the two group of settings:

ODR	Range	Conversion mode	Watermark	Sampling mode	Read mode
1344 Hz	+/- 16g	12 bit	25	Continuous	Auto-increment
400 Hz	+/- 16g	12 bit	24	Continuous	Auto-increment

Figure 51: The two groups of settings

Before discussing about the measurement results, in the next paragraph a brief description of the FIFO mode (one of the multiple continuous conversion modes) will be presented.

Introduction to FIFO operating mode

In FIFO mode, the FIFO buffer continues filling until the watermark or the overrun (OVRN) level (32 slots filled) are reached, then it stops collecting data and the FIFO buffer content remains unchanged until the Bypass mode, used to clear FIFO buffer, is selected.
The steps required to configure this mode and to read the data are below shown:

1. Enable multi conversion mode (Resolution, Enable FIFO mode, axes, number of samples ≤32,…)
2. Start FIFO mode.
3. Wait OVRN or WTM interrupt
4. Read data samples from FIFO

If a new batch conversion is requested:
5. Enable Bypass mode
6. Repeat from point 2

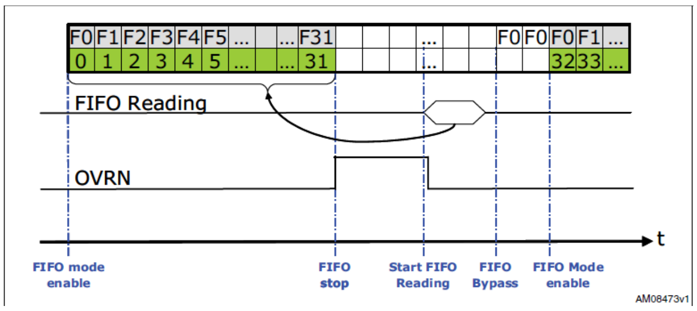


Figure 52: FIFO mode operations

All the steps from 1 to 5 consist in I2C communications between the master and slave.
For evaluating the energy amount for a complete FIFO operation, the sensor should go through steps 1,2,3,4. From practical point of view, as the actual current that is measured is the rms value, the cycle is executed continuously and steps 5 and 6 are used.

A time diagram of the logical transitions during FIFO data conversion and storing, including data sets reading via I2C bus, is shown below:

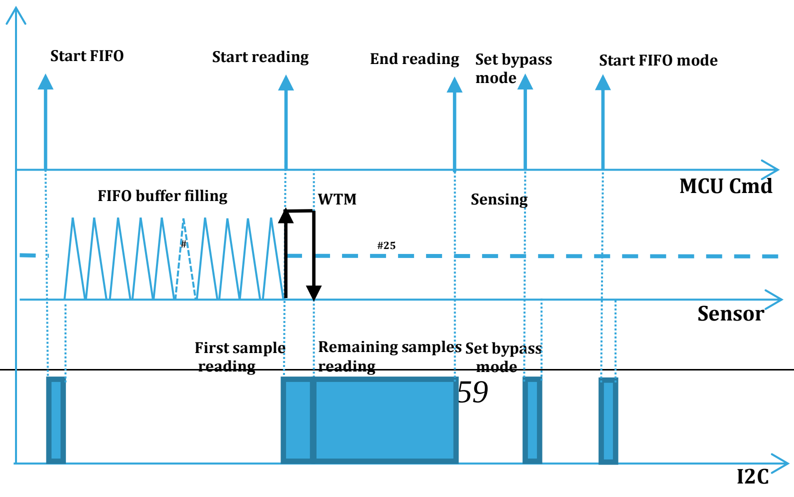


Figure 53: Logical transitions, sensor signals and I2C activity

The figures below show the oscilloscope waveforms during an entire FIFO process.
The central figure shows the entire FIFO process. In details:

- Cyan signal shows the I2C communication during samples transfer;
- Purple is a monitor generated from MCU to show command sequence;
- Green shows the current that flows through the sensor;
- Yellow shows the instant when the watermark signal is set and reset (driven by the sensor)

The main phases are zoomed in pictures 1,2,3:

- #1: the starting of FIFO filling process.
- #2: the starting of the reading phase.
- #3: the I2C transmission.

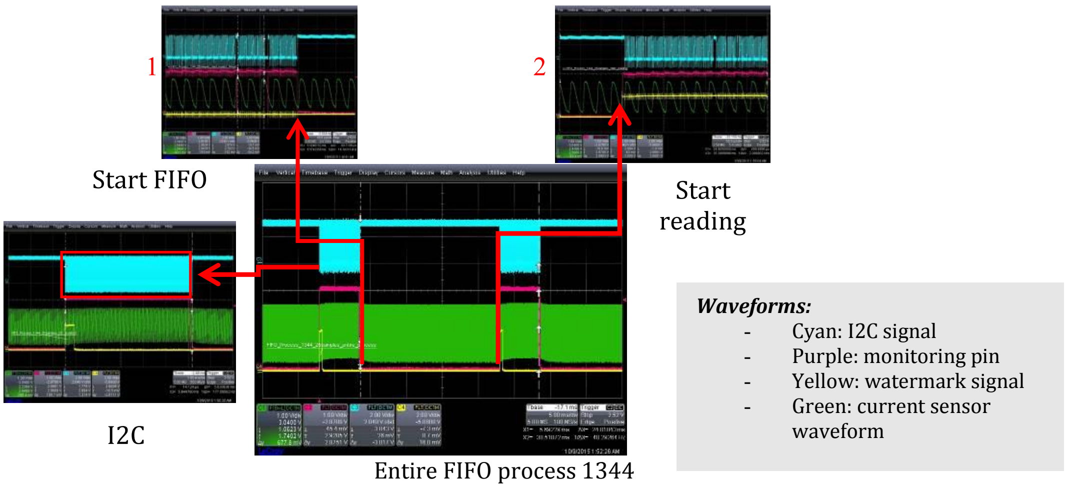


Figure 54: FIFO process waveforms

Case 1: 3-Axes, 12-bit, FIFO mode, ODR @1344,25 Samples

Total power consumption is computed as sum of the single power consumption of each device involved in data conversion and reading: the sensor, the two pull-up resistors of I2C bus and the MCU as shown in the

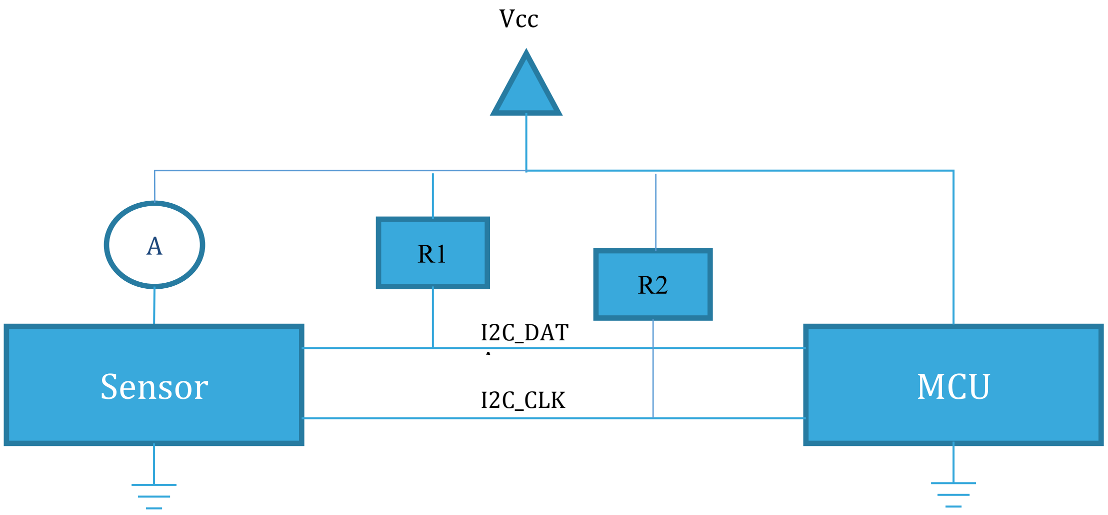
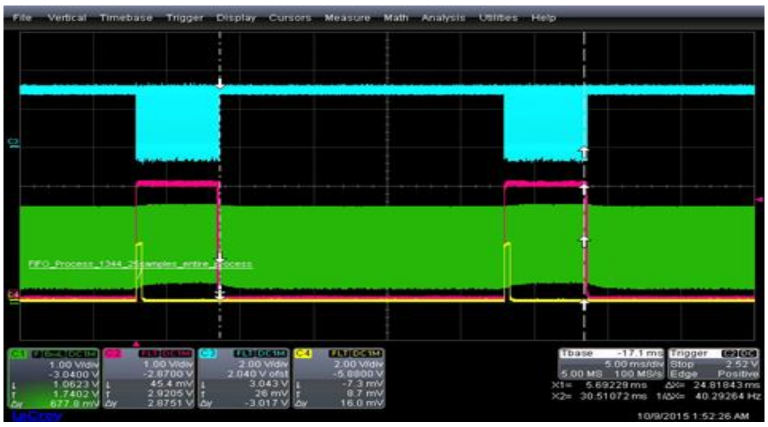


Figure 45.

Compute the power consumption of sensor (P$_{acc}$)

The below picture shows the waveforms generated during a complete FIFO process (filling FIFO and successive samples reading).
When the established numbers of sample data have been stored into the FIFO, the sensor sets a Watermark signal, at one of its output pin. This signal is reset when the first data set is read by the microcontroller (Figure 55: ODR 1344Hz watermark 25, FIFO mode and reading period-Yellow waveform)
A monitoring signal (purple waveform) is set when the reading phase starts, and is reset at the end (25 samples read).
Cyan signal shows the I2C activity during the reading phase.


File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
FIFO_Process_1344_25samples_entire_process
C1
C2
C3
C4
LeCroy

```
C1 1.00 V/div   C2 1.00 V/div   C3 2.00 V/div     C4 2.00 V/div     Tbase -17.1 ms   Trigger
   -3.0400 V       -2.8700 V       2.040 V ofst      -5.8800 V        5.00 ms/div    Stop 2.52 V
↓  1.0623 V     ↓  45.4 mV      ↓  3.043 V        ↓  -7.3 mV        5.00 MS 100 MS/s Edge Positive
↑  1.7402 V     ↑  2.9205 V     ↑  26 mV          ↑  8.7 mV         X1= 5.69229 ms   ΔX= 24.81843 ms
Δy 677.9 mV     Δy 2.8751 V     Δy -3.017 V       Δy 16.0 mV        X2= 30.51072 ms  1/ΔX= 40.29264 Hz
                                                                    10/9/2015 1:52:26 AM
```

Figure 55: ODR 1344Hz watermark 25, FIFO mode and reading period

With the parameter settings established above, the duration of complete process (FIFO filling and data reading) lasts (see Figure 54):

$$t_{FIFO} = 24.81ms$$

The I_{acc_rms} value reading got by the multimeter is:

$$I_{accc\_rms} = (209.2 \pm 0.2)\ \mu A$$

With a power supply voltage of

$$V_{dd} = 2.986\,V$$

the consumed power by the sensor is:

$$P_{acc} = V_{cc} I_{acc\_rms} = 2.986 * 209.2 * 10^{-6} \cong 625\,\mu W$$

Energy consumption is obtained multiplying the power consumed Pacc for the duration of the FIFO process.

$$E_{acc} = P_{acc} * t_{FIFO} = 625 * 10^{-6} * 24.81 * 10^{-3} \cong 15.5\,\mu J$$

P_{R1} and P_{R2} pull-up resistances

During data reading, there will be currents flowing through the two pull-up resistances. This in turns causes a power dissipation that must be accounted in the power consumption evaluation process.
The picture below shows the I2C activity during a samples set reading (25 samples).

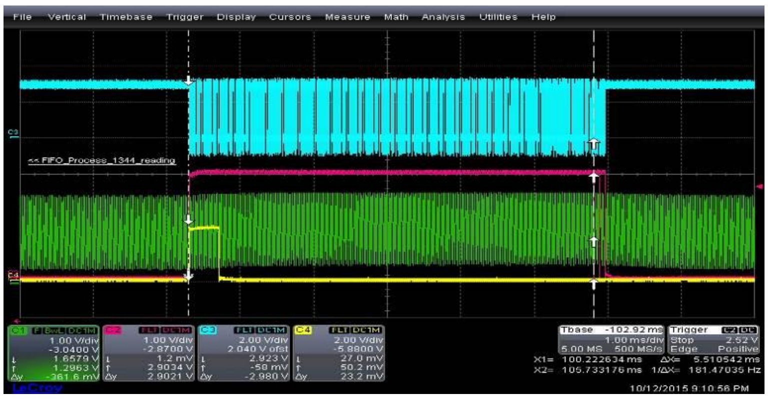

File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
<< FIFO_Process_1344_reading
Tbase -102.92 ms
Trigger
X1= 100.222634 ms ΔX= 5.510542 ms
X2= 105.733176 ms 1/ΔX= 181.47036 Hz
10/12/2015 9:10:56 PM

Figure 56: Duration of the reading period

From the scope, the duration of read period (for all 25 data sets) is:

$$t_{read} = 5.51ms$$

Current that flows through the pull-up resistances can be approximated to a square waveform with a 50% duty cycle.
Current peak is obtained dividing the V_{CC} for the R_{PULL-UP} value:

$$I_{PULL\_UP\_peak} = \frac{V_{cc}}{R_{PULL-UP}}$$

The rms value of a square waveform is obtained using the formula below:

$$I_{PULL-UP\_rms} = \frac{I_{PULL\_UP\_peak}}{\sqrt{2}} = \frac{V_{cc}}{R_{PULL-UP} * \sqrt{2}}$$

Power consumption is equal to:

$$P_{PULL\_UP} = V_{cc} * I_{PULL-UP\_rms} = \frac{{V_{cc}}^2}{R_{PULL-UP} * \sqrt{2}}$$

Energy consumed by one of the pull-up resistances during a 25 sample sets reading is equal to:

$$E_{PULL\_UP} = P_{PULL\_UP} * t_{read} = \frac{{V_{cc}}^2}{R_{PULL-UP} * \sqrt{2}} * t_{read} = \frac{(2.986)^2}{4.7*10^3*\sqrt{2}} * 5.51 * 10^{-3} = 7.4\ \mu\mathrm{J}$$

For the two pull-up resistors, then we have:

$$E_{2xPULL\_UP} = 14.8\ \mu J$$

P_{MCU} - I2C peripheral power consumption

For what has been told in section 0, the power consumption of the MCU is due to its I2C peripheral operations. Under this hypothesis, the current consumption has been taken from the table in Figure 50:

$$I_{I2C\_MCU} = 40\ \mu\mathrm{A}$$

Power consumption is equal to:

$$P_{I2C\_MCU} = V_{cc} * I_{I2C\_MCU}$$

Energy consumed by the micro I2C peripheral during the reading of all sample sets is:

$$E_{I2C\_MCU} = P_{I2C\_MCU} * t_{read} = V_{cc} * I_{I2C_{MCU}} * t_{read} = 658\ nJ$$

Summarizing, the energy consumption, for filling and reading FIFO, is the sum of the various energy contributions:

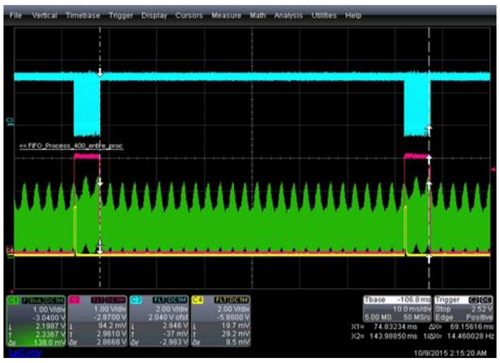
$$\boldsymbol{E_{FIFO\_25} = E_{acc} + E_{2xPULL\_UP} + E_{I2C\_MCU} = 30.96\ \mu J}$$

Case 2: 3-Axes, 12-bit, FIFO mode, ODR @400, 24 Samples

The approach followed to evaluate the global power consumption is the same of that followed in the previous case of study.

Compute the power consumption of sensor (Pacc)

The below picture shows the waveforms generated during a complete FIFO process.



```
C1  F BwL DC1M   C2  FLT DC1M   C3  FLT DC1M   C4  FLT DC1M
   1.00 V/div       1.00 V/div     2.00 V/div     2.00 V/div
  -3.0400 V        -2.8700 V      2.040 V ofst   -5.8000 V
↓  2.1987 V     ↓  94.2 mV     ↓  2.946 V     ↓  19.7 mV
↑  2.3367 V     ↑  2.9610 V    ↑  -37 mV      ↑  29.2 mV
Δy 138.0 mV     Δy 2.8668 V    Δy -2.983 V    Δy 9.5 mV

Tbase  -106.8 ms   Trigger
       10.0 ms/div Stop      2.52 V
5.00 MS  50 MS/s   Edge      Positive
X1=  74.83234 ms    ΔX=  69.15616 ms
X2= 143.98850 ms  1/ΔX=  14.460028 Hz
10/9/2015 2:15:20 AM
```

Figure 57: ODR 400Hz watermark 24, FIFO mode and reading period

With the parameter settings established above, the duration of complete process (FIFO filling and data reading) lasts (see Figure 56):

$$t_{\text{FIFO}} = 69.15ms$$

The $\boldsymbol{I_{acc\_rms}}$ value reading got by the multimeter is:

$$I_{acc\_rms} = (85.5 \pm 0.1)\ \mu A$$

With a power supply voltage of

$$V_{dd} = 2.986\ V$$

The consumed power by the sensor is:

$$P_{acc}\ = V_{cc} I_{acc\_rms} = 2.986 * 85.5 * 10^{-6}\ \cong\ 255\ uW$$

Energy consumption is obtained multiplying the power consumed Pacc for the duration of the FIFO process.

$$E_{acc} = P_{acc}\ *\ t_{FIFO}\ = 255 * 10^{-6}\ * 69.15 * 10^{-3} = 17.6u\text{J}$$

P_{R1} and P_{R2} pull-up resistances

During data reading, there will be currents flowing through the two pull-up resistances. This in turns causes a power dissipation that must be accounted in the power consumption evaluation process.
The picture below shows the I2C activity during a samples set reading (24 samples).

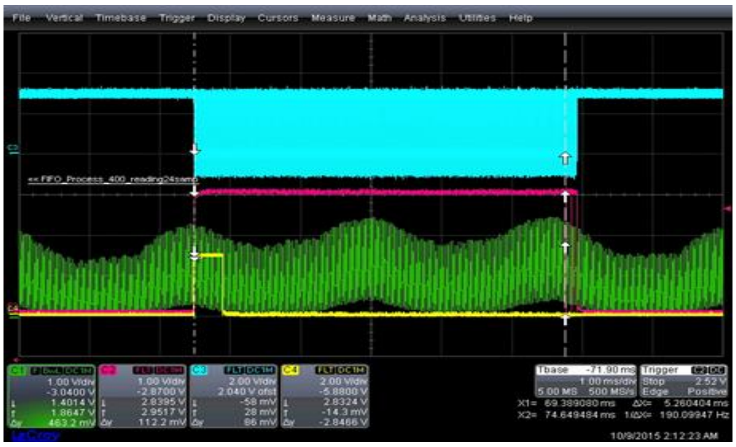

File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help

Figure 58: Duration of the reading period

From the scope, the duration of read period (for all 24 data sets) is:

$$t_{read} = 5.26ms$$

As seen in the paragraph 4.4.2.13, the current that flows through the pull-up resistances can be approximated to a square waveform with a 50% duty cycle.
Current peak is obtained dividing the V_{CC} for the $R_{PULL\text{-}UP}$ value:

$$I_{PULL\_UP\_peak} = \frac{V_{cc}}{R_{PULL-UP}}$$

The rms value of a square waveform is obtained using the formula below:

$$I_{PULL-UP\_rms} = \frac{I_{PULL\_UP\_peak}}{\sqrt{2}} = \frac{V_{cc}}{R_{PULL-UP} * \sqrt{2}}$$

Power consumption is equal to:

$$P_{PULL\_UP} = V_{cc} * I_{PULL-UP\_rms} = \frac{{V_{cc}}^2}{R_{PULL-UP} * \sqrt{2}}$$

Energy consumed by one of the pull-up resistances during a 24 sample sets reading is equal to:

$$E_{PULL\_UP} = P_{PULL\_UP} * t_{read} = \frac{{V_{cc}}^2}{R_{PULL-UP} * \sqrt{2}} * t_{read} = \frac{(2.986)^2}{4.7 * 10^3 * \sqrt{2}} * 5.26 * 10^{-3} = 7.0\ \mu\text{J}$$

For the two pull-up resistors, then we have:

$$E_{2xPULL\_UP} = 14.0\ \mu J$$

P_{MCU} - I2C peripheral power consumption

For what has been told in section 4.4.2.13, the power consumption of the MCU is due to its I2C peripheral operations. Under this hypothesis, the current consumption has been taken from the table in Figure 50:

$$I_{I2C\_MCU} = 40\ \mu\text{A}$$

Power consumption is equal to:

$$P_{I2C\_MCU} = V_{cc} * I_{I2C\_MCU}$$

Energy consumed by the micro I2C peripheral during the reading of all sample sets is:

$$E_{I2C\_MCU} = P_{I2C\_MCU} * t_{read} = V_{cc} * I_{I2C_{MCU}} * t_{read} = 628\ nJ$$

Summarizing, the energy consumption, for filling and reading FIFO, is the sum of the various energy contributions:

$$\boldsymbol{E_{FIFO\_24} = E_{acc} + E_{2xPULL\_UP} + E_{I2C\_MCU} = 32.22\ \mu J}$$

4.4.2.4 Remarks on Accelerometer measurements

The two diagrams below show the time variation of P_{sys} during the respective time intervals t_{FIFO} and the three energies (E_{acc}, $E_{PULL\text{-}UP}$, E_{I2C_MCU}) contribution to the global energy consumption (E_{FIFO_25} at ODR 1344Hz and E_{FIFO_24} at ODR 400 Hz)

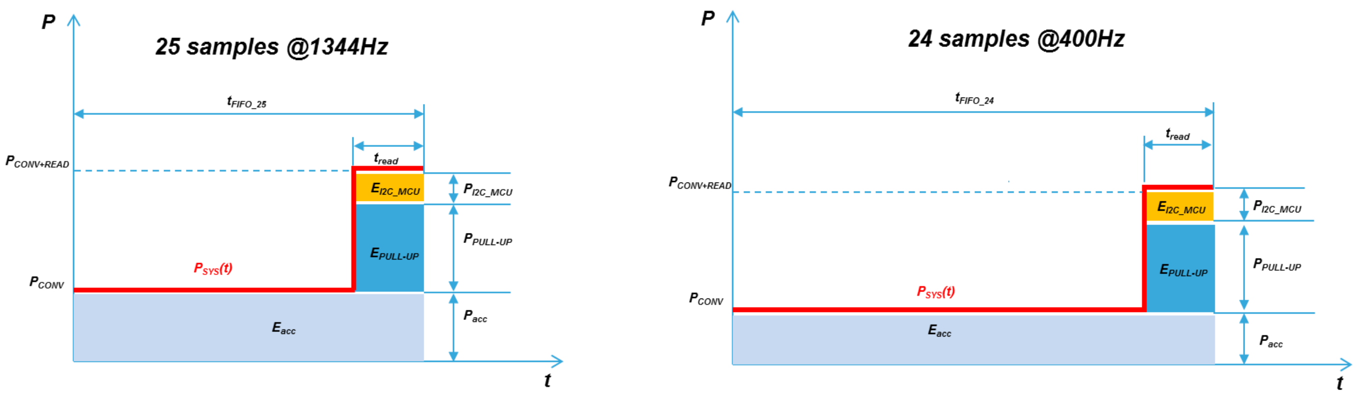


Figure 59: P_{SYS} behaviour 25 samples ODR 1344 vs 24 samples ODR 400 Hz

The two energies requested for FIFO conversion modes are quite similar, though the two ODR frequencies are very different:

$$\boldsymbol{E_{FIFO\_25} = E_{acc} + E_{2xPULL\_UP} + E_{I2C\_MCU} = 30.96\ \mu J}$$

$$\boldsymbol{E_{FIFO\_24} = E_{acc} + E_{2xPULL\_UP} + E_{I2C\_MCU} = 32.22\ \mu J}$$

The figure above explains the reason: the faster frequency mode implies a higher total power conversion (see the red curve named P_{sys}), while the slower conversion mode implies a longer time interval but lower power conversion. This translates in two energies values that result quite similar.
The two contributions E_{I2C_MCU} and $E_{PULL\text{-}UP}$ do not depend by ODR; they differ only for the number of sample readings (25 vs 24).

With these considerations, the two energies involved for each conversion mode, can be considered very similar and the following relationship exists:

$$E_{sys\_25} = \int_0^{t_{FIFO\_25}} P_{sys\_25} dt = \int_0^{t_{FIFO\_24}} P_{sys\_24} dt = E_{sys\_24}$$

4.4.2.5 L3G4200D Gyroscope Sensor

In this paragraph, the L3G4200D IC main features are shown. Even if the gyroscope assembled in iNEMO-M1 is the L3GD20, we have chosen to use the L3GD4200D as already available by our laboratory. By the way, the two sensors are quite similar in terms of power consumption. Slight differences reside in the selectable ODRs values (L3GD20: 95,190,380,760Hz; L3G4200D: 100,200,400,800 Hz)
The steps necessary to configure the sensor operating modes and a description of the measuring procedure for power consumption estimation are shown in the following sections.

4.4.2.5.1 L3G4200D general description

The L3G4200D is a low-power three axis angular rate sensor. It includes a sensing element and an I2C interface capable of providing the measured angular rate to the external world through a dedicated digital interface.
A moving mass, designed with the MEMS (Micro electrical mechanical system) technology is used to transduce the angular velocity in a voltage value.
The device may be configured to generate interrupt signals by an independent wake-up event. Threshold and timing of the interrupt generator are programmable by the end user on the fly.

4.4.2.5.2 Accelerometer conversion settings

The gyroscope sensor, during its operating modes can transit through all the states listed below. When the sensor is powered on, the device automatically enters in Power-down.

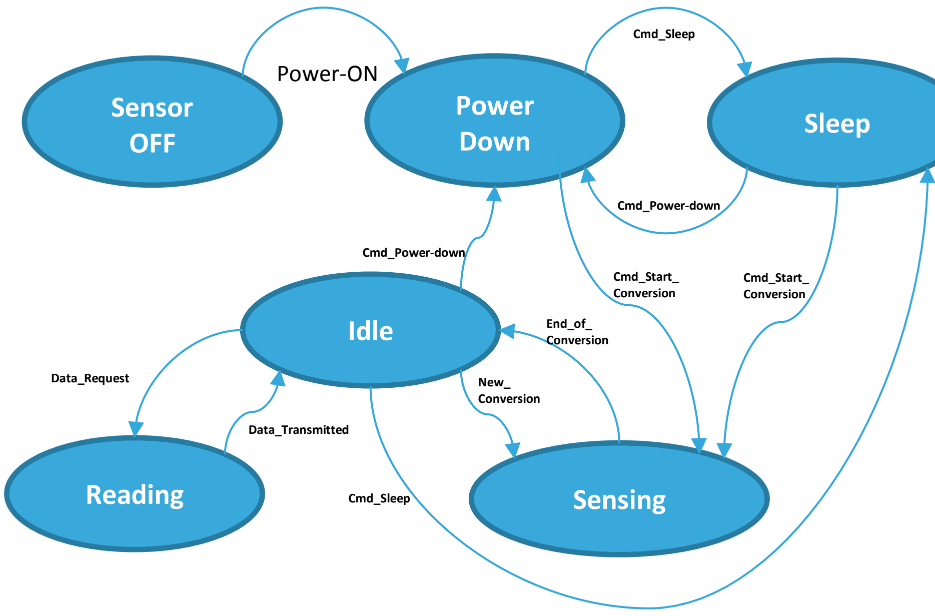


Figure 60: Gyroscope sensor state machine

Before entering the sensing state, the following parameters must be set:

- *Full scale*: the L3G4200D has a full scale of ±250/ ±500/±2000 dps
- *Output data rate* (ODR): each conversion process consists to transform the sensed angular velocity into a correspondent voltage level and then convert it to digital information. The rate at which the input signal is converted depends on the output data rate selected. The following table shows, the possible ODRs:

ODR (Hz)	100	200	400	800

- *Number of axes* to enable: the gyroscope allows acquiring angular velocity data in the 3 directions (roll, jaw and pitch). Anyway, just one or two axes can be enabled to save energy.
- *Conversion* mode: Single- or Multi-continuous (or FIFO) modes (in both cases data are generated at the ODR rate).

4.4.2.5.3 Matching of sensor operative modes with the power state representation

The actual operative sensor states of Figure 60 can be associated to the ones listed in more generic power state machine shown in Figure 32:

- *Idle mode*: device is powered-on but not performing any operation. This state is assumed after a sense and convert operation has been just completed;
- *Sensing mode*: the actual sense and convert operations. We have two modes for getting and storing data after conversion:
  - Continue conversion: the converted data is stored in one internal register produced at programmed ODR frequency; if no reading occurs, a new data will overwrite the previous one;

  - Multi conversion: at established ODR rate, multiple conversions are done and data are stored in an internal FIFO stack. Multi conversion support two further different modes: 1) FIFO mode: a number of FIFO slots are chosen for storing converted data. When this level is reached, an interrupt (watermark int) can be generated to inform a host unit (MCU) about data availability. Until FIFO stack is not read, data remain in the FIFO and won't be overwritten. 2) STREAM mode: Each converted value is placed inside the FIFO. Unlike the FIFO mode, when the FIFO is full, subsequent conversions overwrite the older data.
- *Reading mode:* the receipt of a reading command from a host microcontroller let the sensor to switch in this state. Generally, the host asks for a reading operation after a conversion 'completed' trigger is raised from the sensor itself.
- *Power-down mode:* almost all internal blocks of the device are switched off to minimize power consumption. Only the host communication interfaces are active.
- *Sleep mode:* the driving circuitry making the moving mass of the gyroscope oscillating is kept active. Turn on time from 'Sleep mode' to 'Sensing' is drastically reduced compared to when the device starts from Power-down mode.

4.4.2.6 Gyroscope sensor consumption measurements

This paragraph enters in detail about the measurement procedure for power consumption estimation limited to two operating modes. It provides the actual power/energy level consumed in each state assumed by the sensor.

4.4.2.6.1 Power-down state

When the sensor is powered on, it goes in the *Power-down* state*:* its power consumption is the minimum possible.
The typical current RMS value reported in the datasheet is 5µA. This led to set a value of the Rsense resistance for the current amplifier circuit equal to 10k.
In Figure 61 it is shown the current waveform in *Power-down* mode; as it can be easily seen, the peak value is 2.5 V that in turn corresponds to a current of 5µA (maximum peak).
The rms current value measured by the multi-meter varies between 2.0 and 2.2 uA.

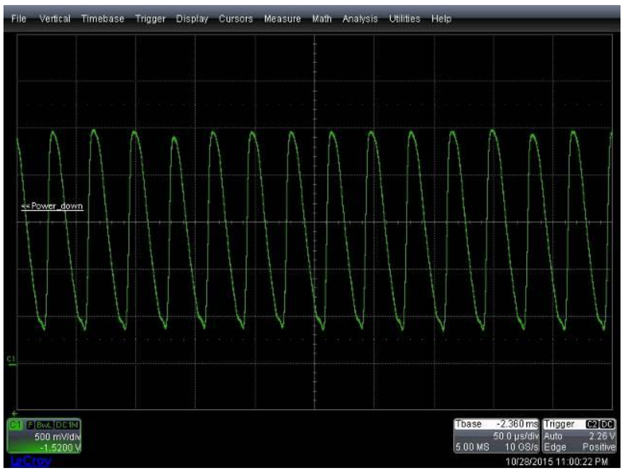


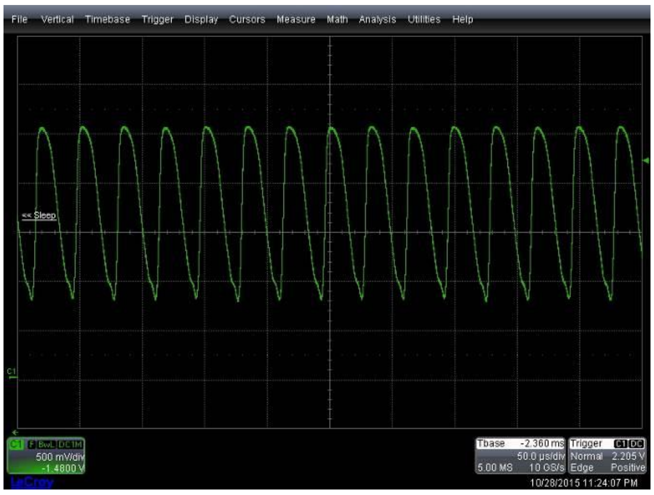
Figure 61: Power down current waveform

Power consumed in *Power-down* is get multiplying the Vcc voltage (2.986V) for the Irms value:

I_{rms} µA	Power µW
2.2	6.56

4.4.2.6.2 Sleep state

When the device is in *Sleep* state the driving circuitry making the moving mass of the gyroscope oscillating is kept active.
The typical current RMS value reported in the datasheet is 1.5 mA.
The Rsense resistance used for the current amplifier circuit is equal to 20 Ohm .
In Figure 62 it is shown the current waveform in *Sleep* mode; as it can be easily seen, the peak value is 2.6 V that in turn corresponds to a current of 2.6 mA .
The rms current value measured by the multi-meter is about 1.49 mA



C1 F BwL DC1M
500 mV/div
-1.4800 V

Tbase -2.360 ms
50.0 µs/div
5.00 MS 10 GS/s

Trigger C1 DC
Normal 2.205 V
Edge Positive

10/28/2015 11:24:07 PM

Figure 62: Sleep current waveform

Power consumed in *Sleep* is get multiplying the Vcc voltage (2.986 V) for the I_{rms} value:

I_{rms} µA	Power mW
1.49	4.44

4.4.2.6.3 FIFO conversion mode: two cases of study

This paragraph will introduce the measurement methodology used to evaluate the energy consumption for two cases of study: FIFO mode at two different ODRs. The table below shows the two groups of settings:

ODR	Range	Precision	Watermark	Sampling mode	Read mode
800 Hz	+/- 2000 deg/s	16 bit	19	Continuous	Auto-increment
100 Hz	+/- 250 deg/s	16 bit	24	Continuous	Auto-increment

Figure 63: The two groups of settings

Total power consumption must be considered as the sum of the single power consumption of each device involved in data transfer: the sensor, the two pull-up resistors of I2C bus and the MCU as shown in the Figure 45.
While sensor contributes to the power consumption both during *Sensing* and *Reading*, pull up resistors and I2C MCU contribute only during *Reading*.
The below figure summarize what above written:

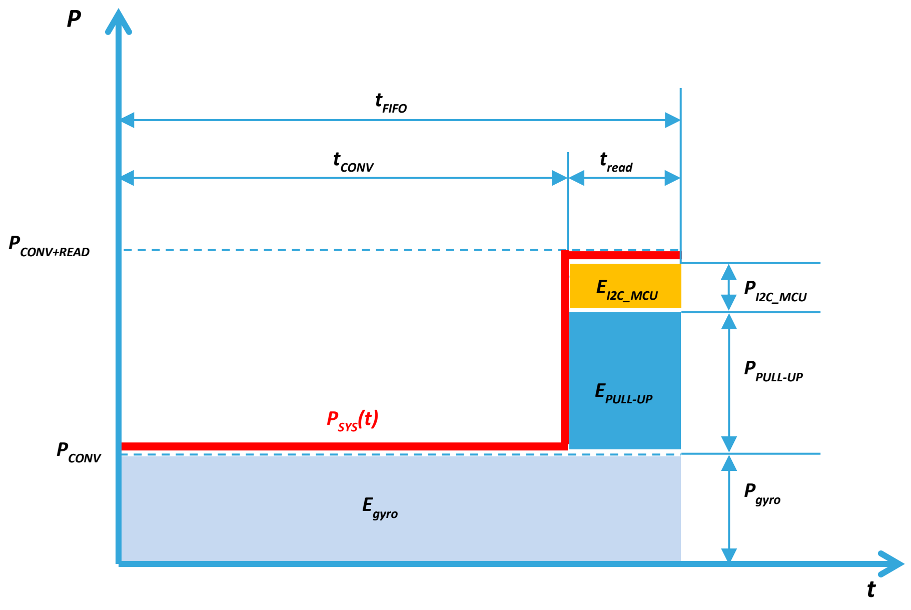


Figure 64: FIFO process SENSING and READING

As done for the accelerometer, the figures below show the oscilloscope waveforms during a complete FIFO process.
The central figure shows the entire FIFO process. For details, please see Figure 52:

- Cyan signal shows the I2C communication during samples transfer;
- Yellow is a monitor generated from MCU to show command sequence;
- Green shows the current that flows through the sensor;
- Purple shows the instant when the watermark signal is set and reset (driven by the sensor)

The main phases are zoomed in pictures 1, 2, 3:

- #1: the starting of FIFO filling process.
- #2: the starting of the reading phase.
- #3: the I2C transmission.

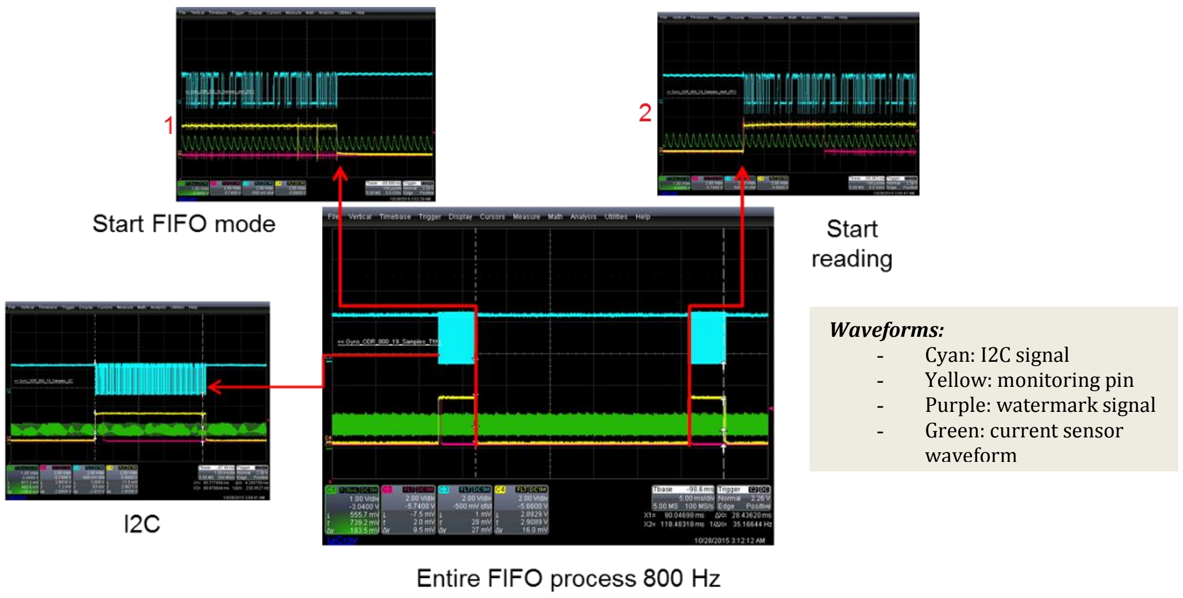


Figure 65: FIFO process waveform

Case1: 3-Axes, 16-bit, FIFO mode, ODR @800, 19 Samples

Total power consumption is computed as sum of single power consumption of each device involved in data conversion and reading: the sensor, the two pull-up resistors of I2C bus and the MCU as shown in the

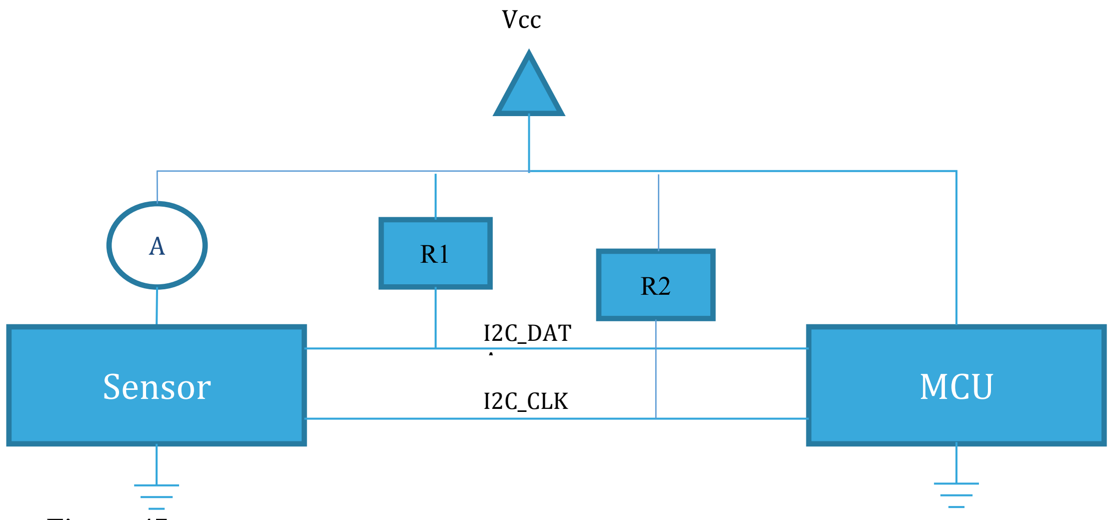


Figure 45.

Compute the power consumption of sensor (Pgyro)

The picture below shows the waveforms generated during a complete FIFO process.
When the established numbers of sample data have been stored into the FIFO, the sensor sets a Watermark signal, at one of its output pin. This signal is reset when the first data set is read by the microcontroller (Figure 66- purple waveform)
A monitoring signal (yellow waveform) is set when the reading phase starts, and is reset at the end (19 samples read).
Cyan signal shows the I2C activity during the reading phase.

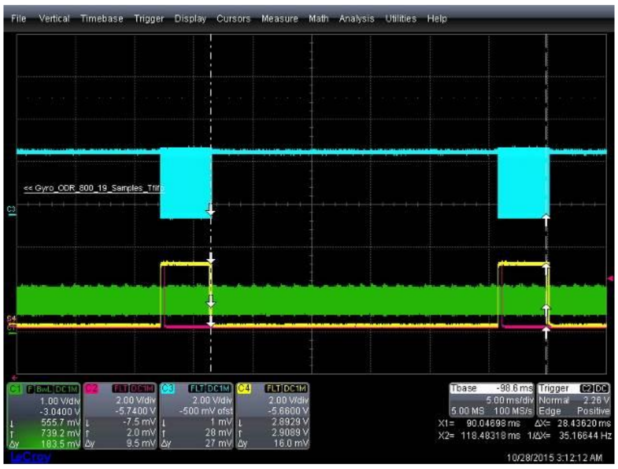

File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
<< Gyro_ODR_800_19_Samples_Tfifo
C1 F BwL DC1M 1.00 V/div -3.0400 V ↓ 555.7 mV ↑ 739.2 mV Δy 183.5 mV
C2 FLT DC1M 2.00 V/div -5.7400 V ↓ -7.5 mV ↑ 2.0 mV Δy 9.5 mV
C3 FLT DC1M 2.00 V/div -500 mV ofst ↓ 1 mV ↑ 28 mV Δy 27 mV
C4 FLT DC1M 2.00 V/div -5.6600 V ↓ 2.8929 V ↑ 2.9089 V Δy 16.0 mV
Tbase -98.6 ms 5.00 ms/div 5.00 MS 100 MS/s
Trigger C2 DC Normal 2.26 V Edge Positive
X1= 90.04698 ms ΔX= 28.43620 ms
X2= 118.48318 ms 1/ΔX= 35.16644 Hz
LeCroy
10/28/2015 3:12:12 AM

Figure 66: ODR 800Hz watermark 19, FIFO filling and reading period

With the parameter settings established above, the duration of complete process (FIFO filling and data reading) lasts (see Figure 66):

$$t_{FIFO} = 28.43ms$$

The I_{gyro_rms} value reading got by the multimeter is:

$$I_{gyro\_rms} = 6.24\, mA$$

With a power supply voltage of

$$V_{dd} = 2.986\, V$$

the consumed power by the sensor is:

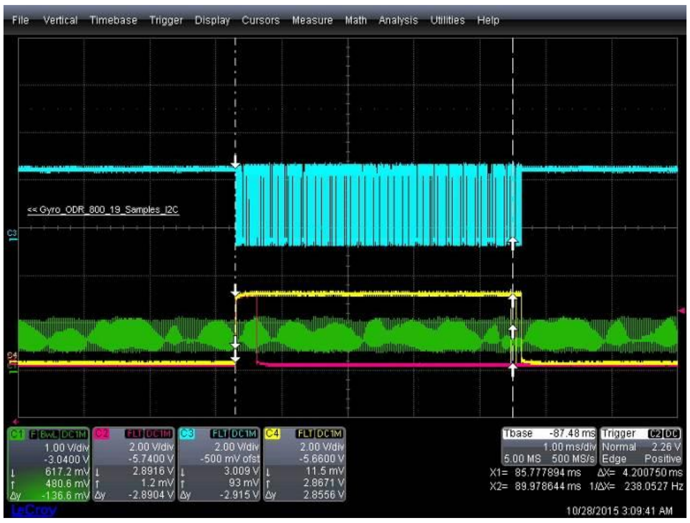
$$P_{gyro} = V_{cc} I_{gyro\_rms} = 2.986 * 6.24 * 10^{-3} \cong 18.63\, mW$$

Energy consumption is obtained multiplying the power consumed P_{gyro} for the duration of the FIFO process.

$$E_{gyro} = P_{gyro} * t_{FIFO} = 18.63 * 10^{-3} * 28.43 * 10^{-3} \cong 529.6\, \mu J$$

P_{R1} and P_{R2} pull-up resistances

During data reading, there will be currents flowing through the two pull-up resistances. This in turns causes a power dissipation that must be accounted in the power consumption evaluation process.
The picture below shows the I2C activity during a samples set reading (19 samples).



```
C1 F BwL DC1M    C2 FLT DC1M      C3 FLT DC1M        C4 FLT DC1M
   1.00 V/div       2.00 V/div       2.00 V/div         2.00 V/div
   -3.0400 V        -5.7400 V        -500 mV ofst       -5.6600 V
↓  617.2 mV      ↓  2.8916 V      ↓  3.009 V         ↓  11.5 mV
↑  480.6 mV      ↑  1.2 mV        ↑  93 mV           ↑  2.8671 V
Δy -136.6 mV     Δy -2.8904 V     Δy -2.915 V        Δy 2.8556 V

Tbase      -87.48 ms   Trigger    C2 DC
           1.00 ms/div Normal     2.26 V
5.00 MS    500 MS/s    Edge       Positive
X1=  85.777894 ms      ΔX=   4.200750 ms
X2=  89.978644 ms      1/ΔX= 238.0527 Hz
10/28/2015 3:09:41 AM
```

Figure 67: Duration of the reading period

From the scope, the duration of read period (for all 19 data sets) is:

$$t_{read} = 4.20ms$$

As seen in the paragraph 0, the current that flows through the pull-up resistances can be approximated to a square waveform with a 50% duty cycle.
Current peak is obtained dividing the V_{CC} for the $R_{PULL\text{-}UP}$ value:

$$I_{PULL\_UP\_peak} = \frac{V_{cc}}{R_{PULL-UP}}$$

The rms value of a square waveform is obtained using the formula below:

$$I_{PULL-UP\_rms} = \frac{I_{PULL\_UP\_peak}}{\sqrt{2}} = \frac{V_{cc}}{R_{PULL-UP} * \sqrt{2}}$$

Power consumption is equal to:

$$P_{PULL\_UP} = V_{cc} * I_{PULL-UP\_rms} = \frac{{V_{cc}}^2}{R_{PULL-UP} * \sqrt{2}}$$

Energy consumed by one of the pull-up resistances during a 19 sample sets reading is equal to:

$$E_{PULL\_UP} = P_{PULL\_UP} * t_{read} = \frac{{V_{cc}}^2}{R_{PULL-UP} * \sqrt{2}} * t_{read} = \frac{(2.986)^2}{4.7 * 10^3 * \sqrt{2}} * 4.20 * 10^{-3} = 5.6\ \mu\text{J}$$

For the two pull-up resistors, then we have:

$$E_{2xPULL\_UP} = 11.2\ \mu J$$

P_{MCU} - I2C peripheral power consumption

The power consumption of the MCU is due to its I2C peripheral operations. Under this hypothesis, the current consumption has been taken from the table in Figure 50: STM32 peripherals power consumption:

$$I_{I2C\_MCU} = 40\ \mu\text{A}$$

Power consumption is equal to:

$$P_{I2C\_MCU} = V_{cc} * I_{I2C\_MCU}$$

Energy consumed by the micro I2C peripheral during the reading of all sample sets is:

$$E_{I2C\_MCU} = P_{I2C\_MCU} * t_{read} = V_{cc} * I_{I2C_{MCU}} * t_{read} = 501\ nJ$$

Summarizing, the energy consumption, for filling and reading FIFO, is the sum of the various energy contributions:

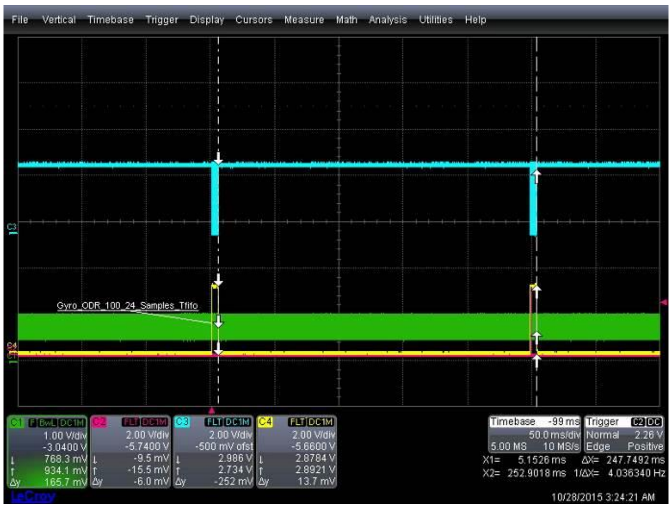
$$\boldsymbol{E_{FIFO\_19} = E_{gyro} + E_{2xPULL\_UP} + E_{I2C\_MCU} = 541.30\ \mu J}$$

Case2: 3-Axes, 16 bit, FIFO mode, ODR @100, 24 Samples

The approach followed to evaluate the global power consumption is the same of the one used in the previous case of study.

Compute the power consumption of sensor (Pgyro)

The picture below shows the waveforms generated during a complete FIFO process.


File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
Gyro_ODR_100_24_Samples_Tfifo
C3
C4
LeCroy

```
C1 F BwL DC1M   C2 FLT DC1M   C3 FLT DC1M      C4 FLT DC1M     Timebase  -99 ms    Trigger  C2 DC
   1.00 V/div      2.00 V/div    2.00 V/div       2.00 V/div             50.0 ms/div Normal   2.26 V
   -3.0400 V       -5.7400 V     -500 mV ofst     -5.6600 V      5.00 MS   10 MS/s   Edge   Positive
↓  768.3 mV     ↓  -9.5 mV    ↓  2.986 V       ↓  2.8784 V    X1=   5.1526 ms   ΔX=  247.7492 ms
↑  934.1 mV     ↑  -15.5 mV   ↑  2.734 V       ↑  2.8921 V    X2= 252.9018 ms  1/ΔX= 4.036340 Hz
Δy 165.7 mV     Δy -6.0 mV    Δy -252 mV       Δy 13.7 mV
                                                                       10/28/2015 3:24:21 AM
```

Figure 68: ODR 100Hz, watermark 24, FIFO mode and reading period

With the parameter settings established above, the duration of complete process (FIFO filling and data reading) lasts (see Figure 66: ODR 800Hz watermark 19, FIFO filling and reading period):

$$t_{FIFO} = 247.74ms$$

The $\boldsymbol{I_{gyro\_rms}}$ value reading got by the multimeter is:

$$I_{gyro\_rms} = 6.22\ mA$$

With a power supply voltage of

$$V_{dd} = 2.986\ V$$

the consumed power by the sensor is:

$$P_{acc} = V_{cc} I_{gyro\_rms} = 2.986 * 6.22 * 10^{-3} \cong 18.57\ mW$$

Energy consumption is obtained multiplying the power consumed P_{gyro} for the duration of the FIFO process.

$$E_{gyro} = P_{gyro} * t_{FIFO} = 18.57 * 10^{-3} * 247.74 * 10^{-3} = 4.6mJ$$

P_{R1} and P_{R2} pull-up resistances

During data reading, there will be currents flowing through the two pull-up resistances. This in turns causes a power dissipation that must be accounted in the power consumption evaluation process.
The picture below shows the I2C activity during a samples set reading (24 samples).

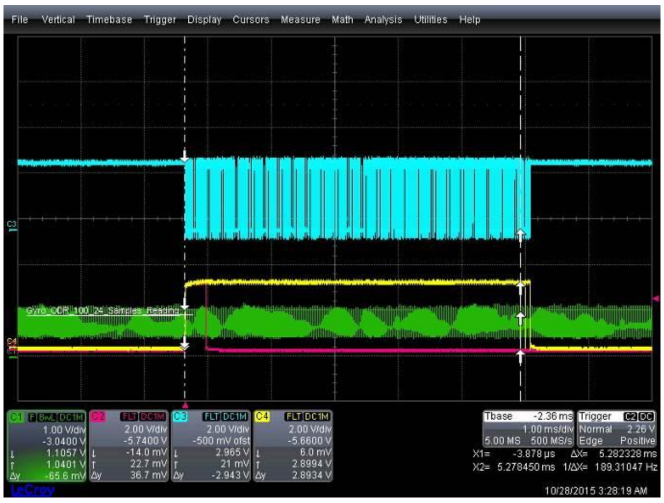

File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
Gyro_ODR_100_24_Samples_Reading
C1 F BwL DC1M 1.00 V/div -3.0400 V 1.1057 V 1.0401 V Δy -65.6 mV
C2 FLT DC1M 2.00 V/div -5.7400 V -14.0 mV 22.7 mV Δy 36.7 mV
C3 FLT DC1M 2.00 V/div -500 mV ofst 2.965 V 21 mV Δy -2.943 V
C4 FLT DC1M 2.00 V/div -5.6600 V 6.0 mV 2.8994 V Δy 2.8934 V
Tbase -2.36 ms 1.00 ms/div 5.00 MS 500 MS/s
Trigger C2 DC Normal 2.26 V Edge Positive
X1= -3.878 µs ΔX= 5.282328 ms
X2= 5.278450 ms 1/ΔX= 189.31047 Hz
LeCroy
10/28/2015 3:28:19 AM

Figure 69: Duration of the reading period

From the scope, the duration of read period (for all 24 data sets) is:

$$t_{read} = 5.28ms$$

As seen in the paragraph 0, the current that flows through the pull-up resistances can be approximated to a square waveform with a 50% duty cycle.
Current peak is obtained dividing the V_{CC} for the R_{PULL-UP} value:

$$I_{PULL\_UP\_peak} = \frac{V_{cc}}{R_{PULL-UP}}$$

The rms value of a square waveform is obtained using the formula below:

$$I_{PULL-UP\_rms} = \frac{I_{PULL\_UP\_peak}}{\sqrt{2}} = \frac{V_{cc}}{R_{PULL-UP} * \sqrt{2}}$$

Power consumption is equal to:

$$P_{PULL\_UP} = V_{cc} * I_{PULL-UP\_rms} = \frac{{V_{cc}}^2}{R_{PULL-UP} * \sqrt{2}}$$

Energy consumed by one of the pull-up resistances during a 24 sample sets reading is equal to:

$$E_{PULL\_UP} = P_{PULL\_UP} * t_{read} = \frac{{V_{cc}}^2}{R_{PULL-UP}*\sqrt{2}} * t_{read} = \frac{(2.986)^2}{4.7*10^3*\sqrt{2}} * 5.28 * 10^{-3} = 7.0\ \mu\text{J}$$

For the two pull-up resistors, then we have:

$$E_{2xPULL\_UP} = 14.0\ \mu J$$

P_{MCU} - I2C peripheral power consumption

The power consumption of the MCU is due to its I2C peripheral operations. Under this hypothesis, the current consumption has been taken from the table in Figure 50:

$$I_{I2C\_MCU} = 40\ \mu\text{A}$$

Power consumption is equal to:

$$P_{I2C\_MCU} = V_{cc} * I_{I2C\_MCU}$$

Energy consumed by the micro I2C peripheral during the reading of all sample sets is:

$$E_{I2C\_MCU} = P_{I2C\_MCU} * t_{read} = V_{cc} * I_{I2C_{MCU}} * t_{read} = 628\ nJ$$

Summarizing, the energy consumption, for filling and reading FIFO, is the sum of the various energy contributions:

$$\boldsymbol{E_{FIFO\_24} = E_{gyro} + E_{2xPULL\_UP} + E_{I2C\_MCU} = 5.29\ mJ}$$

4.4.2.7 Remarks on Gyro measurements

The two diagrams below show the time variation of P_{sys} during the respective time intervals t_{FIFO} and the three energies (E_{gyro}, $E_{PULL-UP}$, E_{I2C_MCU}) contributions to the global energy consumption (E_{FIFO_19} at ODR 800Hz and E_{FIFO_24} at ODR 100 Hz)

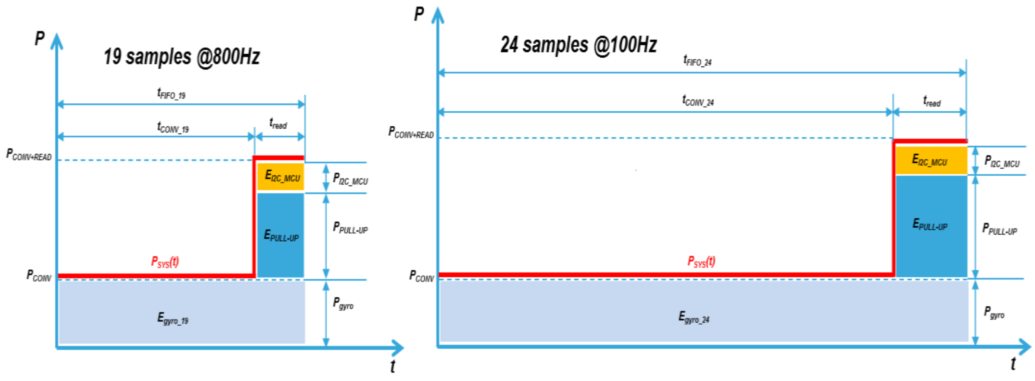


Figure 70: P_{SYS} behaviour 19 samples ODR 400 vs 24 samples ODR 100 Hz

As seen in previous section, the two energies requested for FIFO conversion modes are quite different:

$$E_{FIFO\_19} = E_{gyro} + E_{2xPULL\_UP} + E_{I2C\_MCU} = 541.30\ \mu\mathrm{J}$$

$$E_{FIFO\_24} = E_{gyro} + E_{2xPULL\_UP} + E_{I2C\_MCU} = 5.29\ mJ$$

Looking at the figure above, one can see the two P_{CONV} values are quite similar; this because power consumed by the gyroscope to transduce the external angular velocity into a voltage level does not depend by the selected ODR but it is consumed by the internal sensor driving circuitry for keeping in motion the inertial mass.
On the contrary, the two energies (E_{gyro_19}, E_{gyro_24}) are strictly linked to the ODRs because, time to convert 19 samples at 800 Hz ODR is lower than the one to convert 24 samples at 100 Hz ODR.

$$T_{CONV\_19} \ll T_{CONV\_24}$$

The two contributions E_{I2C_MCU} and $E_{PULL-UP}$ do not depend by ODR; they differ only for the number of sample readings (19 vs 24).
With these considerations, the two energies involved for each conversion mode, are different and the following relationship exists:

$$E_{sys\_19} = \int_0^{t_{FIFO\_19}} P_{sys\_19} dt < \int_0^{t_{FIFO\_24}} P_{sys\_24} dt = E_{sys\_24}$$

For detailed description of communication protocol specifications and frame formats, please refers to UM1744 from STMicrolectronics.

4.4.3 Node level service manager: BBQLite

At the node level one of the main service offered by the CONTREX tools is constituted by the service manager BBQLite, i.e. a module that enable and disables:

- Entire application tasks
- Individual functionality within each task
- Hardware devices

The choices made by the service manager are based on three classes of information:

1. The current **functional status** of the system. The notion of status – also referred to as **operating mode** – of a system is strongly application dependent but is usually defined at a very high-level of abstraction. Typical operating modes of a system are, for example, “Startup”, “Initialization”, “Power-on self-test”, “Normal”, “Stand-by”, “Error”, “Safe-state” and so on. In the specific application of the automotive use-case the states (to be better defined and, possibly, extended) mostly refer to the condition of the vehicle and might be: “Initialization”, “Normal driving”, “Steady, engine on”, “Steady, key on”, “Key off”, “Long-term parking”. This information is either explicitly specified in the application structure (e.g. the system initialization phase is part of the application code) or the result of some analyses performed by the application itself (e.g. detection of vibrations to infer whether the engine is on or off). It is important noting that the operating mode may depend on some extra-functional properties.

2. The current **extra-functional status** of the system, i.e. the status of the system with respect to some of its extra-functional properties. An example of non-functional status information is the level of charge of the battery, the current power consumption of the entire system or of some specific subsystems, the operating frequency, the microprocessor load and so on. Such information can be retrieved through the extra-functional properties abstraction layer proposed by PoliMi within the Task 3.4. activities (and described in more detail in deliverables D3.4.1-D3.4.3) As discussed in the previous point, extra-functional parameters may influence the functional status of the device.

3. The **design-time system characterization** of both hardware and software components, obtained through node-simulations performed with the N2Sim simulator developed by PoliMi in Task 3.1. In turns, as discussed in deliverable D3.1.3, the system characterization is based on models, simulations and measurements on sub-systems and sub-components (e.g. the execution time of a task when executed in a specific system state and with specific input data depends on the flags controlling the execution path of the tasks and on the execution time of the functions called by it).

The functionality of the service manager BBQLite can be summarized by the logic scheme reported in Figure 71, where a clear distinction between the activities performed at design-time (modelling and simulation) and at run-time (application management based on functional and extra-functional data) is clearly shown. The support tool BBQLiteConf takes as input the results of modelling and simulation activities and generates the static infrastructure (summarized into “policies”) used at run-time by the BBQLite service manager.

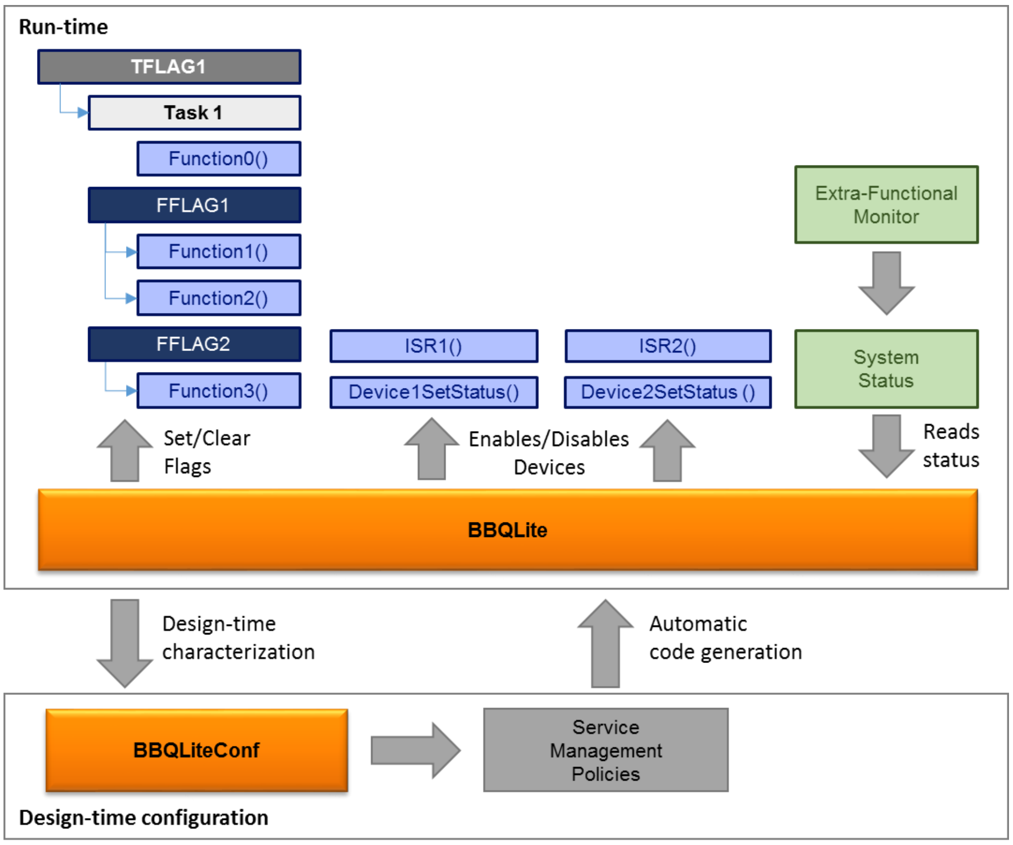


Figure 71: Logic scheme of the node-level run-time management service

To allow for a completely automated flow, some parts of the application shall be modelled in XML. This is necessary to support fully automatic code generation of the entire run-time management infrastructure.
Consider the simple task whose code is shown in Figure 72, and suppose that the first “block” to be managed is constituted by functions 2 and 3, while the second “block” simply coincides with function 4. In addition, suppose that also the entire task can be activated and deactivated under the control of the service manager.

```
void TASK1(void)
{
  while(1)
  {
    // Not to be managed
    Function1();

    // Managed activity 1
    Function2();
    Function3();

    // Managed activity 2
    Function4();
  }
}
```

Figure 72: Task code example

Figure 73 shows how the sample task shall be modelled in XML: as it can be seen, modelling requires very limited human effort when starting from existing code and even less if the task code is being written from scratch. As the figure shows, the model defines a relation between the user-defined flags (TFLAGi for tasks and FFLAGi for functions and/or code blocks.

```
<task managed="true" name="TASK1" flag="TFLAG1">

<block managed="false">
    Function1();
</block>

<block managed="true" flag="FFLAG1">
    Function2();
    Function3();
</block>

<block managed="true" flag="FFLAG2">
    Function1();
</block>

</task>
```

Figure 73: Task model for automatic code generation

In addition to the task specification, an additional input is required to associate the device driver functions needed to enable and disable hardware devices to their flags (DFLAGi). Again, as Figure 74shows, this information can either be provided through code annotations or XML description.

```
<device managed="true"
        name="DEVICE1SetStatus"
        flag="DFLAG1"\>

<device managed="true"
        name="DEVICE2SetStatus"
        flag="DFLAG2"\>
```

Figure 74: Device driver models for automatic code generation

The next conceptual step consists in simulating with N2Sim the generated code, with all the functionally meaningful combinations of the task and function flags. This simulation phase produced both timing/energy reports and traces and allowed the designer to select the suitable combinations of flags to be used in each operating mode of the system. We refer to this set of flags and actions as the "Service Management Policies". Since this phase is subject also to functional constraints, we expect it to be performed manually. Though outside of the goals of the project, the possibility to partly automate this process – for example generating all the "non-functionally feasible" solutions, from which the designer can choose – has been also investigated.

The flag and device status combinations for each operating mode has been then passed to BBQLiteConf to generate the code infrastructure that has been integrated with the application code to implement the service management. From a logical point of view, the input to the configuration tool is a table with the structure shown in Figure 75.

	OpMode1	**OpMode2**	…	**OpModeN**
TFLAG1	0/1	…	…	0/1
...	…			…
TFLAGk	0/1	…	…	0/1
FFLAG1	0/1	…	…	0/1
...	…			…
FFLAGm	0/1	…	…	0/1

DFLAG1	0…n	…	…	0…n
. . .	…			…
DFLAGs	0…n	…	…	0…n

Figure 75: Example of input specification for the BBQLiteConfig configuration tool

The main activities and improvements that have been implemented in the BBQLite run-time engine consists in the integration at design-time with the non-functional metric infrastructure developed in Task 3.1 (and described in D3.1.3), at run-time with the extra-functional metric monitoring infrastructure developed in Task 3.4 (and described in D3.1.4) and finally the interaction with the low-level platform as in the following list:

- A generic interface for managing the microcontroller mode of operation (Normal mode, Sleep mode at different operating frequencies, Stop mode). This interface – of course – requires a specific low-level implementation, since the operating modes and the way they are configured are strongly microcontroller-dependent. The specific implementation for the STM32F103RE, i.e. the microcontroller integrated in the iNemo platform has been implemented.
- The operating mode management functions have been integrated in the actual Vodafone Automotive application. In particular – due to some limitations related to the impossibility to wake-up the core on an external UART event – the microcontroller cannot be put in stop mode. Consequently, the two modes that can be exploited are the following:
    - Normal Mode. The core is executing normally at a frequency 64MHz and the peripherals are all active and operating at their nominal speed.
    - Sleep Mode. The core is in the WFI (Wait For Interrupt) state and is not fetching any instruction from memory. In this state the main clock is reduced to 8MHz and the peripherals, being fed by this clock, are reconfigured consequently. No power gating is possible to further reduce power consumption.

  Integration in the application is rather simple as it only requires to add a few macros in selected code points. Such macros expand to function calls to the BBQLite API. In particular, a call shall be added in the idle operating system task to bring the microcontroller into sleep mode and a call shall be added to each interrupt service routine (UART, SPI, I2C) to bring the microcontroller back to its normal operating mode.
- Integration with a function devoted to retrieve the application operating mode (DRIVING, STOP, PARKING, …) has been performed. Integration with the non-functional metric propagation infrastructure has been completed. This activity allows determining the non-functional state of the device, e.g. current power consumption, average power consumption, battery charge level, communication bandwidth usage, and so on. It is worth noting that the entire infrastructure is automatically generated from an XML description and thus requires little knowledge of the implementation details and a short development time
- Introduction of a non-functional management task that is run periodically. This task determines the functional and non-functional operating modes of the system and application and, based on the policies defined at design time, configures the application and the device in the appropriate mode.

Having completed the simulation and modelling tasks, whose results are collected in Deliverables D3.1.3 and D3.3.3, it has been possible to complete the specification provided as input to the BBQLite run-time manager.

As anticipated in Deliverable D3.2.2, the BBQLite runtime manager provides a rather simple API through which individual software functionality and individual devices can be configured. In particular, a functionality can either be “enabled” or “disabled”, while a device can be put into different power states. Changing the state of a functionality or of a device is done based on a static, compile-time initialized configuration table, resulting from the analysis of the simulation traces described in Deliverable D3.1.3.

For the application at hand, the accelerometer has always been “on”, i.e. in its normal operating mode, and the microcontroller did alternate between its “normal” and “sleep” mode. The table below is the result of simulation analysis.

Functionality	**Operating mode / Battery charge status**			
	KeyOn	**KeyOff BatFull**	**KeyOff BatMed**	**KeyOff BatLow**
Preprocessing	X	X	X	X
Crash Detection	X	X	X	X
Self-Calibration	X			
Low Energy Event Detection		X	X	
Wakeup/Shock	X	X	X	
Driver DNA	X			

It is important noting that the actual operating mode is the cartesian product between the vehicle operating mode (depending on the status of the ignition key) and the current charge status of the backup battery. In particular, the following modes are identified:

1. Ignition key

Status	**Description**
KeyOn	The ignition key is inserted and the power is available. In this state is not relevant whether the vehicle is actually moving or not. Dynamic-based mechanisms (not described here for the sake of conciseness) are running in some of the algorithms to change the algorithm behaviour depending on the dynamic status of the vehicle (vibrations, speed, acceleration, and other derived figures).
KeyOff	The ignition key is off, and thus the vehicle is either steady or being moved by hands by someone. At present it is not necessary to determine wether the vehicle is moving or not.

2. Backup battery charge status

Status	**Description**
BatFull	The battery is charged more than 90% of its capacity
BatMed	The battery is charged more than 20% and less than 90% of its capacity

BatLow	The battery is charged less than 20 % of its capacity

These thresholds are obviously arbitrary, and are the result of a trade-off between the technical requirements and the market needs.

Looking at the table, it is easy to note that the behaviour of the system with the charge status BatMed and BatFull are identical, which shall suggest to merge the two modes into one. They have been left separated keep the flexibility and to simplify further extension with the integration of new functionality envisaged by VOD.

4.4.4 SecSoC application services

With the introduction of low power techniques in the development flow of system on chip, the design has essentially changed from an optimization in two dimensions (performance and area) to optimizing a three dimensional problem, i.e., performance, area and power. The increasing complexity of analysing and optimizing such interacting non-functional properties, in an environment that includes the software application implemented on a hardware platform, can by tackled only at a higher level of abstraction than the RTL.

Electronic System Level (ESL), in the EDA domain, has been conceived to addresses these complexity problems. Resorting to ESL it is possible to propose virtual platform of the system where software and hardware can be simulated together.

4.4.4.1 SecSoC virtual platform

A virtual platform is an environment where the hardware and software components of a design meet for their conjoint modelling, execution and verification. A virtual prototype can be created early in the project flow and used for software development, as well as to assess the system architecture. A virtual platform is the model of system architecture on which the application, including the operating system, can be executed.

A TLM Platform is developed for three main purposes: 1) to provide an early platform for software development: it allows executing the application software on a hardware like system platform which encompasses the models of the peripherals, 2) to allow system level architectural analysis exploration and 3) to provide system verification. Purpose 1) entails functional accuracy, purpose 2 and 3 entail both functional and non-functional requirements.

Our work intends to present an approach were first of all the functional accuracy is addressed, but later one the timing concern is also taken into account, and finally also the power concern is added, allowing to explore the architectural solutions for power management.

The SecSoC development environment is endowed of a Virtual Platform shown in Figure. It encompasses an ISS of the ReISC processor, which executes the software modelled in C language. The rest of the hardware, including the simple bus model and the peripherals is modelled in systemC-TLM. A systemC wrapper implements the interface among the instruction-set simulator (ISS) and the rest of the system.

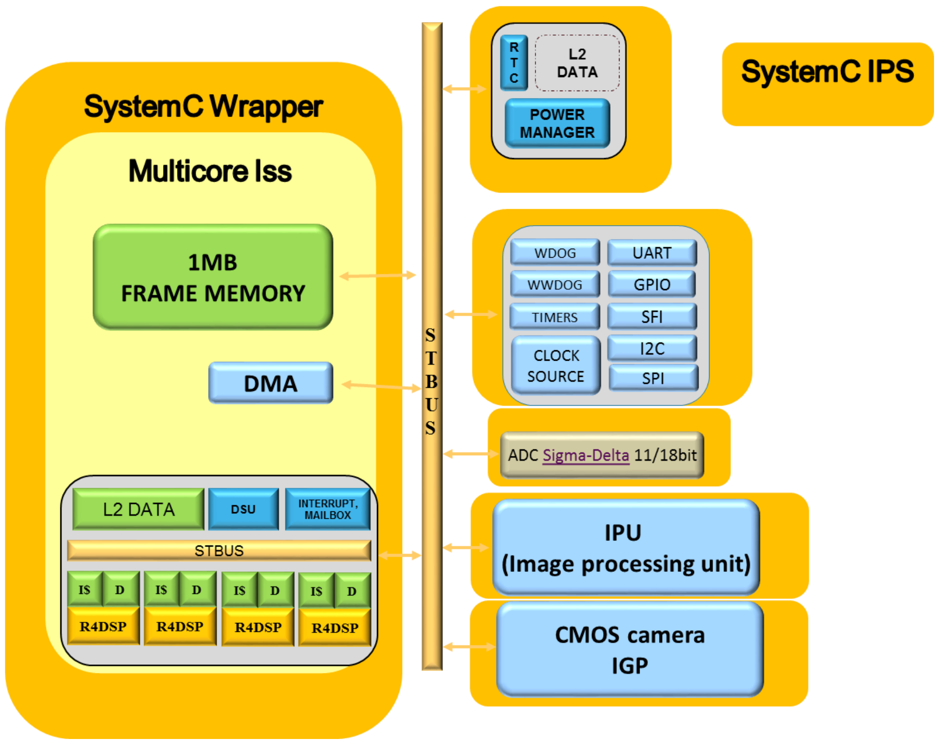


Figure 76: SecSoc Virtual Platform

The virtual platform of SecSoc provides three levels of accuracy:

Functional accuracy

Functionality of the application is modelled by the software application running on the ISS and interacting with rest of the system through communication media and memory. It includes RTOS services that handle tasks and drivers.
The ISS, encapsulated in a systemC wrapper, models the instruction set execution capabilities of the processor as well as its communications on the bus.
The functionality of the hardware parts is modelled and simulated in systemC,

Timing Accuracy

Timing accuracy is mainly modelled by the cycle time accurate nature of the ISS that issues timed transaction to the rest of the system. It is also modelled by the interrupt synchronization mechanism from the system peripherals to the µ-processor.

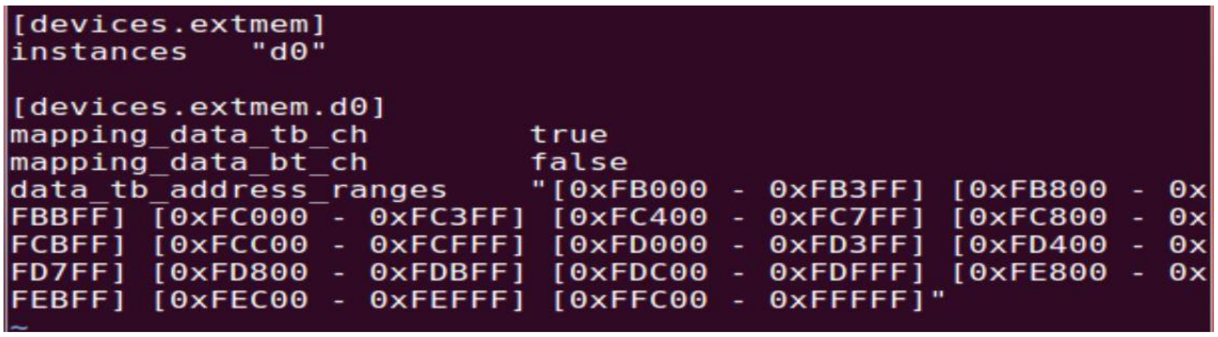
Power Accuracy

The SecSoC TLM virtual platform has been enriched with power information that drives the evaluation of power consumption during the simulation. It is made of a set of power state machines for computing power and energy consumption.
A power state machine is made of power states (indicating levels of power consumption), and edges (corresponding to state transitions activated by events on signals).
These power state machines are dynamically added to the virtual platform at simulation time, there no need of modifying the functional or timing model when the power model changes. A power state machine is usually assigned to a system component to compute its power and energy consumption.

All power state machines execute in concurrently computing power and energy consumption of their associated components.
State transitions of the state machines are triggered by events on driving signals in the design.
Power state machines can be grouped into power islands, in this cases their energy and power consumption are assigned to a power island.
The power virtual platform is the virtual platform empowered by a set of power state machines.

Virtual Platform Configuration

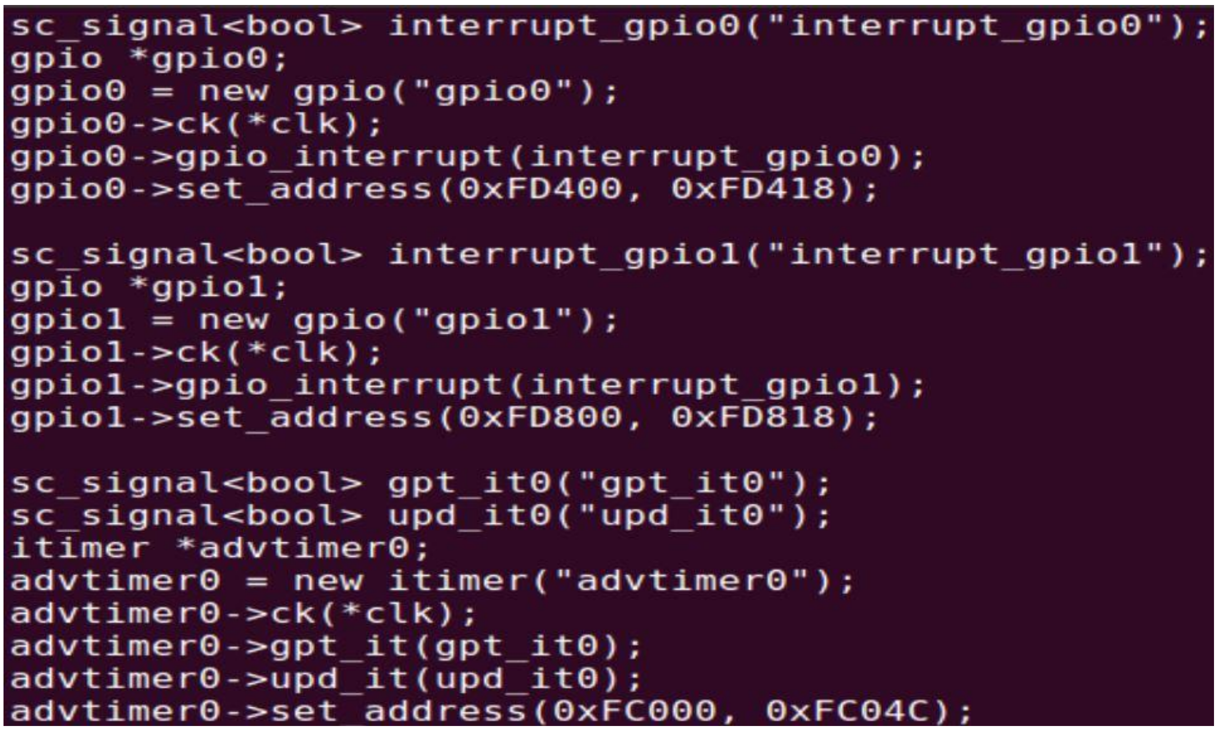
In Figure 76 it is possible to notice that there are two kind of components drawn in orange and in yellow. The components that are closely linked to the cores or to the memory have been left under the direct control of the re-targetable multi-core architecture simulation framework, and they are shown on yellow. The multi-core architecture simulation framework can be considered a multi-core Instruction Set Simulator (since the granularity is the single instruction in the cores) and this simulator must be provided a configuration by a specific file in order to instantiate and configure all the necessary cores/IPs present in the system
The component shown in orange are modelled in systemC, their model must be there in the systemC platform. Configuration of such components in the virtual platform is possible by file where we have to specify their memory mapped registers addresses.
First of all, in the ISS configuration file. we have to specify that the address of the peripherals that are implemented in systemC is mapped externally. Figure 77 shows how components can be mapped to external memory, in this way the ISS translates any transaction with them with a read/write operation to the bus.



```
[devices.extmem]
instances    "d0"

[devices.extmem.d0]
mapping_data_tb_ch        true
mapping_data_bt_ch        false
data_tb_address_ranges    "[0xFB000 - 0xFB3FF] [0xFB800 - 0x
FBBFF] [0xFC000 - 0xFC3FF] [0xFC400 - 0xFC7FF] [0xFC800 - 0x
FCBFF] [0xFCC00 - 0xFCFFF] [0xFD000 - 0xFD3FF] [0xFD400 - 0x
FD7FF] [0xFD800 - 0xFDBFF] [0xFDC00 - 0xFDFFF] [0xFE800 - 0x
FEBFF] [0xFEC00 - 0xFEFFF] [0xFFC00 - 0xFFFFF]"
~
```

Figure 77: mapping components to external memory

Finally, it is necessary to notify to that any of the systemC peripherals has its own memory mapping addresses as well its set of interrupts, that is handled by the ISS file configuration.

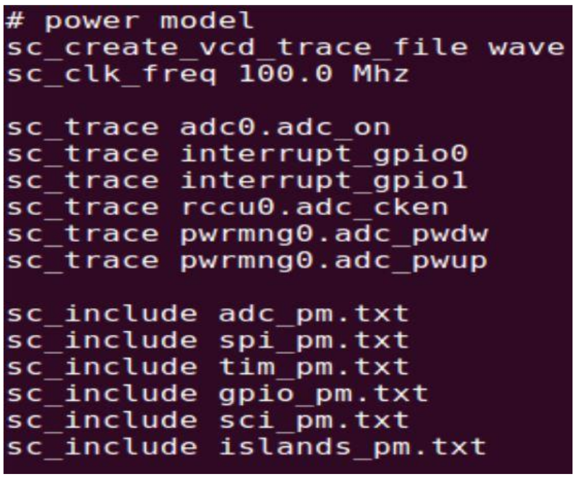


```
sc_signal<bool> interrupt_gpio0("interrupt_gpio0");
gpio *gpio0;
gpio0 = new gpio("gpio0");
gpio0->ck(*clk);
gpio0->gpio_interrupt(interrupt_gpio0);
gpio0->set_address(0xFD400, 0xFD418);

sc_signal<bool> interrupt_gpio1("interrupt_gpio1");
gpio *gpio1;
gpio1 = new gpio("gpio1");
gpio1->ck(*clk);
gpio1->gpio_interrupt(interrupt_gpio1);
gpio1->set_address(0xFD800, 0xFD818);

sc_signal<bool> gpt_it0("gpt_it0");
sc_signal<bool> upd_it0("upd_it0");
itimer *advtimer0;
advtimer0 = new itimer("advtimer0");
advtimer0->ck(*clk);
advtimer0->gpt_it(gpt_it0);
advtimer0->upd_it(upd_it0);
advtimer0->set_address(0xFC000, 0xFC04C);
```

Figure 78: configuration of systemC peripherals

Power Model Configuration

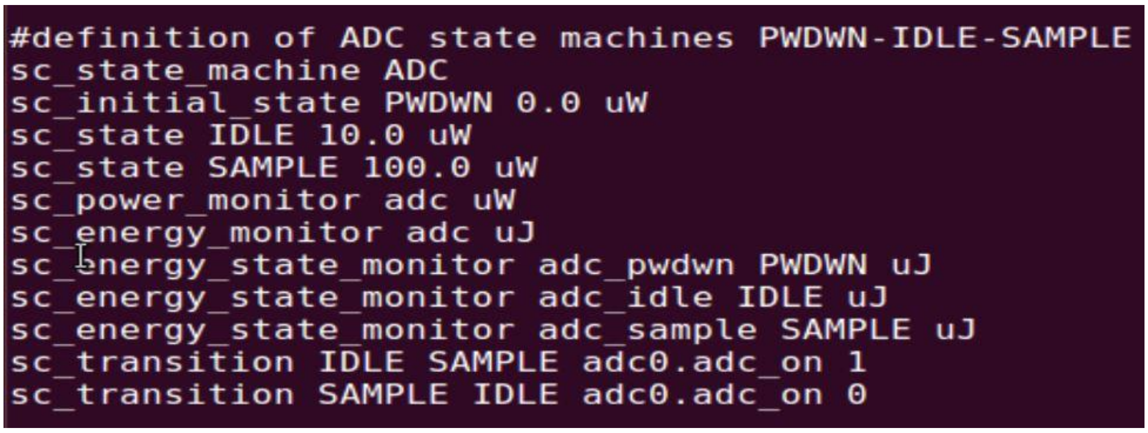
The power model corresponding to the simulation platform can be configured in a textual file: in the following implementation, it is called power_model.txt. It is possibly a hierarchical model; there is in fact the possibility to organize it in more sub-files, each one of them contains the power model of a system component. The power model is described in a proprietary language; it allows to define power state machines for the system components, signals to trace, monitoring variables for power and energy consumption and power islands.
The top power model file is shown in the following Figure 79.
The hierarchical structure of the power model is evidenced by including other files for the system components (sc_include), issuing the presence of other files, one for each component.



```
# power model
sc_create_vcd_trace_file wave
sc_clk_freq 100.0 Mhz

sc_trace adc0.adc_on
sc_trace interrupt_gpio0
sc_trace interrupt_gpio1
sc_trace rccu0.adc_cken
sc_trace pwrmng0.adc_pwdw
sc_trace pwrmng0.adc_pwup

sc_include adc_pm.txt
sc_include spi_pm.txt
sc_include tim_pm.txt
sc_include gpio_pm.txt
sc_include sci_pm.txt
sc_include islands_pm.txt
```

Figure 79: Power model (file powe_model.txt)

An example of power state machines is shown in Figure 80. It specifies a state machine, its states, transitions, triggering signals, level of consumptions and power/energy monitoring variables.

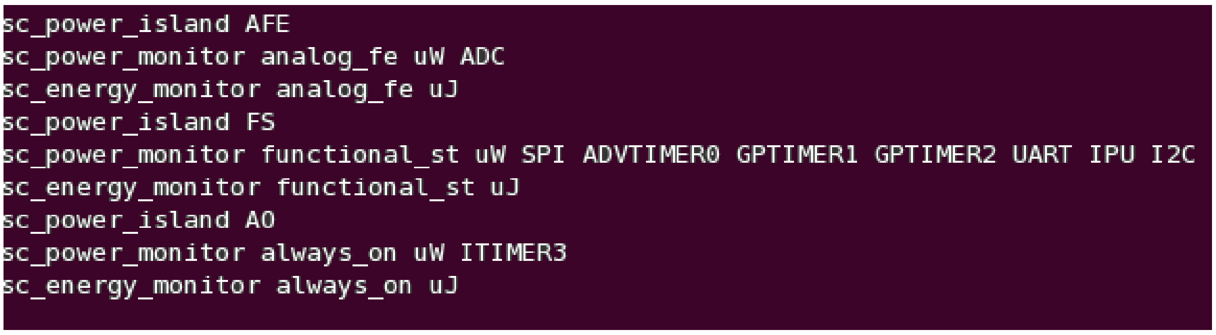


```
#definition of ADC state machines PWDWN-IDLE-SAMPLE
sc_state_machine ADC
sc_initial_state PWDWN 0.0 uW
sc_state IDLE 10.0 uW
sc_state SAMPLE 100.0 uW
sc_power_monitor adc uW
sc_energy_monitor adc uJ
sc_energy_state_monitor adc_pwdwn PWDWN uJ
sc_energy_state_monitor adc_idle IDLE uJ
sc_energy_state_monitor adc_sample SAMPLE uJ
sc_transition IDLE SAMPLE adc0.adc_on 1
sc_transition SAMPLE IDLE adc0.adc_on 0
```

Figure 80: Power state machine Example (file adc_pm.txt)

Power Model Configuration

Configuration of power islands happens by grouping other state machines that define power and energy monitors in one asc_power_island statement as shown in Figure 81. Where there is the definition of three power islands: AFE (contains ADC), FS (contains SPI, ADVTIMER0, GPTIMER1 and GPTIMER2, I2C, UART, IPU), and AO (containing ITIMER3).
The power of the state machines that belong to the island is dynamically added to the power monitor of the island. This figure shows also how power islands are defined by grouping power state machines



```
sc_power_island AFE
sc_power_monitor analog_fe uW ADC
sc_energy_monitor analog_fe uJ
sc_power_island FS
sc_power_monitor functional_st uW SPI ADVTIMER0 GPTIMER1 GPTIMER2 UART IPU I2C
sc_energy_monitor functional_st uJ
sc_power_island AO
sc_power_monitor always_on uW ITIMER3
sc_energy_monitor always_on uJ
```

Figure 81: Definition of Power Islands (file islands_pm.txt)

4.4.4.2 SecSoc SDK (Software development kit)

SeCSoC application can be developed by using the SeCSoC SDK, that includes API, libraries files, and compiler based on gcc.

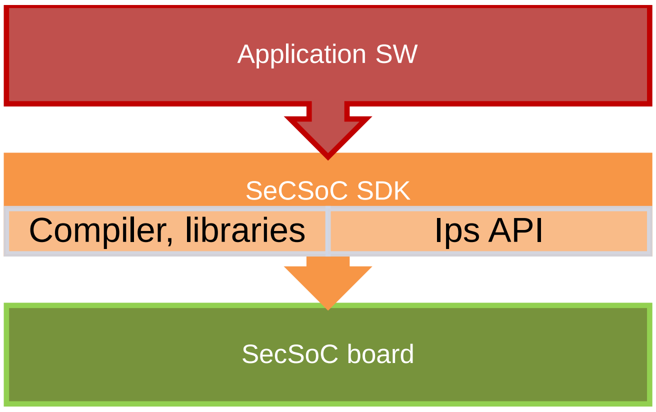


Figure 82: SecSoC SDK (at Contrex M0)

During the CONTREX projects the SecSoC APIs has been enriched of more abstracts API for image processing unit, thanks to the experience acquired in the development of the face detection application in WP4.

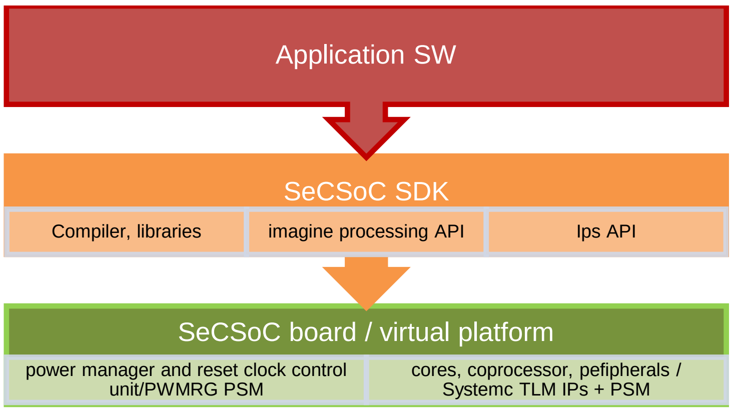

Application SW
SeCSoC SDK
Compiler, libraries
imagine processing API
Ips API
SeCSoC board / virtual platform
power manager and reset clock control unit/PWMRG PSM
cores, coprocessor, pefipherals / Systemc TLM IPs + PSM

Figure 83: SecSoC SDK (at Contrex M36)

```
/* ****** functions for reading and writing pixels ****** */
/**
 * \name Functions for reading and writing pixels
 */
/*@{*/

/**
 * Read a 64-bit word from memory.
 *
 *  \param[in]  engine_cfg  is  the  pointer  to  the  Stream  Interface  Engine
configuration that must be used
 * \param[in] address    is the address where the 64-bit word must be read
 *
 * \return the 64-bit word read from memory
 */

uint64_t  read64(const  ipu_stream_if_engine_cfg_t  *engine_cfg,  uint32_t
address);
 /**
 * Swap the coordinates of two points.
 *
 * \param[in,out] x1 is the pointer to the x coordinate of the first point
 * \param[in,out] y1 is the pointer to the y coordinate of the first point
 * \param[in,out] x2 is the pointer to the x coordinate of the second point
 * \param[in,out] y2 is the pointer to the y coordinate of the second point
 */

void swap_points(int *x1, int *y1, int *x2, int *y2);

/**
 * Draw a horizontal line.
 *
 *  \param[in]  engine_cfg  is  the  pointer  to  a  Stream  Interface  Engine
configuration that contains information on how pixels are formatted
 * \param[in] x1         is the value of the x coordinate of the first pixel
of the line
 * \param[in] y1         is the value of the y coordinate of the first pixel
of the line
 * \param[in] x2         is the value of the x coordinate of the second pixel
of the line
 * \param[in] ch0        is the value of the pixels' channel R/Y
 * \param[in] ch1        is the value of the pixels' channel G/Cb
 * \param[in] ch2        is the value of the pixels' channel B/Cr
 *
 * \retval #true  on success
 *  \retval  #false  otherwise  (the  required  pixel  is  outside  the  frame
boundaries)
 */

void draw_hline(const ipu_stream_if_engine_cfg_t *engine_cfg, int x1, int y1,
int x2, uint8_t ch0, uint8_t ch1, uint8_t ch2);
```

Figure 84: Imaging processing APIs

4.5 Use-case 3: Ethernet Over Radio Systems

The Ethernet Over Radio System application in Use Case 3 makes use of a number of services offered by the underlying platform in order to define and manage the application and its associated extra-functional properties. The principal services are listed in the following.

The services of the Open Virtual Platform (OVP) are used to enable simulation of the application for design space exploration and control and management of the extra-functional properties, in particular timing, power, and related energy.

Layered over the basic OVP is a set of services implemented by OFFIS and EDALab in order to enable the simulation of the specific target environment of the Ethernet Over Radio application (Zynq). The application accesses these services directly rather than the naked OVP platform.

At this level, the KTH-provided mechanisms for Design Space Exploration, used primarily for the determination and manipulation of the extra-functional timing properties of the Use Case 3 application, access the OFFIS-provided simulation environment services over the OVP to obtain feedback on the timing characteristics of different configurations of the application.

OFFIS and EDALab extended the basic set of simulation environment services with an additional extension package of services for obtaining execution traces of the extra-functional power and thermal properties of the application (see description in sections 4.1 and 4.2).

Intel (Docea's team) has provided a tool implementing a set of services that allow the application, using the traces obtained through the OFFIS extended OVP services, to analyse and optimize the thermal and power-related extra-functional properties.

5 Extra-functional-aware runtime management analysis

For CONTREX, a complete **power and thermal modelling and simulation framework** is available to allow design and analysis of runtime management routines at the node level, such as ARM APA described in D3.3.3. The framework is composed of:

- The tools for modelling and estimating power consumption and temperature at the node level, including their stimulation interface, as described in deliverable D3.1.1.
- The monitoring module as described in deliverable D3.4.1, including the access to the probes that model sensors.
- And optionally a runtime management model, including in particular a model of the runtime power and thermal mitigation.

The figure below shows the framework structure and its connection to a runtime execution. The word runtime must be understood here in its extensive meaning: it is the execution of the SW programs, including the resulting activation of the underlying HW components necessary for the execution. HW activation is then the root cause for dynamic power consumption in electronic systems.

In the simulation framework, simulation of the runtime has been done for instance by running the real SW in Instruction Set Simulators (ISS) instantiated on top of a HW virtual model (one that can be created with components written in SystemC and relying on the TLM formalism). The real SW can include the Linux Kernel Scheduler, including itself the runtime management, as implemented in CONTREX UC2. This is the situation shown on the right hand side of the figure. Alternatively, runtime simulation can be achieved using an abstraction of the SW tasks (represented by their computational loads), processed by some virtual Processing Units (PUs) that consume the loads. The PUs are interconnected by SystemC/TLM bus models or even higher-level behavioural models of interconnects. In this case, the user of the framework must additionally create an abstract representation of the runtime management behaviour. This is where the runtime management model steps in. This is the situation shown on the left hand side of the figure.

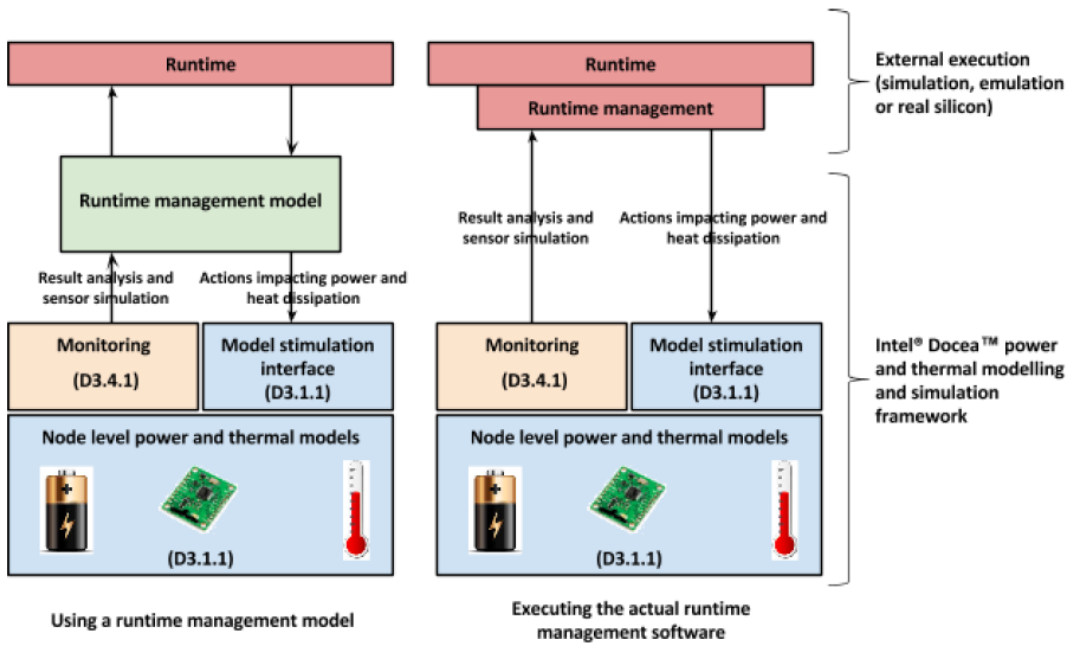
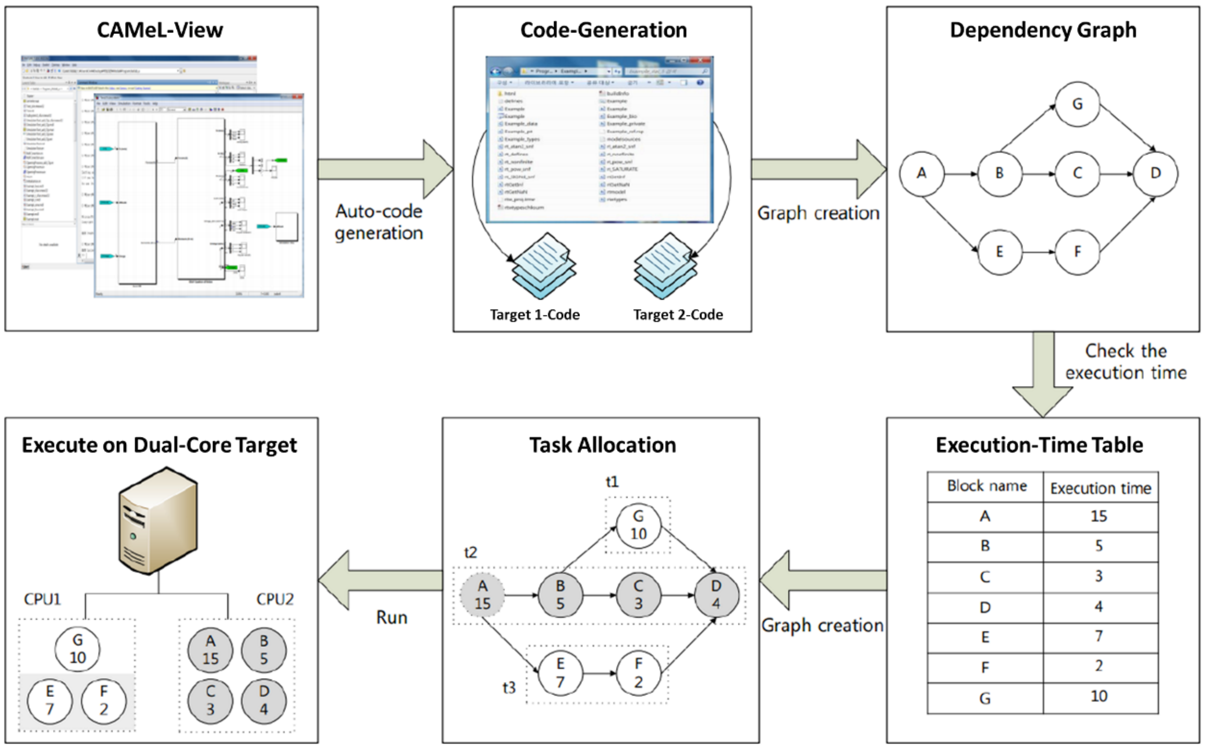

Runtime
Runtime management model
Result analysis and sensor simulation
Actions impacting power and heat dissipation
Monitoring (D3.4.1)
Model stimulation interface (D3.1.1)
Node level power and thermal models
(D3.1.1)
Using a runtime management model
Runtime
Runtime management
Result analysis and sensor simulation
Actions impacting power and heat dissipation
Monitoring (D3.4.1)
Model stimulation interface (D3.1.1)
Node level power and thermal models
(D3.1.1)
Executing the actual runtime management software
External execution (simulation, emulation or real silicon)
Intel® Docea™ power and thermal modelling and simulation framework

Figure 85: Intel® Docea™ power and thermal modelling and simulation framework

A runtime management can receive power (actually current most of the time) and thermal estimates measured by sensors, to make the system able to react at runtime to power consumption and thermal conditions. The runtime management can use the information transmitted by the sensors to adapt the task scheduling (this is adaptation in time) and task mapping (this is adaptation in space) when there is a variety of HW components available to run the processing tasks. Alternatively, the runtime management can send commands to the system's power control to modify the operation conditions (voltage levels, power switches). If the runtime decisions affect the system's Quality of Service (QoS), the impact must be communicated up to the Service Abstraction Layer. In mixed-criticality systems, the runtime management must moreover do its best to preserve the execution of the most critical functions. If this execution is at risk, then this also must be communication to the Service Abstraction Layer. Intel® Docea™ modelling and simulation framework available in CONTREX enables the analysis of these mechanisms.

6 Code optimisation algorithms

The task in WP 3.2 for iXtronics was to provide a code optimisation algorithm for efficient execution of ECU codes under hard real time conditions in CAMeL-View and on real time targets. The actual implemented code generation is limited to single core processor systems and is a 1:1 transformation of the original equations given by the user.



Block name	Execution time
A	15
B	5
C	3
D	4
E	7
F	2
G	10

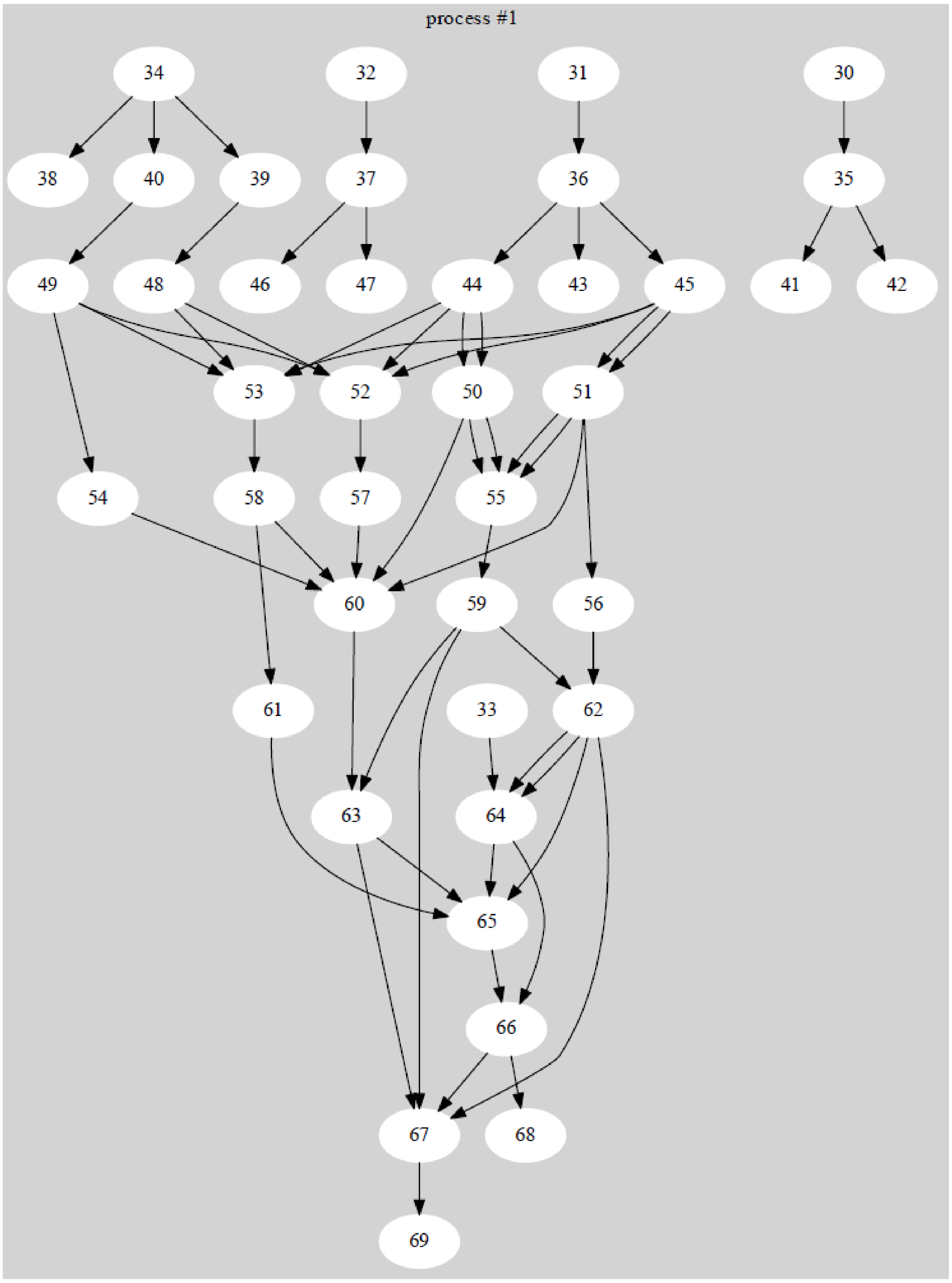
To enhance the code generation towards a faster and more reliable execution under hard real-time conditions 2 enhancement techniques were tested and prototypically implemented:

- A performance enhancement algorithm using computer algebra methods. The user configures the variables (parameters, inputs, outputs and state variables) which are needed for monitoring and control of the application. Then the algorithm simplifies the code by inserting and eliminating not needed variables and pre-calculation of static calculations.

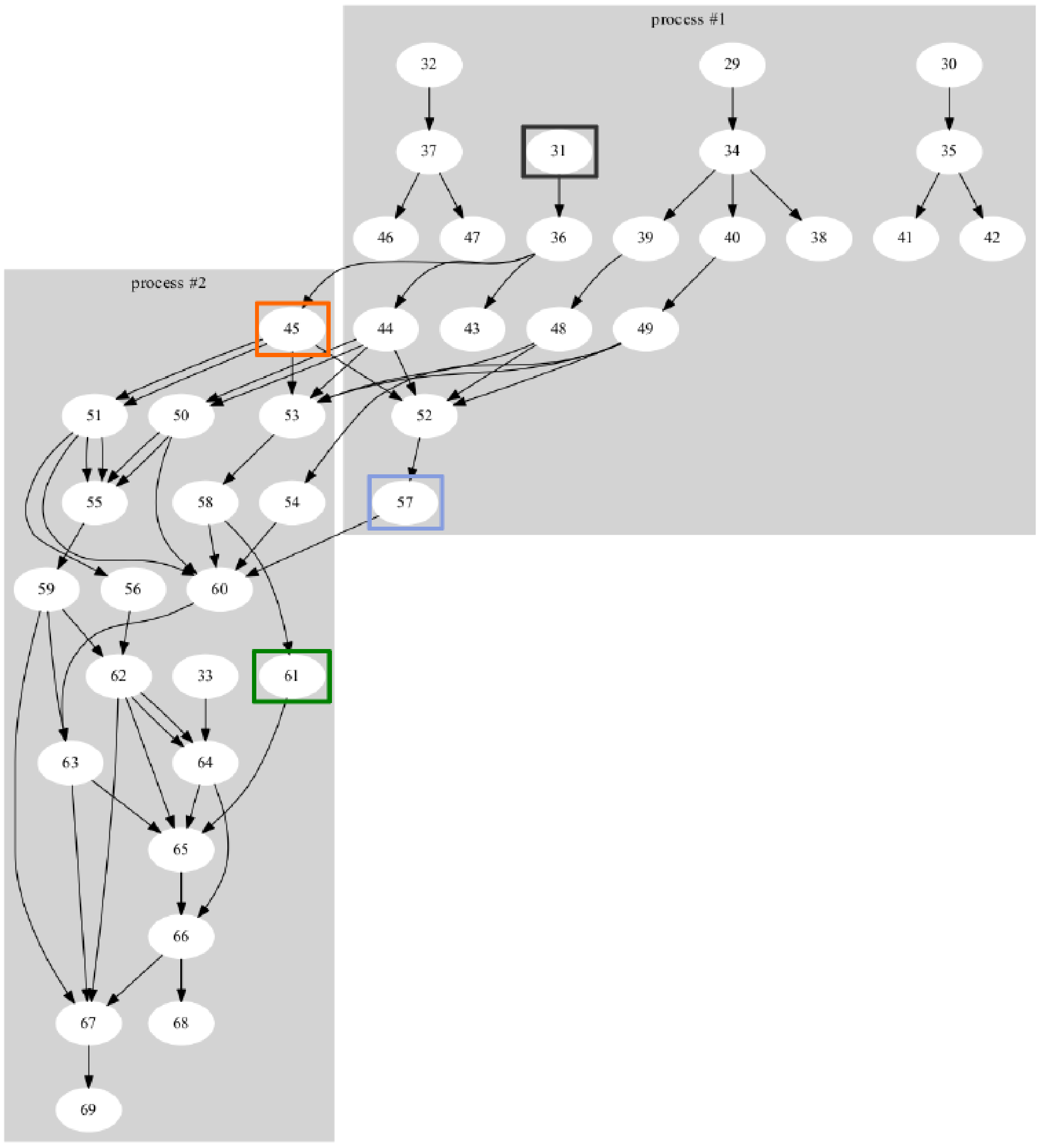
- Parallelisation Algorithms to execute the code on symmetric multiprocessing platforms like the ZYNQ 7000. For that purpose, different parallelisation algorithms were explored and the suitability for the CONTREX field of applications was checked. After the conversion of the equations of the original model into a data-flow graph different partition algorithms were applicated and the results were compared.

The combination of the two techniques allow a significant acceleration of the execution on a double core system like the ZYNQ 7000. For this double-core system it leads to an acceleration of the code by 70%.

Example of a data flow graph for a single processor execution without applied optimization algorithms:



Example of a partitioned data flow graph for a dual core processor execution with applied optimization algorithms:



7 Conclusions

This deliverable presented the final implementation of the general layer responsible for the abstraction of the functional, extra-functional and monitoring features of the execution platform and cloud services. Regarding the execution platform, the deliverable described the final version of the hardware prototypes of the embedded devices composing the platform. Regarding the cloud services, the deliverable provided a description of the final versions of all the software components of the device-to-cloud approach, including the cloud abstraction layer, the Kura pervasive framework and of the cloud integration platform. Finally, the deliverable explained how the execution platform and the abstraction layers are adopted in the project use cases and, for each use case, it provided a description of the final implementation of the use case specific services.

8 References

[1] Xilinx Zynq-7000 All Programmable SoC, http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/

[2] Trenz Electronic – TE0720 Series, http://www.trenz-electronic.de/de/produkte/fpga-boards/trenz-electronic-te0720-zynq.html

[3] The Barbeque Open Source Project – a highly modular and extensible run-time resource manager, http://bosp.dei.polimi.it

[4] Mike Muller, "Power constraints: from sensors to servers", keynote in Hot Chips - Symposium on High Performance Chips, 2014.

[5] Deliverable D3.1.2: Extra-functional property models (preliminary)

[6] Open Virtual Platforms – the source of Fast Processor Models & Platforms, Website: http://www.ovpworld.org/, Last visited: Sept. 24th 2015

[7] Filippo Cucchetto, Alessandro Lonardi, Graziano Pravadelli, A common architecture for co-simulation of SystemC models in QEMU and OVP virtual platforms in Proc. of IFIP/IEEE International Conference on Very Large Scale Integration , Proc. of IFIP/IEEE International Conference on Very Large Scale Integration, Playa del Carmen, Mexico, 6-8 October, 2014, pp. 67-72.

[8] M. Monton, J. Carrabina, and M. Burton, "Mixed simulation kernels for high performance virtual platforms," in ECSI FDL, 2009, pp. 1–6.

[9] M. Monton, J. Engblom, and M. Burton, "Checkpointing for virtual platforms and SystemC-TLM," IEEE TVLSI, vol. 21, no. 1, pp. 133-141, 2013.

[10] EUTH Reliagate 10-11: http://www.eurotech.com/en/products/ReliaGATE 10-11

[11] EUTH Reliacell: http://www.eurotech.com/en/products/ReliaCELL 2010-20

[12] EUTH ESF: http://www.eurotech.com/en/products/software+services/everyware+software+framework

[13] OSGi web site: http://www.osgi.org