EMBEDDED SYSTEMS PROJECT

v1.1

Date: 11/02/2023

Authors:

Thomas MAYNADIE

Marcin KOVALEVSKIJ

Table of contents

Versioning

Date	Version	Description
11 févr. 2023	v1.0	Creation of the document
19 févr. 2023	v1.1	First draft of the Software Design section done.

Reference documents

This document refers to the following applicable documents.

Reference	Document	Description
R1		Project description document
R2	https://www.star-dundee.com/wp-content/star_uploads/general/SpaceWire-Users-Guide.pdf	Space wire user guide
R3	http://spacewire.esa.int/content/Standard/documents/SpaceWire%20RMAP%20Protocol%20Draft%20F%204th%20Dec%202006.pdf	RMAP normative
R4	https://www.microchip.com/content/dam/mchp/documents/OTH/ProductDocuments/Brochures/41005C-AEROSpaceRadHardProcessor_E_US_050615_web.pdf	
R5	https://www.microsemi.com/document-portal/doc_download/130713-rtax-s-sl-and-rtax-dsp-radiation-tolerant-fpgas-datasheet	RTAX-S/SL datasheet
R6	https://www.digikey.com/htmldatasheets/production/1027442/0/0/1/ad7621.pdf	ADC datasheet
R7	https://www.3d-plus.com/data/doc/products/references/3dfp_0605_4.pdf	512 MB SDRAM datasheet
R8	https://www.3d-plus.com/data/doc/products/references/3dfp_0688_4_.pdf	10 MB MRAM

Glossary

This document uses the following acronyms.

Name	Description
ADC	Analog to digital converter
CAN	Controller Area Network
CCSDS	Consultative Committee for Space Data Systems
DPRAM	Dual Ported Random Access Memory
ECSS	European Cooperation for Space Standardization
EDAC	Error Detection And Correction
EEPROM	Electrically Erasable Programmable Read Only Memory
ELG	Equivalent Logic Gate
EOP	End Of Packet
FDIR	Fault Detection, Isolation, and Recovery
FEE	Front End Electronics
FPGA	Field Programmable Gate Array
HK	Housekeeping
IFS	Instrument Flight Software
IP	Intellectual Property
LUT	Look Up Table
LVDS	Low Voltage Differential Signaling
MRAM	Magnetic Random Access Memory
OBC	On-Board Computer
PROM	Programmable Read Only Memory
R/W	Read/Write
RAM	Random Access Memory
RMAP	Remote Memory Access Protocol
S/C	SpaceCraft
SDRAM	Synchronous Dynamic Random Access Memory
SpW	Space wire
TC	TeleCommands
TM	TeleMetry

About the project

As part of phase A of a space instrument, this report summarizes the feasibility and design study of the digital processing subsystem of a space instrument made of 50 linear sensors. Each linear sensor contains 100 pixels (column) and is associated with a Front End Electronics subsystem (FEE) which is responsible for the 16 bit analog-to-digital data conversion and data transfer to the instrument Digital Processing Unit.

The data transferred from the instrument to the DPU through a SpaceWire link using the RMAP protocol. The SpW link offers a throughput of 50 Mb/s. The timing of the instrument is performed by an external clock. A single acquisition is performed every 100 ms, the integration time of one sensor is 10 ms.

System specifications

Numbering	Specification	Description
S1	Processor : LEON @ 50 MHz	CPU is a LEON family processor running @ 50 MHz.
S2	Data is transferred using SpW link @ 50 Mb/s.	The chosen SpW throughput is 50 Megabits per second.

Instrument specifications

Numbering	Specification	Description
I1	$N_{SENSORS} = 50$	The instrument is composed of 50 linear sensors
I2	$N_{PXL\_PER\_SENSOR} = 100$	Number of pixels per sensor.
I3	$N_{ADC\_RES} = 16$ bits	Resolution of the FEE ADC is 16 bits.
I4	$T_{INT} = 10$ ms	Single sensor integration time is 10 milliseconds.
I5	$T_{ACQ} = 100$ ms	Single sensor acquisition corresponds to 10 milliseconds.

Hardware subsystem

In the following sections we develop hardware requirements by considering System specifications and Instrument specifications.

SpaceWire architecture

For the SpW architecture we choose a simple point-to-point link between the instrument and DPU board. This arrangement is represented on the figure below. We justify our choice because we have only one source node (instrument) and one destination node (dpu/memory). Advantages of point-to-point links are: simplicity and low power per Mbit/s. We do not use a router based architecture because we have only one SpW source and destination node. As a consequence, the point-to-point link design is simple and requires less power than router based architecture.

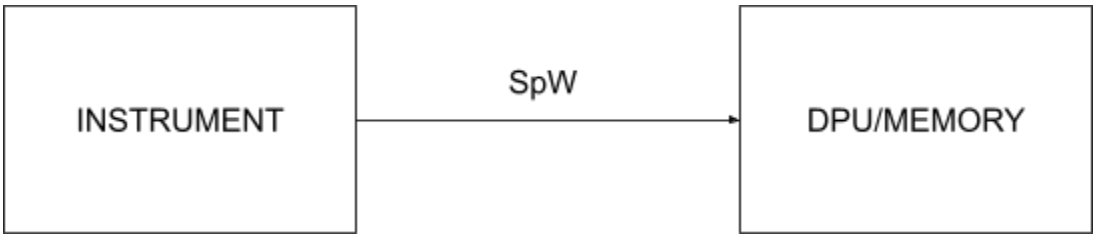

Figure 1.0: SpaceWire interface between the DPU and the Instrument

The SpaceWire interface will be used for both Data (TM), Housekeeping (HK) and Command (TC) transmissions between the DPU and Instrument boards. As described in the following sections, a second connection from the DPU Board to the S/C has been added to communicate with the OBC (TM/TC).

SpaceWire RMAP

As specified by the instrument's high level requirements, RMAP protocol will be used for SpW communication between the Instrument board and the DPU. While the primary purpose of RMAP is to configure SpW networks, it can also be used to gather and send data between multiple SpW nodes.

The RMAP *Write* command is used to transfer data between two SpW nodes. It allows the source node to write multiple bytes of data into the memory of a destination node (given that the latter is compatible with SpW RMAP). To obtain a more detailed description of the RMAP protocol please refer to the [R3] in the Reference documents section.

The *Write* command format is summarized in the table below. From the former, we infer that the RMAP format consists of 8 bits-long atomic blocks (however, some format encodings can be

written using 32 bits or 40 bits, which is not shown on the table below). In summary, an additional 140 bits per packet are required by the RMAP protocol ($N_{PACKET\_BITS} = 140$). Note that in addition to that, RMAP also requires a total of 10 effective bits to transfer 8 raw data bits, corresponding to a data encoding factor $\alpha_{ENCODING} = 10/8 = 1.25$.

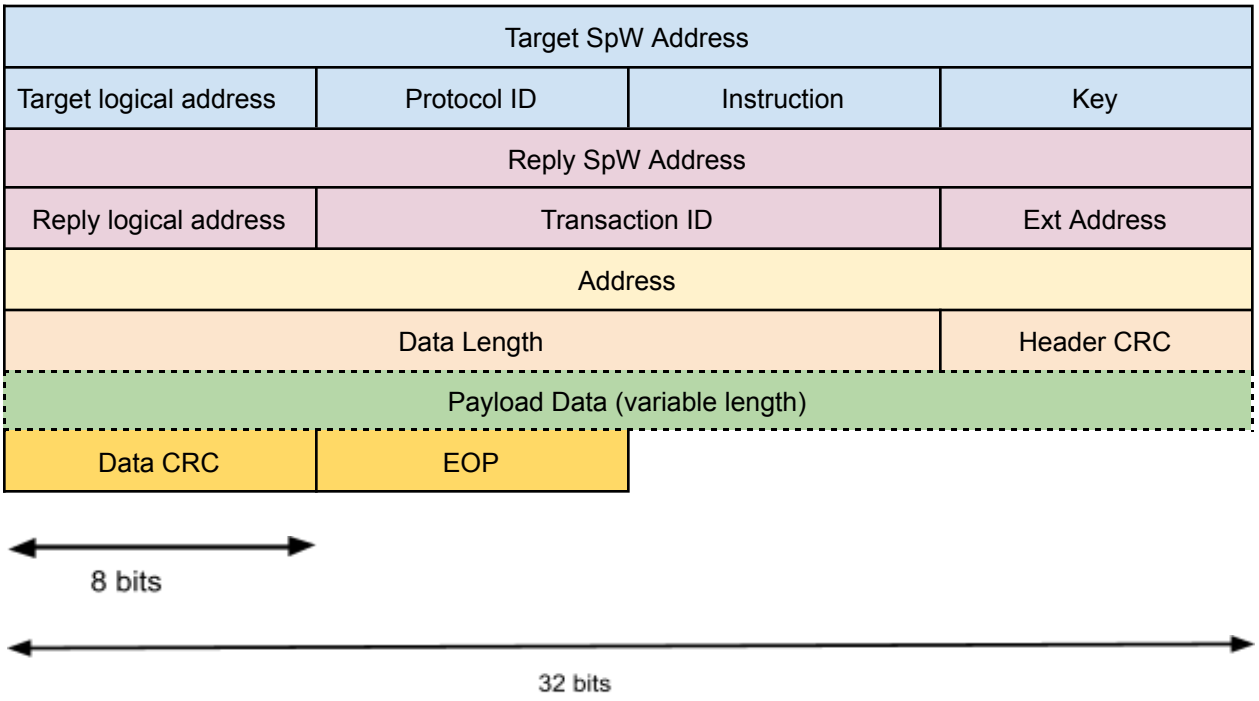

Figure 2.0: Structure of a RMAP *Write* packet.

Memory architecture

DPRAM

Instrument specifications and System specifications state that the data transfer from the instrument board to the DPU board should be established through a buffer memory block. To comply with this requirement, a DPRAM memory has been chosen to serve as an interface between the Instrument and the CPU memory bus. We calculate the memory size requirements for one acquisition as follows :

$$N_{ACQ\_BITS} = N_{SENSORS} \times N_{PXL\_PER\_SENSOR} \times N_{ADC\_RES} = 50 \times 100 \times 16\,b = 80\,kb = 10\,kB.$$

Rounding the memory N_{MEM_SIZE} to the next closest available memory standard, we get $N_{DPRAM\_SIZE\_REQ} = 16\,kB$. The extra memory can for example be used for the storage of HK data (Temperature, Voltages, …).

SDRAM

As the current project is still in the Phase A of the development, we have no SDRAM specifications. Since the main mission objective is a scientific study, a maximum acquisition time of $T_{TOTAL\_ACQ} = 1\,h$ has been chosen before transfering the measurement data to the high capacity storage of the S/C OBC. The corresponding total number of acquisition can be expressed as $N_{ACQ} = T_{TOTAL\_ACQ}/T_{ACQ} = 36 \times 10^3$, from which we can deduce the SDRAM size requirement $N_{SDRAM\_SIZE\_REQ} = N_{ACQ} \times 1B = 36\,kB$ (1 byte is enough to store 100 hotspots, please refer to the Software subsystem for a detailed description about the hotspots computation algorithm). We round the memory $N_{SDRAM\_SIZE\_REQ}$ to the next closest available memory standard of 64 kB, thus $N_{SDRAM\_SIZE\_REQ} = 64\,kB$. At the current mission phase, we don't specify a specific mission component for the SDRAM.

MRAM

We have chosen to add a 10 MB 3D PLUS radiation tolerant MRAM memory (3DMR10M40VS5688). The main purpose of this memory in our application is to store both the Instrument Flight Software (IFS) and 2x FPGA configuration memory. We choose the MRAM memory instead of the typically used EEPROM memory as the former offers greater protection against Single Event Upsets (SEU). For more information about the selected component, pleas refer to component datasheets [R8] in the Reference documents section.

PROM

For our application, we use PROM to store the DPU Bootloader software. From the previous state of the art space based missions we estimate the required PROM size of $N_{PROM\_SIZE\_REQ} = 128\,kB$. At this mission phase, we don't specify specific PROM components.

Instrument architecture

The instrument architecture is imposed by the system specifications. Mainly, the instrument is composed of the set of $N_{SENSORS}$ and every sensor comprises $N_{PXL\_PER\_SENSOR}$. Each sensor is associated with the FEE which is described below.

FEE

Among others, the FEE is responsible for analog-to-digital conversion and data handling between instrument and DPU modules. We have chosen an AD7621ACPZ ADC which is developed by Analog Devices Inc.. Among others, the AD7621ACPZ ADC features 16 bit output denoted here as N_{ADC_RES} at 3 MHz rate denoted as f_{ADC}. For the ADC datasheet please refer to the [R6] in the Reference documents section. The simplified architecture of the FEE is described in the following Figure :

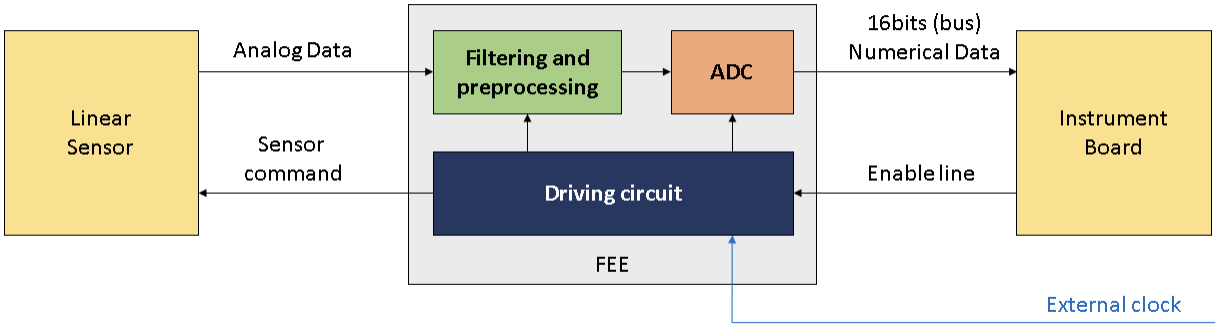

Figure 3.0: Simplified architecture of the FEE subsystem.

The FEE can be described by an equivalent component which performs both the driving of the sensor circuit and the preprocessing/conversion of raw filter data. After studying the available solutions, the selected ADC architecture consists of one Analog Data input, a 16bits parallel output port which is used to transmit the converted number and one Enable port which enables the transmitting of the stored result of the ADC on the output port (see section Hardware block diagram).

When the enable line is activated by the Instrument board, the Driving Circuit issues a set of command signals to the Linear Sensor in order to drive the acquisition of the next pixel. Once the signal of the corresponding pixel is available, the analog signal produced by the sensor is acquired by the FEE, filtered, pre-processed and converted to a 16bits binary word. When the conversion is done, the ADC notifies the driving circuit, which then enables the output of the ADC to be transmitted on the 16bits output bus towards the Instrument board. The acquisition of the 100 pixels of a single sensor is therefore performed by sending a hundred *Enable* pulses to the FEE, which will in return send a hundred 16bits words back to the instrument 16bits numerical data bus. The architecture of the bus will be specified in more details in the following sections.

Processor

For the processing needs we have chosen a LEON-REX family CPU operating at 50 MHz. We justify the choice of this CPU because of its operating frequency of 50 MHz and the availability of two SpW ports which can be used to communicate with the Platform OBC. We summarize other characteristics of this CPU in the table below.

AT7913E Characteristics	
Description	32-bit SPARC V8 SpaceWire Remote Terminal Controller
Performance	40MIPS @ SYSCLK = 50 MHz
Consumption	Typ. 1W 25mW/MIPS
Operating Voltage	1.65V to 1.95V (Logic) 3V to 3.6V (Buffers)
TID; Latch-up	>300 Krad ; 80 MeV/mg/cm2 @125°C
SpW ports	2

Table 1.0: Characteristics of the selected LEON CPU.

Despite their rather medium performances, LEON-REX CPUs offer a unique reliability in the presence of radiations, which has led to their extensive use in the space industry for decades. Due to the absence of requirements regarding the reliability of the system, it has been chosen to use a single CPU instead of using multiple redundant CPUs (which drastically increase cost and development time). However, given the modular architecture of the instrument, it is feasible to use two completely redundant DPU boards for critical applications.

FPGA

During mission phase A we have chosen to include two FPGAs which can be seen on the Hardware block diagram. The FPGA of our choice is the RTAX250s. We have chosen this solution as the RTAX family FPGAs are radiation hardened. Another metric of our choice is the number of equivalent logic gates (250 000) of the RTAX250s FPGAs. One FPGA is used per board.

DPU FPGA

The main task of the FPGA located on the DPU board is to handle communication between the DPU and Instrument modules. Specifically, it acts as an intermediary link between the instrument board and the DPRAM memory for the data transfer via the SpW bus (RMAP). Also, it sends acquisition related commands to the instrument module. In the table below we provide

a first estimate of the number of Equivalent Logic Gates (ELG) required to implement the main functions (IP cores) of the DPU FPGA as dictated per mission requirements.

IP Core/Subsystem	Eq. Logic Gates	Count	Total
SpaceWire Interface	8000	2	16000
RAM Controller	3200	1	3200
CAN Controller	9000	1	9000
RMAP Controller	215	2	429
Total			**28629**
Total (50% margin)			**42944**

Table 2.0: Estimation of the required ELG (DPU FPGA).

The number of ELGs has been calculated using the implementation performances of each IP core. The resources necessary to implement each core is given by the manufacturer as a number of ELGs for ASICs or of Look Up Tables (LUTs) for FPGAs. The conversion from LUT to ELG (which has been chosen as the main metric for our application) assumes that 8 ELG are necessary for 1 LUT, which is a rather conservative estimate as our documentary research showed that 1 LUT can often be obtained using 4 to 8 logic gates.

INST FPGA

The main task of the FPGA located on the Instrument (INST) board is to select a specific sensor by interfacing with a specific FEE. The selection of a specific sensor is done by using a MUX (50x1 multiplexer implemented in the FPGA) to select a given sensor according to the digital clock. The selection of a given FEE is done by enabling the corresponding enable line as shown on the Hardware block diagram. Once a specific FEE is selected, the INST FPGA acquires the data from a given sensor by transferring it via a common 16 bit bus (the 16 bit bus was chosen as a result of the 16 bit ADC requirement). We note that we use a single 16 line bus to transfer the data from a given sensor to the INST FPGA. The data acquisition of the $N_{SENSORS}$' conversion and full transfer from the INST board to the DPU board is done in the 90 ms time constraint.

IP Core/Subsystem	Number of logic gates	Count	Total
SpaceWire Interface	8000	2	16000
Demultiplexer (1:4)	6	21	126
Parallel to Serial Interface (1 bit)	9	32	288
CAN Controller	9000	1	9000
RMAP Controller	215	2	429
Total			**25843**
Total (50% margin)			**38765**

Table 3.0: Estimation of the required ELG (INST FPGA).

Hardware block diagram

As per system specifications, the hardware architecture is separated into two subsystems: the instrument board (INST) and the DPU board. The main hardware architecture choice rationale is described in the sections above.

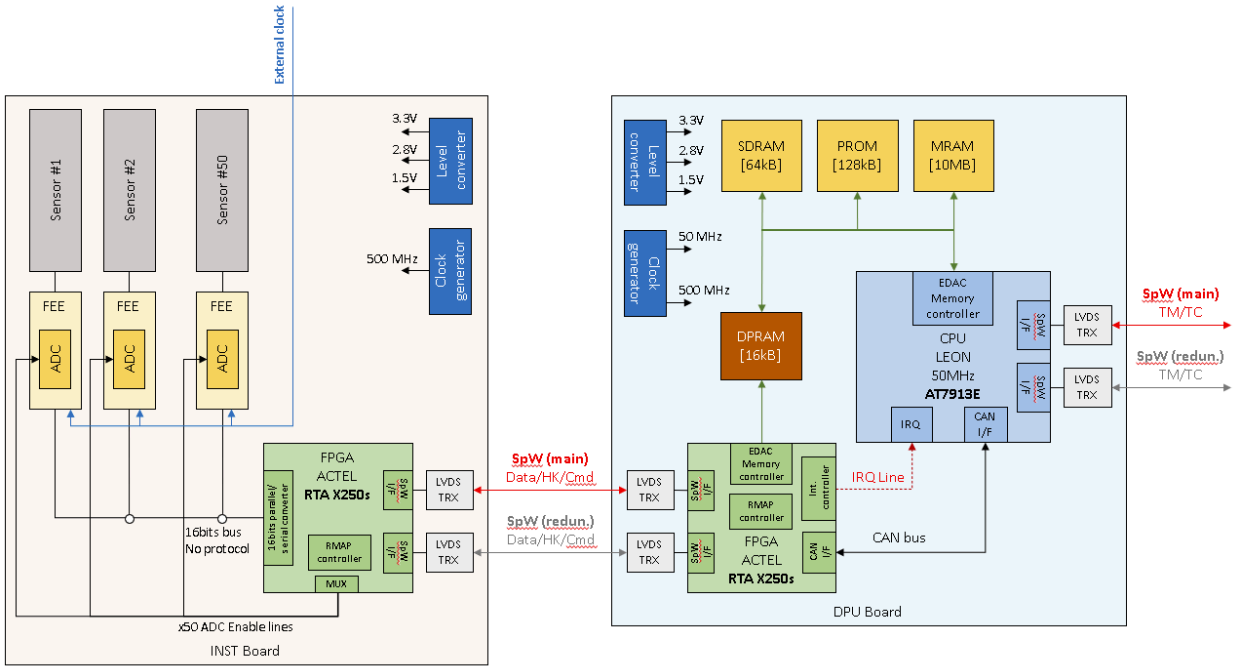

Figure 4.0: Hardware block diagram of the system. For clarity purposes, only 3 of the 50 FEEs and Sensors are shown.

To start a new measurement, the SpaceCraft OBC sends an instruction to the CPU through the CPU TM/TC bus. Once the command has been validated and once the instrument is ready to start the acquisition, the CPU will send a set of commands to the DPU FPGA to initiate the measurement procedure. The DPU FPGA will in turn send a set of measurement commands to the INST FPGA (through the SpW bus) which will start the acquisition process.

During the first initialization, the INST FPGA will readout all the data contained in all sensors to clear all the previously accumulated charges. As described in the FEE section, each FEE/ADC is read using a common 16bits bus. As only one of the 50 ADCs is selected at one time (using the 50x enable lines), there is no risk of interference on the bus.

The RTAX250s contains four small core tiles, each one containing three 4kbits SRAM units, leading to a total FIFO RAM size of 48kbits per FPGA. The internal memory of the FPGA can therefore not hold all the data of one measurement. As the sensor readout time is not known, continuous transfer of the data is not possible. To solve this problem without adding an external RAM to the FPGA, it has been chosen to split the full readout sequence into 10 separate

smaller sequences of 8kbits each (5 sensors per readout sequence). This means that SpW IP cores can be configured to share three FIFO memory blocks (12kbits, corresponding to 25% of the internal FPGA RAM). Calculations have shown that other major IP cores require about 4 memory blocks in total, leading to a 40% memory development margin. Therefore, each measurement will lead to 10 RMAP packets in total.

Once processed by the INST FPGA (Parallel to Serial, …) the raw data is sent to the DPU FPGA which then transfers the data to the DPRAM (buffer). At this point in the project, DPRAM transfer time has been neglected, but documentary research showed that common transfer times tend to be on the order of 10-30ns, leading to a nominal write time of 200μs per packet (note that the DPRAM writing sequence can be performed at the same time as sensor readout). As DPRAM can be accessed from two ports, the CPU can start processing the data by reading the first packets as the FPGA is still writing some data in the DPRAM. Therefore, the DPU FPGA will have enough time to write the remaining data of the last packet as the CPU is starting to read the first packets. The following figure shows the simplified timing diagram of a measurement from the start of the acquisition to the end of the processing.

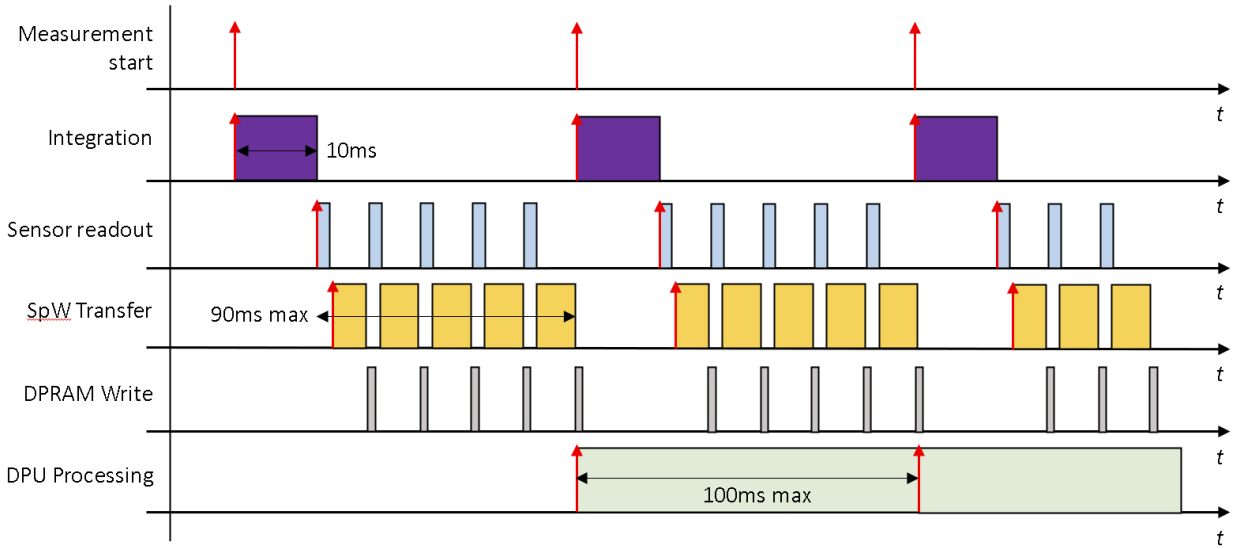

Figure 5.0: Simplified measurement timing diagram.

Data throughput

In the following section we make order of magnitude calculations to estimate required transfer rate and compare it with the [S2] System specifications of 50 Mb/s.

As T_{ACQ} and T_{INT} are set by the Instrument specifications, we can calculate the effective transfer time as $T_{TRF} = T_{ACQ} - T_{INT} = 90\,ms$. Neglecting the sensor readout time, FPGA processing time and DPRAM write time, the ADC conversion and readout time (for one packet) can be written as $T_{ADC\_RO} = N_{ACQ\_BITS}/f_{ADC} \approx 1.7\,ms$.

As described in section Hardware block diagram, a measurement is transferred as a sequence of $N_{PACKETS} = 10$ RMAP packets, which means that each packet contains $N_{PACK\_BITS} = 8kb$. The following table summarizes the standard RMAP packet structure and associated size (in bits).

SpaceWire RMAP Packet (write)				
Command	Header	Target address	32	bits
		Reply address	32	bits
		Address	32	bits
		Data Length + CRC	32	bits
	Data		8000	bits
	Data CRC + EOP		12	bits
Reply	Header	Reply address	32	bits
		Target address	32	bits
		EOP	4	bits
Total			8140	bits

Table 4.0: Simplified measurement timing diagram.

Using this data, we can calculate the nominal size of a single RMAP TM packet, which can be expressed as $N_{PACK\_SIZE} = N_{ACQ\_BITS} + N_{PACKET\_BITS} = 8.14\,kb$. Therefore, $N_{PACKETS} = 10$ RMAP packets will lead to a minimum $N_{DATA\_SIZE} = N_{PACKETS} \times N_{PACK\_SIZE} = 81.4kb$, from which we can compute the minimum required SpW data transfer rate for the RMAP protocol: $D_{IN\_RATE} = \alpha_{ENCODING} \times \frac{N_{DATA\_SIZE}}{T_{TRF} - T_{ADC\_RO}} = 1.15\,Mb/s$. By choosing a margin of 20% the required transfer rate becomes $D_{IN\_RATE} = 1.38\,Mb/s$. For a SpW throughput speed of 50 Mbps, the SpW occupancy rate is on the order of 2.76 % which satisfies system specification. The following table summarizes the timing and data rate properties of the SpW link:

SpW RMAP Packet (write) data transfer		
Integration time	10	ms
Transmission time	90	ms
ADC readout time	1.7	ms
SpW transfer rate	50	Mbits/s
Required transfer rate	1.38	Mbits/s
SpW occupation rate	2.76	%

Table 5.0: Simplified measurement timing diagram.

Power consumption budget

The following table summarizes the estimated power consumption of our system using a set of empirical rules.

Component	Number of units	Power consumption in (mW)
Processing		
Processor	1	$1 \times 1000 = 1000$
FPGA's	2	$2 \times 500 = 1000$
SpW controllers	2	$2 \times 250 = 500$
Memory		
DPRAM	1	$1 \times 400 = 800$
SDRAM	1	$1 \times 1000 = 1000$
MRAM	1	$1 \times 1000 = 1000$
Instrument		
FEEs	50	$50 \times 50 = 2500$
Sensors	N/A	$0.02\,mW/pxl \times 100 \times 50\,pxl = 100$
Other		
Oscillators	2	$2 \times 400 = 800$
Power Supply	1	$1 \times 500 = 800$
Total power budget		
Total (mW)	N/A	9500
Total (W)	N/A	9.5

Table 6.0: Simplified measurement timing diagram.

Spacecraft interface

Due to the absence of instrument scientific requirement specifications (orbit, quantity of interest, type of linear sensor, …), it has been chosen to leave the measurement data storage responsibility to the spacecraft manufacturer. This decision has been taken in order to maximize the Data Processing Unit adaptability to multiple mission and sensor types (optical, …).

The Instrument has 4 main states : *OFF, IDLE, READY* and *MEASUREMENT*. To turn off the instrument (*OFF* mode), the *MODE_OFF* command first has to be sent to the DPU board to initiate the turnoff sequence. Once this mode has been initiated, the instrument will switch to low-power mode, only running survival functions related to thermal regulation and wake up

listening. The instrument can be powered-up by sending the *START_INSTRUMENT* command through the SpaceWire interface. Once this command has been received, the instrument will move to *IDLE* mode and will await further instructions. Before starting a measurement procedure, the instrument must first be switched to *READY* mode by sending the *INIT_MEASUREMENT* command to the instrument. This command will initiate a quick instrument self-check (Inst. SpW interfaces, FEE, Sensors, Memory, …). If the self-check is successful, the instrument will return an acknowledge signal to the OBC and switch to ready mode. To start a new series of acquisition, the spacecraft OBC shall send the *START_MEASUREMENT XX* command through the SpaceWire interface of the DPU board, with XX the number of required measurements (1h max. = 36000 consecutive measurements, see section SDRAM). During this phase, the instrument will perform XX measurements (10ms integration at 10Hz) and will return the number of hotspots found during each acquisition. As only one hotspot can be found per pixel, only one byte is necessary to transmit the number of hotspots to the platform. The instrument will also transmit the housekeeping (HK) data corresponding to each measurement (characteristic voltages, temperature of the sensors, …). The instrument must be in READY mode to be able to perform measurements. The TM/TC data is transmitted using the standard SpaceWire **CCSDS protocol**:

- The previously defined TCs (commands) are identified through a unique & single byte address (256 commands max.) followed by a 2 byte configuration number (e.g. number of acquisitions to be performed, …). The standard CCSDS TC packet is comprised of a 10 bytes header followed by a 6 bytes packet secondary header, 3 bytes command word and a 2 byte CRC, leading to 210 bits (26.3 bytes) of TC raw data transmission from the platform to the instrument.
- The processed measurement data is sent after performing the XX acquisitions programmed by the *START_MEASUREMENT XX* command. In the worst case scenario, 36000 acquisitions are performed (10s measurement at 10Hz). In that case, the standard TM CCSDS packet consists of a 4 byte SpaceWire header, a 6 byte CCSDS primary header, a 14 byte secondary header and 36 kBytes of processed data with the corresponding 2 byte CRC. Assuming that HK data contains the temperature of each sensor at the start of the acquisition (2 bytes per sensor) and a set of 10 characteristic voltages and temperatures (two bytes per value), we get a total of 45.2kbits (5.6kB) of raw transmitted data per measurement.

It is recommended to perform frequent memory integrity checks by sending the MEM_CHECK command to the DPU board. Once this command has been received, and no measurements are running, the CPU performs a full memory scan of the DPRAM and SDRAM to correct potential errors (EDAC).

Software subsystem

This section covers the design, prototyping and simulation of the Instrument Flight Software (IFS). The former will give a detailed description of the software architecture and of the data processing algorithm. This description will be followed by a rough CPU load estimation using a database of reference performance measurements. The implementation of the processing algorithm's simulator (TSIM) will then be discussed, leading to a more precise performance and CPU load estimation.

Software architecture

The software system of the instrument's Digital Processing Unit (DPU) can be decomposed into two main software subsystems. The first one corresponds to the Boot software (outside of the scope of this report), whose role is to handle the low level tasks required for starting the second software subsystem called the Instrument Flight Software (IFS).

To reduce computational overhead, communication (SpW, CAN, …) and memory management functions were implemented in hardware (memory controller IP, SpaceWire Interface IP, …). In turn, the IFS will cover basic numerical processing of raw sensor data. To comply with the instrument design requirements, the software will be written in C++ and integrated on a 50MHz LEON CPU located on the main DPU board of the instrument.

The IFS architecture will take advantage of the Object-Oriented Programming features provided by C++. A main *ProcessingCtx* class was created to handle numerical processing of raw instrument data. Additional features, such as *FDIR*, *Sorting*, *Communication* or *HouseKeeping* will be implemented in separate classes to improve maintainability and reliability.

Data processing algorithm

The objective of the *ProcessingCtx* class is to extract the number of hotspots from the raw data acquired by 50 linear detectors of 100 pixels each for each acquisition. This relatively simple processing chain can be decomposed into five main functions whose role will be given in more detail in the following sections of this report.

Sensor Data Acquisition

The first step of the processing chain consists in copying the sensor data $V_{ci}(k)$ located in the DPRAM to the CPU *MRAM* using the dedicated CPU memory bus. This step (performed using the *acquire_sensor_data* function) ensures that the FPGA located on the DPU is able to copy new measurement data onto the DPRAM while preserving previous measurement data during processing.

Median Computation

The second step of the algorithm consists in computing the median of each pixel measurement $V_{ci}(k)$ along all individual sensors (function *compute_median*). This step requires all the values of a given pixel along all sensors to be sorted in ascending order. At this stage, two sorting algorithms are considered: the HEAP and QUICK sorting algorithm. The performances of each algorithm will be evaluated in Selection of the sorting algorithm. Once all the k^{th} pixels of each sensor are sorted, the median $V_M(k)$ corresponding to the k^{th} pixel is computed as depicted in the following figure.

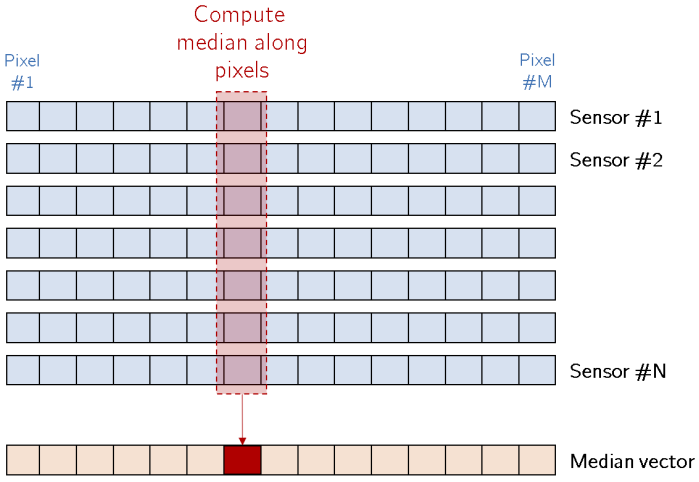

Figure 6.0: Simplified measurement timing diagram.

Mean vector computation

The mean vector $V_\mu(k)$ of the pixels of index k among all sensor is computed by the *compute_mean* function. As depicted in the following figure, outliers are removed by comparing the difference $\Delta_M(i,\ k) = |V_{ci}(k) - V_M(k)|$ to a predefined threshold value e_M (set to $e_M = 100$).

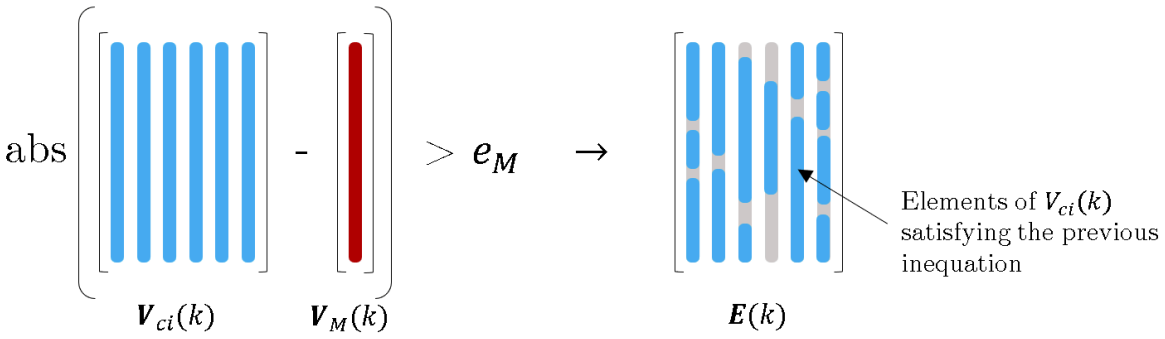

Figure 7.0: Simplified measurement timing diagram.

If $\Delta_M(i,\ k) < e_M$, then the value $V_{ci}(k)$ is used in the computation of the mean value $V_\mu(k)$ of each k^{th} pixel for every sensor i. If not, the value is discarded.

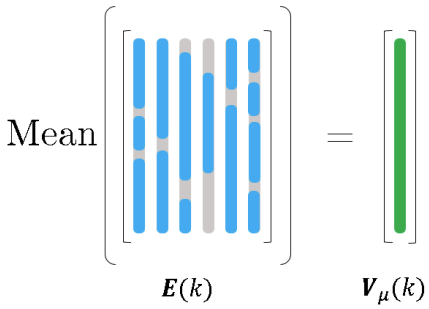

Figure 8.0: Simplified measurement timing diagram.

Reference vector

The reference mean vector $V_{ref}(k)$ is used for the identification of hotspots (see section Counting hotspots). In this algorithm, the first $V_\mu(k)$ vector (computed during the first acquisition of a given measurement sequence) is used to define the reference vector $V_{ref}(k)$ to which following $V_\mu(k)$ vectors will be compared to. Therefore, the *store_reference* function is used only during the first acquisition to set $V_{ref}(k) = V_\mu(k)$.

Counting hotspots

In the next processing loops following the definition of the reference vector, the mean values $V_\mu(k)$ are compared to the reference vector $V_{ref}(k)$ in order to detect and count the number of hotspots n_h present within the acquired data. The *compute_hotspot_count* function compares the $\Delta_\mu(k) = V_\mu(k) - V_{ref}(k)$ to a predefined detection threshold h_μ. If $\Delta_\mu(k) > h_\mu$, then the k^{th} pixel is considered to contain a hotspot, and 1 is added to the hotspot counter n_h.

CPU load estimation

CPU load estimation plays a crucial role in ensuring the feasibility of the instrument's numerical processing subsystem. Assuming that the CPU runs at its nominal frequency, the minimum duration of a cycle is 20ns, yielding a maximum value of 5 million cycles to complete the processing of a single measurement (100ms).

In this section, a rough estimate of the CPU load will be computed using CPU cycle measurements performed over a set of standard building-block functions (mean, sum, …). Using this data, a quick assessment of the feasibility of the system will be performed. Once validated, a more precise CPU load estimation will be computed using a simulation of the IFS.

Estimation using a reference execution time database

The CPU load corresponding to the algorithm described in previous sections was first estimated by decomposing its functions into multiple core operations (mean, sum, …). Using a database of reference CPU cycle measurements, the number of cycles required to perform each operation could be computed. The analysis of the algorithm showed that most operations were performed over the measurements of the k^{th} pixel of each of the 50 sensors, leading $n = 50$ array elements for each core operation. However, as CPU cycle measurements were performed using arrays of 10, 100 and 1000 elements, a linear interpolation was required to obtain the mean number of CPU cycles required per element for arrays of size 50:

$$n_{50} = \frac{n_{10} - n_{100}}{10 - 100}(10 - 50) + n_{10} \approx 0.44 \cdot (n_{10} - n_{100}) + n_{10}$$

In the case of the data acquisition and of the median and mean computations, core operations are iterated over each one of the 100 pixel indices. The final CPU load is obtained by dividing the available time (100ms) by the total loop processing time. The maximum CPU load estimation for each of the two execution configurations (first & nominal iterations) can be found in the Table below. Using this simple estimation method, we obtain a maximum CPU load of 27%, which is lower than the 30% upper limit set for phase A of the instrument design. At this point, we can be confident in the feasibility of the processing loop.

Rough CPU load estimates using standard operations	Cycles per element	Iterations	Cycles		Execution time (ms)	
			First iteration	Nominal	First iteration	Nominal
Copy data to RAM	17	100	85556		1.71	
Median (HeapSort)	225	100	1127222		22.54	
Mean	30	100	148333		2.97	
Comparison (Mean)	22	1	2222		0.04	
Copy to ref array	17	1	1711		0.03	
Sum (difference)	15	1		1467		0.03
Comparison (Hotspots)	22	1		2222		0.04
Total (ms)					27.30	27.34
CPU load (%)					**27%**	**27%**

Table 7.0: Simplified measurement timing diagram.

Development of the IFS simulator

To obtain more precise CPU load estimations, a C++ TSIM IFS simulator was implemented to perform precise step-by-step performance measurements of the main process loop functions (source code available on Github).

Simulations were performed using a dataset of 50 simulated measurements. This dataset was generated by a model whose inputs correspond to the hotspot count and additive noise level. The testing data, together with the initial number of Hotspots used during generation, are given inside the *data.cpp* source file. The main simulation loop runs the processing chain computations on each simulated measurement, as described in the Data processing algorithm section, and measures the amount of cycles required by each function.

ProcessingCtx class

The **ProcessingCtx** class implements a simplified version of the actual *ProcessingCtx* class functions described in the Data processing algorithm section. All variables used throughout the execution of the processing chain were encapsulated within the *ProcessingCtx* class to improve readability and to preserve data integrity.

To simulate the raw data transfer from the DPRAM to the SDRAM, the *acquire_sensor_data* function performs a copy of one of the reference measurements (50x100 unsigned short array) to the *raw_data* array of the *ProcessingCtx* class. This array simulates an SDRAM memory block which is allocated to raw data during processing. While the actual transfer rate can vary between SDRAM and DPRAM, this technique allows for the measurement of the average number of cycles needed to perform the raw data acquisition. The *compute_median* function simulation is based on an implementation similar to the one of the final IFS. The code has been optimized to reduce the number of transfers between the SDRAM and the local FLASH memory of the CPU. However, as we will see in the following sections, the performances of the *compute_median* algorithm are strongly coupled to the performances of the sorting algorithm used. The *compute_mean* function was simulated using a simple mean calculation by first looping over each pixel of index k and then computing the mean value $V_{\mu}(k)$ for each sensor index i. To optimize computation time and memory usage, the mean was computed by first summing all pixel of index k (using a temporary variable of type *unsigned int* to prevent overflow) and then dividing by the number of valid measurements. As stated in the Software architecture section, the removal of outliers was performed by only summing the pixel values satisfying $\Delta_M(i,\, k) = |V_{ci}(k) - V_M(k)| < e_M$. The *store_reference* function was implemented to perform a simple copy of the mean vector $V_{\mu}(k)$ to the reference vector $V_{ref}(k)$ stored within the *ProcessingCtx* class on the first acquisition of a given measurement sequence. Finally, the *compute_hotspot_count* function was simulated using a simple for loop over all elements of the mean vector $V_{\mu}(k)$. As we wish to compare the value of the difference $\Delta_{\mu}(k) = V_{\mu}(k) - V_{ref}(k)$ clipped to zero (meaning that $\Delta_{\mu}(k) = 0$ if $V_{\mu}(k) - V_{ref}(k) < 0$) to the detection threshold

$h_{\mu} > 0$,we can directly compare $V_{\mu}(k) - V_{ref}(k)$ to h_{μ} without performing the zero clipping operation. This trick allows for a substantial improvement in computation time.

Selection of the sorting algorithm

As mentioned in the ProcessingCtx class section, the performances of the *compute_median* function are strongly coupled to the sorting algorithm used. As part of this project, two sorting function implementations are provided: **QuickSort** and **HeapSort**.

A quick complexity analysis shows that the QuickSort execution time scales with $O(n^2)$ (worst case) while HeapSort only scales with $O(n\,log(n))$, which means that the HeapSort algorithm should be quicker than QuickSort in theory. However, the HeapSort algorithm is usually substantially slower than the QuickSort algorithm for small n (which is true in our case) as execution time is limited by memory R/W time in this regime. In fact, it is possible to show that the HeapSort algorithm will lead to the swapping (most expensive operation in that case) of all the array elements at least once. On the other hand, QuickSort does not perform unnecessary value swapping, which considerably lowers memory usage. Another consideration is that HeapSort requires lower memory than QuickSort, but given the small arrays considered in our case, we are more concerned with execution time than memory usage. Both sorting algorithms are unstable (meaning that after sorting, the sensor index/pixel value relationship is lost: we cannot say which sensor has produced which data). However, since the final goal is to compute the median, the stability of those algorithms is not relevant. Therefore, based on performance, it would seem that QuickSort is the best sorting algorithm for this application.

Performance measurements of the *compute_median* function show that the HeapSort algorithm leads to an average value of 1017232 cycles, while the same function implemented using QuickSort only requires an average of 517461 cycles. QuickSort cycle counts seem to show slightly higher variance between iterations, but the minimum and maximum cycles measured for this algorithm are still much lower than the ones measured for the HeapSort algorithm (see the Table below).

Sorting algorithm comparison (median)	**Mean (cycles)**	**Minimum (cycles)**	**Maximum (cycles)**	**Standard deviation (cycles)**
HeapSort	1017232	1014488	1021618	1429
QuickSort	517461	514483	520361	1465

Table 8.0: Simplified measurement timing diagram.

Following those measurements, given the much better performance obtained with the QuickSort algorithm, the former has been chosen to implement the median computation within the *compute_median* function.

Final CPU load estimation

Comparison between the expected number of Hotspots (used to generate the testing data samples) and the predictions of our simulation revealed some discrepancies between the predicted and expected number of Hotspots for the pixels 4 and 8.

Investigations showed that the most probable explanation for this discrepancy was the fact that the reference number of Hotspots given as part of the testing data didn't correspond to the number of Hotspots that would be detected by a functioning algorithm, but rather to the number of Hotspots used to generate the testing data. It has been brought to our attention that the algorithm used to generate the testing data starts by generating the pixel values corresponding to the number of hotspots, and then adds gaussian noise to the testing data. This random noise, which modifies pixel values, can decrease the mean signal to values below the detection threshold, hence modifying the number of actual Hotspots inside the testing data. The high success rate of our algorithm in finding the correct expected number of Hotspots (96% of simulations) and this observation led to the validation of the IFS simulation predictions.

Counting the total number of hotspots, we get 447 hotspots found instead of the 449 expected (over 50 testing samples), which would a priori lead to 99.6% success rate with only a 0.4% false negative rate. However, as the position of the detected hotspots is not given (pixel index), there is no way of knowing if some false detections and non-detections are not canceling out for a given image. Therefore, we have chosen to only compare the total amount of hotspots per image, leading to the 96% success rate found before.

Follow-on performance measurements performed over the whole testing sample led to the results depicted in the figure below, showing the number of simulated CPU cycles measured for each simulated raw measurement.

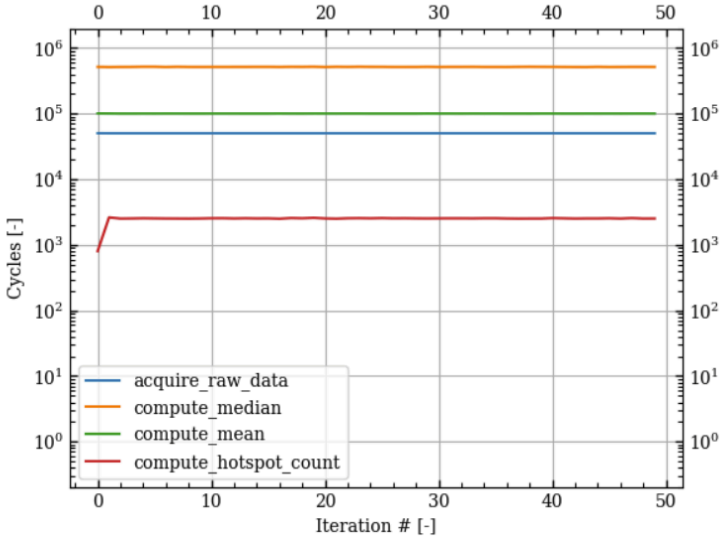

Figure 9.0: Simplified measurement timing diagram.

It is clear that the main bottleneck corresponds to the median computation, which is itself dependent on the efficiency of the sorting algorithm. To estimate the average CPU load, the number of cycles were averaged over all simulated raw measurements. Multiplying the resulting average number of cycles per acquisition by the nominal duration of a CPU cycle, one obtains the execution time of each function. Finally, dividing the total execution time of the loop by the maximum duration of a measurement (100ms) yields the total CPU load/occupancy rate. The results of the measurements (optimization level -o2) are available in the Table below.

QuickSort	**Computed Reference Sample (cycles)**				**First Iteration**			**Nominal iteration**		
	Mean	**Min**	**Max**	**Std**	**Mean**	**Min**	**Max**	**Mean**	**Min**	**Max**
Data acquisition	50434	50433	50503	10	1.01	1.01	1.01	1.01	1.01	1.01
Compute median	517461	514483	520361	1465	10.35	10.29	10.41	10.35	10.29	10.41
Compute mean	100519	100411	101019	104	2.01	2.01	2.02	2.01	2.01	2.02
Store reference vector	804	804	804	0	0.02	0.02	0.02			
Compute hotspot count	2540	2508	2625	20				0.05	0.05	0.05
Total (ms)					13	13	13	13	13	13
CPU Occupation Rate (%)					**15%**	**15%**	**15%**	**15%**	**15%**	**15%**

Table 9.0: Simplified measurement timing diagram.

Our measurements show that CPU load levels stay at all time lower than 15%, which is compatible with the requirement that CPU load shall be lower than 30% for phase A. This calculation shows a compliance of the developed IFS architecture with the ECSS regulations with a margin of 50%.

Therefore, while software features such as communication and FDIR have not been implemented and simulated yet, those results show a high confidence rate in the feasibility of the developed IFS architecture and implementation.

Conclusion

During the first part of the system design, a complete hardware architecture was developed with respect to system specifications. Mainly, the instrument and DPU board architecture were chosen to minimize the number of the SpW links. As a result, an optimal choice of a 16 bit common bus for the INST board was selected. In addition, order of magnitude estimates were made for the memory sizes of the memory units and the calculation of the data throughput via the SpW. Such calculations, allowed to show that the system design satisfies systems specifications. Lastly, the system power budget allowed us to estimate nominal system power consumption.

During the second part of this work, an IFS software architecture compatible with the requirements of the instrument was developed. The decomposition of the main processing functions of the DPU into a set of core operations allowed for the computation of a first CPU load estimate based on a reference performance measurements database. As this estimate was close to the upper CPU load limit given by the ECSS regulations for Phase A development, an IFS prototype was developed and implemented in a TSIM simulator.

Performance measurements on the simulator allowed for the comparison of multiple sorting algorithms, leading to the selection of the QuickSort algorithm for our application. CPU load estimations computed using the updated and optimized algorithm showed a maximum CPU load of 15%, which complies with ECSS regulations (50% margin) and therefore validates the feasibility of this IFS architecture.

Deliverables

Deliverable	Status
Report	Approved
Source code	Approved

Planning

Task	Assigned to	Deadline	Status
Create the GitHub	maynadie.t@gm…	Feb 11, …	Approved
Complete Hardware subsystem	Marcin Kovalevskij	Feb 14, …	Approved
Complete Software subsystem	maynadie.t@gm…	Feb 14, …	Approved