



# **Bare Metal Porting of Tasking Framework on a Xilinx Board**

**Submitted by**

**Ashish Gopal**



Institut Supérieur de l'Aéronautique et de l'Espace

**Master of Science Aeronautical and Space Systems**

**Major in S2: Avionics**

**Major in S3: Embedded Systems**

**Tutors:**

**Dr. Olaf Maibaum**, Simulations and Software Technology, DLR

**Dr. Arnaud Dion**, DEOS, ISAE



## **ABSTRACT**

The Tasking Framework is DLR's solution for distributed and parallel computation. In many aspects this can be considered similar to an Operating system which does not have any computation functions of its own. It provides access to the resources like CPU time for tasks which requires processing. These tasks can be requests from any sensor to process its data which could be crucial in attitude control of a space craft. This particular framework is currently employed by DLR in many of their existing projects. Each of these projects has different hardware requirements and hence the platform on which the framework is implemented differs from project to project. In order to have some uniformity and to reduce the efforts for more such implementation, the entire framework is divided into an API and a hardware dependent layer. As part of this project the API is relatively unchanged. All the modifications and changes are to be made only in the hardware dependent layer.

The objective of this internship is to achieve a bare metal implementation of this framework. In all the existing implementations there has always been an underlying operating system. In this case the operating system is inexistent. The idea is stemmed from the observations made regarding excessive usage of resources, undesired over heads etc when an operating system is running. The hardware selected for this implementation is provided by Xilinx. The evaluation is board is based on Xilinx's popular Zynq architecture. It comes with 2 ARM cortex A9 based processors which forms the core of the processing system on the board. It also accommodates an FPGA block which can be configured for specific purposes and used as a third processor for sharing the workload. In this internship however the FPGA was not used. The FPGA is however mapped with a bitstream file generated using the Vivado software package to be able to access the other peripherals. The implementation was successfully carried out and SMP architecture is implemented for sharing the workload between the two available processors.

## **ACKNOWLEDGEMENT**

This is to express my sincere gratitude towards those who have made this project work, a possibility. In retrospect, many have to be thanked. But a few names have to be singled out as they have helped me immensely in this regard.

Firstly I would like to thank Dr. Olaf Maibaum, Simulations and Software Technology, DLR for his continued support and guidance from the first day. His help and inputs regarding various technical aspects was a crucial part in this internship. I would like to thank Mr. Jan Sommer for inputs regarding the evaluation board and different software packages. Their technical expertise played a major role in successful completion of this project.

I would also take this opportunity to thank Mr. Daniel Lüdtke, Group Leader, Embedded Systems Group for his support and encouragement. The faith he placed in me and my abilities was a great motivation.

I wish to thank Dr. Andreas Gerndt and all technical and non-technical staff at the Simulations and Software Technology Institute at German Aerospace Center, Braunschweig for having me there and creating a wonderful working environment.

I would also like to thank Dr. Janette Cardoso, ISAE for her continues support and guidance throughout the duration of the internship. I also extend my heartfelt thanks to my internal tutor Dr. Arnaud Dion, ISAE and other staff of ISAE for all their help and support.

## **DECLARATION BY STUDENT**

This assignment is entirely our own work. Quotations from literature are properly indicated with appropriated references in the text. All literature used in this piece of work is indicated in the bibliography placed at the end. We confirm that no sources have been used other than those stated.

We understand that plagiarism (copy without mentioning the reference) is a serious examinations offence that may result in disciplinary action being taken.

## **CONTENTS**

<b>Simulations and Software Technology, Braunschweig .....</b>	<b>4</b>
--	----------

### **CHAPTER 1**

<b>1.1 Introduction to tasking framework .....</b>	<b>5</b>
<b>1.2 Functionalities of the framework.....</b>	<b>5</b>
<b>1.3 Application programming interface .....</b>	<b>6</b>
<b>1.4 Hardware layer .....</b>	<b>7</b>

### **CHAPTER 2**

<b>2.1Baremetal Porting .....</b>	<b>8</b>
<b>2.1.1 Hardware .....</b>	<b>9</b>
<b>2.1.2 Processing System .....</b>	<b>11</b>
<b>2.2 Global Timer .....</b>	<b>12</b>
<b>2.3 Memory .....</b>	<b>15</b>

### **CHAPTER 3**

<b>3.1 Application development for Linux Implementation .....</b>	<b>15</b>
<b>3.2 Getting acquainted to the hardware.....</b>	<b>16</b>
<b>3.3 Getting acquainted to Bare Metal and clock in the Zynq.....</b>	<b>18</b>
<b>3.4 Porting on single core.....</b>	<b>20</b>
<b>3.5 Starting the Second core.....</b>	<b>22</b>
<b>3.6 Multi Core Processing.....</b>	<b>23</b>
<b>3.6.1 Asymmetric Multi Processing.....</b>	<b>23</b>
<b>3.6.2 Symmetric Multi Processing.....</b>	<b>24</b>
<b>3.7 SMP implementation .....</b>	<b>26</b>
<b>3.8 Data sharing between processors.....</b>	<b>27</b>
<b>3.9 Writing SMP configuration to register.....</b>	<b>28</b>
<b>3.10 Synchronization primitives .....</b>	<b>29</b>

<b>3.11 Conclusion.....</b>	<b>30</b>
<b>3.12 Result.....</b>	<b>30</b>
<b>References.....</b>	<b>32</b>
<b>APPENDIX.....</b>	<b>34</b>

## **LIST OF FIGURES**

Sl. No	Figure	Description	Page No.
1	Fig 1.2 (a)	Tasking Framework Overview	6
2	Fig 1.4 (a)	General Control flow of Tasking Framework	8
3	Fig 2.1.1 (a)	Microzed 7020	9
4	Fig 2.1.1 (b)	SoC Block Diagram	10
5	Fig 2.2 (a)	Global Timer control register	13
6	Fig 3.2 (a)	GUI of vivado with Zynq PS7	17
7	Fig 3.4 (a)	Graphical representation of existing implementation	21
8	Fig 3.6.1 (a)	Pictorial representation of AMP	24
9	Fig 3.6.2 (a)	SMP architecture	25
10	Fig 3.6.2 (b)	Comparison of SMP and AMP	26
11	Fig 3.8 (a)	Snapshot of Configuration file generated in Xilinx SDK	28
12	Fig 3.12 (a)	Minicom window with output	31

## **LIST OF TABLES**

Sl. No	Table	Description	Page No.
1	Table 2.2 (a)	Registers for Global Timer access	13

## **Institute for Simulations and Software Technology, Braunschweig**

The institute for Simulations and Software Technology takes up extensive research work in the field of on board computers and software technologies employed for space applications. The field of research spans across various systems and services that uses software technology. There is a continuous effort being made to improve the existing technology along with coming up with innovative ideas for future use.

The institute is divided into 3 major working groups, Embedded Systems Group, Interactive Visualization Group and Modeling and Simulation Group. All three of them work in close association with each other to ensure successful completion of projects. The institute has developed various software modules which are employed on some of the existing projects of DLR e.g. eu:CROPIS (under development), ATON, MAIUS, BIROS, TET-1 etc.

Current areas of research include software for distributed systems and mobile systems, software technologies for embedded systems, visualization, and high performance computing [19]. Apart from these a number of projects from the European Union and other DLR institutes spread across Germany are addressed by personnel of this institute.

The institute strives to excel in its domain. Innovation and research are the key aspects of this institute. A good amount of time and funding it receives are channeled towards research and development. Within the Embedded Systems group a lot of work is being under taken to improve the computing technology on space missions. The OBC NG (on board computers- next generation) and SCOSA are examples of projects that are working on this issue. Besides research, there is continuous monitoring of various software modules already deployed on some of the space missions by DLR.

The Modeling and Simulations working group are extensively working on the concept of Virtual Satellites. This particular project is used by some of the universities for educational purposes and is developed and supported by the institute.

The institute attracts a number of students from Master's and Bachelor's courses for undertaking various positions like Thesis, internships etc.



# CHAPTER 1

## **1.1 Introduction to the Framework:**

The tasking framework is DLR's solution for parallel and distributed computing to be used for space applications. The entire project involving the Tasking Framework is classified under a broader project titled the "Scalable On-board computing for Space Avionics (SCOSA)". As the name suggests the idea is to have smarter and more efficient ways to process the data which are received from various sensors mounted on a satellite.

The framework is itself divided into 2 separate parts. One which consists of various classes and functions which an application programmer uses to create and execute tasks on the framework implemented as an API. The other consists of classes and functions which deal with how the incoming tasks are processed and how they are scheduled. The latter also consists of some hardware specific parameters which are crucial for the implementation. The idea behind such an implementation is to make the entire framework portable and minimum or no changes to the API will be required for each porting. Having this separation also facilitates the portability of an application. It is easy to run an application on different operating systems and hardware by separating the API from the Hardware layer.

The framework is implemented using C++ language. The existing implementations were working on top of an operating system (example linux, RTEMS etc). Having an operating system running on the processor to handle the applications can be beneficial when it comes to services that are offered by the OS. Although the presence of operating system would be a big help for application programmers, it also introduces unwanted overheads resulting in wasted CPU time. It was for this reason that a bare metal implementation was considered for the tasking framework which helps in reducing the unwanted over heads. Another reason why a bare metal implementation was considered is for the size of the binary files that are created. When you have an operating system, e.g. Linux the size of the binary file generated is too high as compared to a bare metal. Accommodating a file this big on an embedded system is an expensive affair from the memory perspective. This is also a point of concern when uploading these binary files on a project.

## **1.2 Functionalities of the Framework:**

As mentioned in the introduction, the framework is used to process the data from sensors and give respective output. The data being processed are important as it plays a part in the satellite's trajectory and attitude control. One of the standout features of the framework is the reactive behavior it offers. The processing of a task begins only when the required inputs are available on the channel. In order to understand the framework it is essential to have a brief understanding about the implementation and how to use it.

The framework can be broadly represented by the picture below:



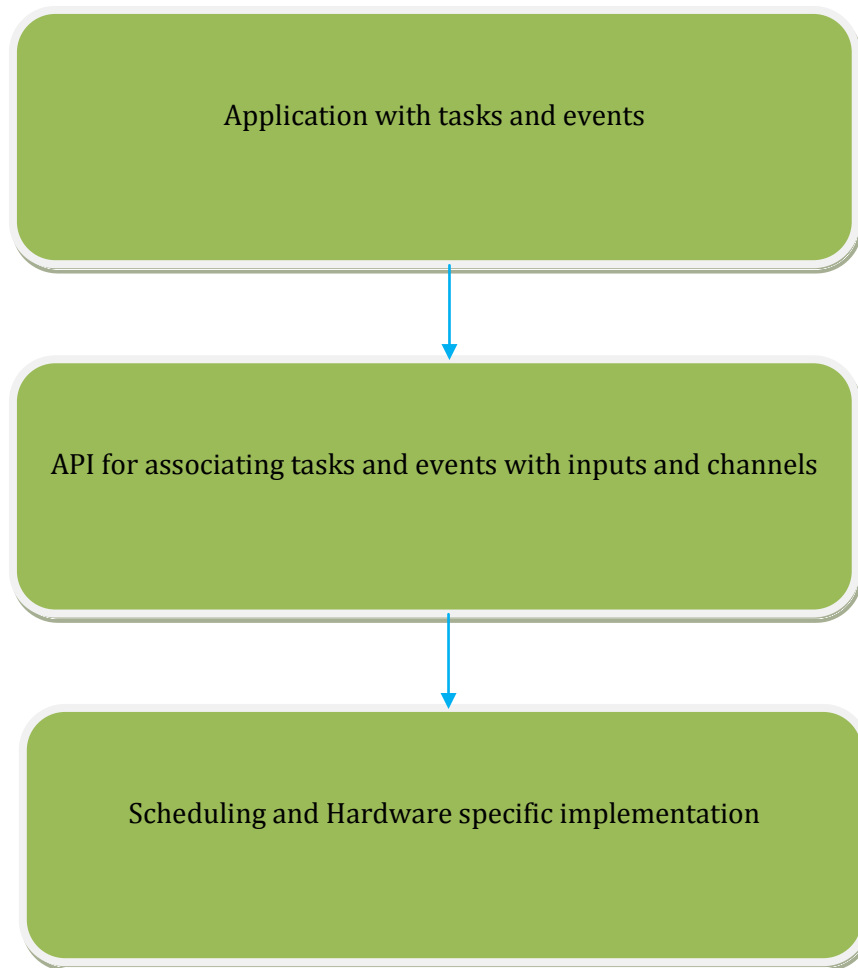


Figure1.2 (a): Tasking framework overview

The second and third block of the above diagram is where the important aspects about the framework is housed.

### **1.3 Application Programming Interface:**

The API of the framework has an elaborate structure and is very well organized. It gives the programmers multiple ways to access the services offered by the framework. Each access to the framework is made by using “Tasks”.

**Task:** A task corresponds to a single processing operation. A task is associated with one or many inputs which are available on different channels. If all the inputs associated with a task are activated (available), then that particular task is activated and is queued up for being executed. An option of input marked as final is made available. If a task has an input marked as “final”, when this input is activated the corresponding task is activated irrespective of the status of other associated inputs. All the functionalities associated with a task are provided in the class “Task”

which forms a very important part of the API. The processing that is required to be done by each task is performed by overloading the method “execute” of the class Task.

**Task Inputs:** This is an essential part of each Task. The data to be processed are passed on to a task as inputs. Inputs are associated with channels. The relation being a 1:N relation, i.e. one input is associated to only 1 channel where as each channel can be associated to N inputs. The functionalities are provided in the class TaskInput. It is possible to control the activation of an input by specifying a threshold for the number of activations. As mentioned earlier there is also a flag indicating an input as “Final”, which when available activates the task irrespective of status of other inputs. This was included as a time out mechanism to avoid a task from waiting for more than a pre-decided time.

**Task Channel:** The Task Channel is a base for a data container implementation. This does the function of storing and distributing the data for multiple tasks. Each channel can be associated with more than 2 inputs. A method “push” is provided which notifies all associated task inputs that a new set of data is available on the channel.

In a practical scenario, many of the sensors associated with a satellite are designed to send data at a specific interval of time. In order to serve such recurring process requests the concept of what is called a “TaskEvent” is provided in the API. It is to be noted that the TaskEvents are not tasks. These are extension to the TaskChannel. By default these are designed to execute a push method to push data on to the channel at a specified time. There are 2 types of such events which are supported by the framework, one which executes on a periodic basis and other which is executed after a specified time lapses after the call to “reset” method of that particular channel.

## **1.4 The Hardware layer:**

This forms the Heart and Brain of the Tasking Framework. This part of the framework houses the 2 most important implementations, Scheduler class and Clock.

**Scheduler:** As the name suggests the Scheduler class is responsible for running the tasks on the available cores. It maintains a list of objects which are to be executed. A request for execution is made by calling the “perform” method of the scheduler class. We make use of the name mangling functionality here to have multiple functions by the name perform all accepting different parameters. Once a task is activated, it calls the perform method of the scheduler indicating that the task is ready to be scheduled. The scheduler class holds a list of objects called “Executable objects”. Each executable object represents a task or an event which needs to be executed. To schedule the execution of any task/events we make an executor object. Ideally the number of executors = number of cores. So when a task calls the perform method, we first assign it to an executable object. Once an executable object is assigned we search for a free executor thread, if found the executor is immediately scheduled on one of the cores if not, we place the executor in a queue of objects waiting for an executor to be free. The executor makes a call to the “execute” function of the Task which contains all the task specific computations and calculations.

**Clock:** The clock is the other important class worth mentioning in this part. The clock is extensively used for addressing the timing requirements of the events. When a preset value of the timer is reached it triggers a push on the event which then gets scheduled by the scheduler. The events and the tasks associated with the events have the highest priority and hence are among the first tasks to be scheduled.

In the existing implementation, these classes were using the “pthread” library for threads and synchronization purposes. In the bare metal implementation these classes underwent major changes and shall be explained in detail later in the document.

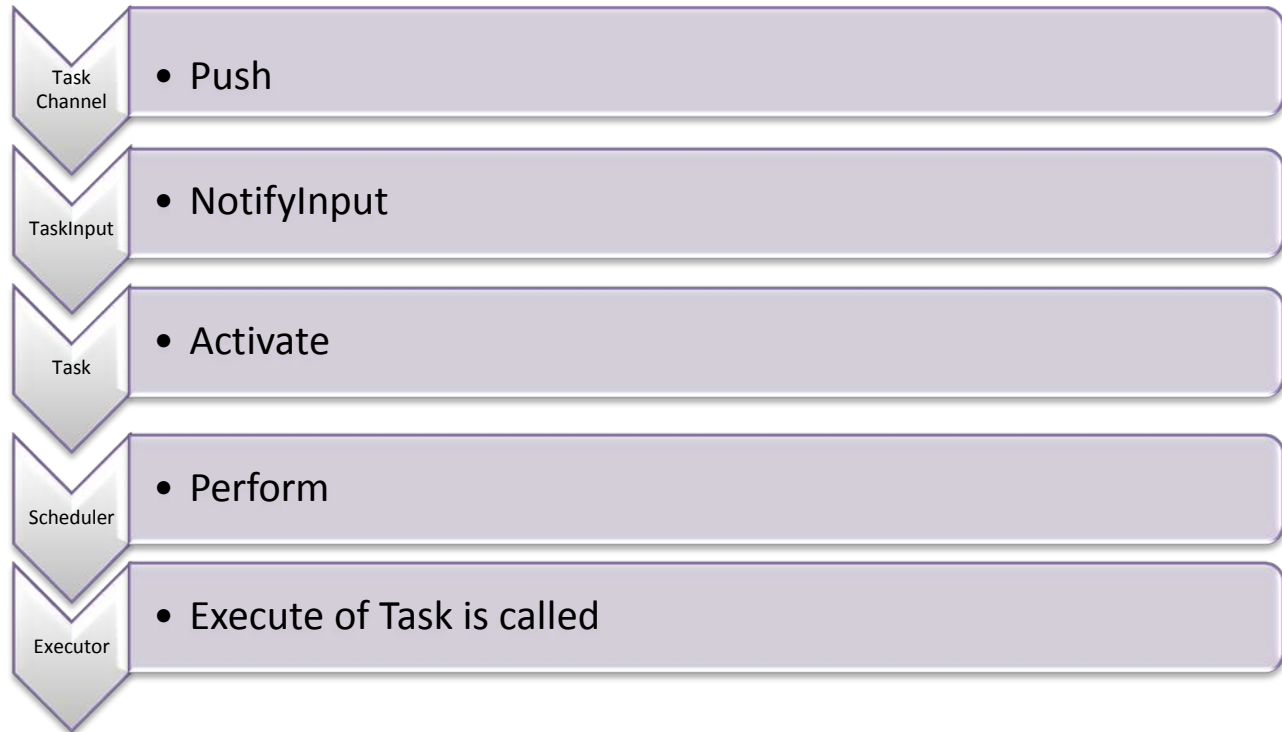


Figure 1.4 (a): General control flow of tasking framework

**Monitor:** This is one of the more important classes in the existing implementations. This class provides for synchronization between threads. This is extensively used for have a thread safe implementation. From the perspective of this project this class has been completely omitted as the synchronization mechanisms for bare metal model is a simple locking function which has been included in the scheduler class itself.

## CHAPTER 2

### 2.1 Baremetal Porting:

The objective of this internship was to achieve the Baremetal porting of the above mentioned Tasking Framework on an evaluation board. Before going any further in discussing the details of this project it is very important to understand what the word “bare metal” implies.

The word “Baremetal programming” means programming without various layers of abstraction or as some would call it programming without an operating system. There are many reasons why one would take the OS out of the equation and program directly on the hardware. The application is the only software running on a processor in the bare metal implementations. The most common is simply because operating system uses up more resources like RAM, flash etc for a rather simple application. These resources are available on a limited quantity when working with embedded systems and the programmer needs to be very careful in using them.

Before jumping into considering a bare metal implementation the application programmer needs to carefully access the requirements and complexity of the application. The presence of an operating system increases the ease of programming as it brings in host of useful services which are otherwise very difficult to handle. Trying to implement some complex services which are already provided by operating systems is not considered a good idea. So careful study and requirement analysis needs to be performed before deciding a bare metal implementation.

### **2.1.1 Hardware:**

Before getting into the specifics about the bare metal implementation it is important to introduce the hardware that is being used for this project. This section will introduce the readers to some of the key aspects of the Microzed board which are essential for getting an idea of how the porting is achieved. The hardware that is being used to host the framework is a Microzed 7020 evaluation kit. The picture of this kit is included below.

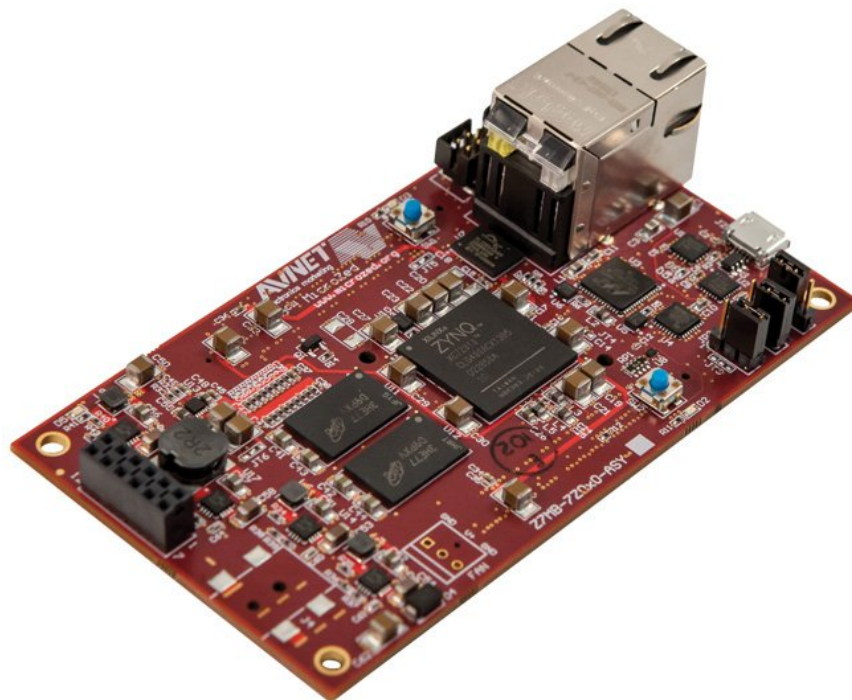


Figure 2.1.1 (a): Microzed 7020 [18]

This kit is based on Xilinx’s Zynq-7000 All Programmable SoC architecture. This provides integration of two ARM Cortex A9 MPCore based Processing Systems (PS) and a Xilinx

programmable logic (PL). The board provides an interesting mixture of features. From the perspective of the project the following will be the more essential:

1. Processing System :
  - Application Processing Unit (APU)
  - Memory Interfaces- DDR RAM, OCM
  - Input/output peripherals
  - Interconnects
2. Programmable Logic
3. Global clock

The following image has been taken from the Xilinx user guide to give a brief idea of the entire architecture.

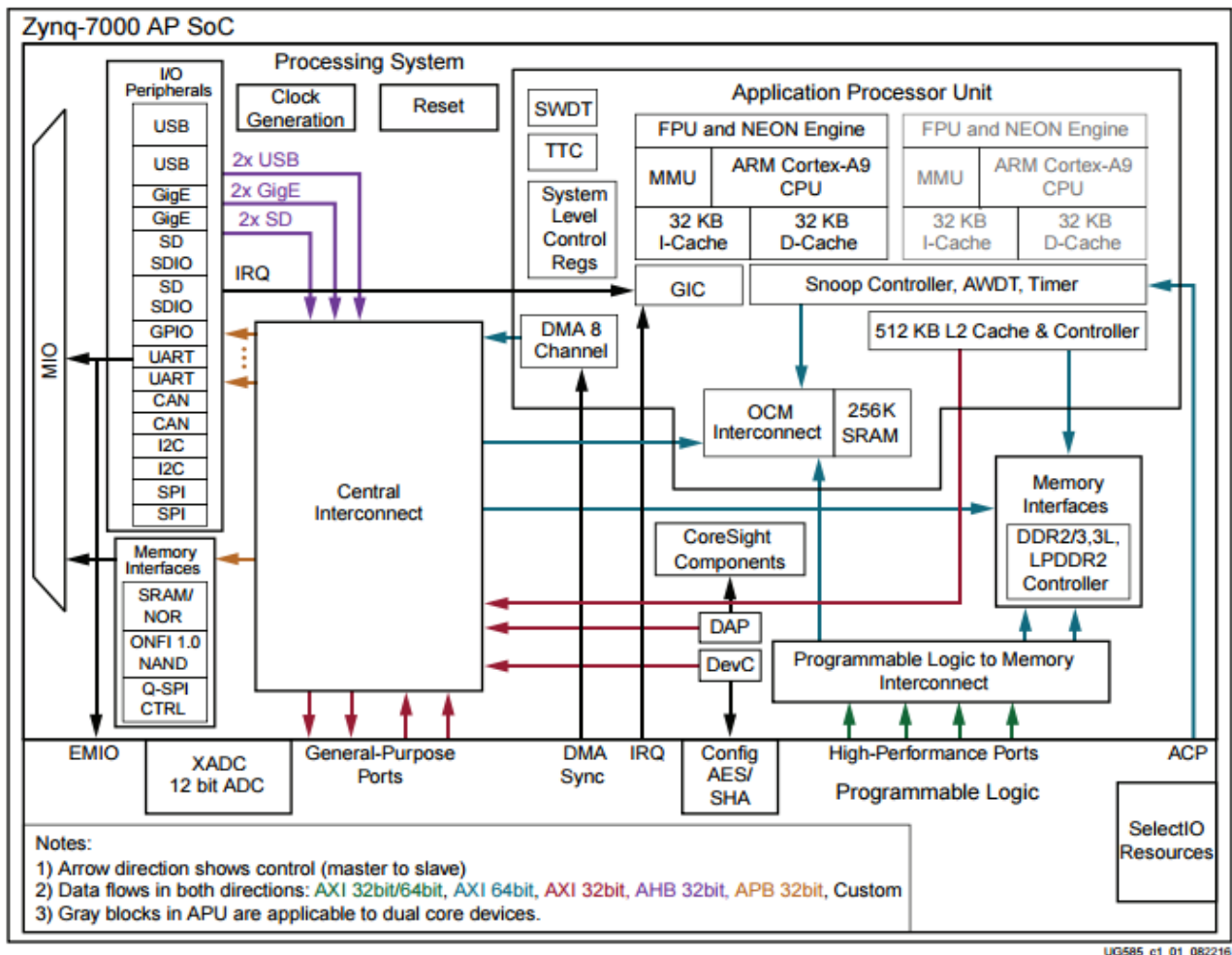


Figure 2.1.1 (b): Zynq SoC block diagram [1]

### **2.1.2 Processing Systems:**

Application Processing Unit (APU): The APU used is ARM cortex A9 processor based on the ARM v7 architecture.

The Cortex-A9 MPCore processor consists of:

- From one to four Cortex-A9 processors in a cluster and a Snoop Control Unit (SCU) that can be used to ensure coherency within the cluster.
- A set of private memory-mapped peripherals, including a global timer, and a watchdog and private timer for each Cortex-A9 processor present in the cluster.
- An integrated Interrupt Controller that is an implementation of the Generic Interrupt Controller architecture. The integrated Interrupt Controller registers are in the private memory region of the Cortex-A9 MPCore processor.[2]

The SCU connects one to four ARM cortex processors to the memory system through AXI interfaces. It is an important element in maintaining cache coherency related issues. The ARM processors have a shared L2 cache and independent LI caches. When working in multi core mode the coherency becomes a very important issue to address. SCU plays an important role here. The SCU is clocked synchronously and at the same frequency as the processors.[2]

The SCU functions are to:

- Maintain data cache coherency between the Cortex-A9 processors.
- Initiate L2 AXI memory accesses.
- Arbitrate between Cortex-A9 processors requesting L2 accesses.
- Manage ACP accesses.

The SCU on the cortex A9 does not provide hardware management of coherency on the instruction cache.

Generic Interrupt Controller (GIC): The PS also includes a Generic Interrupt Controller which manages and assigns all the interrupts that happen on the system. The Interrupt Controller is a single functional unit that is located in a Cortex-A9 multiprocessor design. There is one Cortex-A9 processor interface per Cortex-A9 processor. The processors access it by using a private interface through the SCU. [2]

The GIC is a centralized resource for supporting and managing interrupts in a system that includes at least one processor. It provides:

- registers for managing interrupt sources, interrupt behavior, and interrupt routing to one or more processors
- support for:
  - the ARM architecture Security Extensions
  - the ARM architecture Virtualization Extensions
  - enabling, disabling, and generating processor interrupts from hardware (peripheral) interrupt sources

- Software-generated Interrupts (SGIs)
- interrupt masking and prioritization
- uniprocessor and multiprocessor environments
- Wakeup events in power-management environments.

A device that uses a GIC can handle 4 different types of interrupts. Out of these we are interested in two types which are:

- Peripheral Interrupts: As the name suggests these are interrupts that are generated by a peripheral device e.g. timer. The GIC architecture further breaks down peripheral interrupts into
  - I. Private Peripheral Interrupt: These interrupts are private for each processor.
  - II. Share Peripheral Interrupts: These interrupts can be routed or assigned by the distributor on any processor that is connected to the GIC.
- Software Generated Interrupts: This is an interesting feature provided by the GIC here. The software generated interrupts are raised by writing to a particular register in the GIC. This register is called the GICD\_SGIR. This serves as a means of inter processor communication. When an SGI occurs in a multiprocessor implementation, the CPUID field in the Interrupt Acknowledge Register, GICC\_IAR, or the Aliased Interrupt Acknowledge Register, GICC\_AIAR, identifies the processor that requested the interrupt.

When dealing with Interrupts in a multi processor environment, Interrupt banking and register banking is a very important concept because this gives us the flexibility of configuring each processor in a different way by accessing their respective copies of the register. The term banking means to have an additional copy. When we say a register is banked it means it has two separate copies one for each processor. Similar is the case with interrupt banking, in such cases it is possible to have interrupts with the same Id and is identified by the combination of Interrupt Id and CPU interface.

## **2.2 Global Timers:**

The Global timer is an important feature from the perspective of this project as the clock functionality from the existing implementations of the tasking framework is ported to this Global timer. This utility has been used to implement the Task Events which require timing information for triggering. The Global timer is a count up timer starting from the value zero. The 64 bit timer is available for access as two 32 bit registers the lower 32 bits and the upper 32 bits. Such a distribution of the timer utility calls for extra precaution to be taken when reading the timer values. One widely accepted way to read such timer registers is the following:

1. Read the upper 32 bit register.
2. Read the lower 32 bit register.
3. Read the upper bit of timer again, if the value is different from step 1 perform step 2 again else the 64 bit value is correct.



The global timer has the following features:

- The global timer is a 64-bit incrementing counter with an auto-incrementing feature. It continues incrementing after sending interrupts.
- The global timer is memory mapped in the private memory region.
- The global timer is accessible to all Cortex-A9 processors in the cluster. Each Cortex-A9 processor has a private 64-bit comparator that is used to assert a private interrupt when the global timer has reached the comparator value. All the Cortex-A9 processors in a design use the banked ID, ID27, for this interrupt. ID27 is sent to the Interrupt Controller as a Private Peripheral Interrupt.

An important property about the Global timer is that it does not stop counting when either of the 2 processors is in debug state unlike the private timers where when the associated processor enters a debug state the private timer stops counting.

The global timer is not started unless we explicitly write into the GLOBAL TIMER CONTROL register located at the address 0xF8F00208. The global timer is clocked at half the CPU frequency. The CPU frequency being 666 MHz the global timer frequency is 333 MHz. The table below shows briefly the associated registers and their addresses to work with Global Timer.

Address	Type	Register Description
0xF8F00200	R/W	Timer Low
0xF8F00204	R/W	Timer High
0xF8F00208	R/W	Timer Control
0xF8F0020C	R/W	Timer Interrupt Control
0xF8F00210	R/W	Comparator Low
0xF8F00214	R/W	Comparator High
0xF8f00218	R/W	Auto increment Register

Table 2.2 (a): Registers for Global Timer access

The “Timer Low” and “Timer High” registers in the above table represents the timer registers which represent the time.

**Timer Control Register:** The bits of this register enable the Global Timer along with other functionalities associated with the timer. A figure showing the explanations of various buts of this register is shown below.

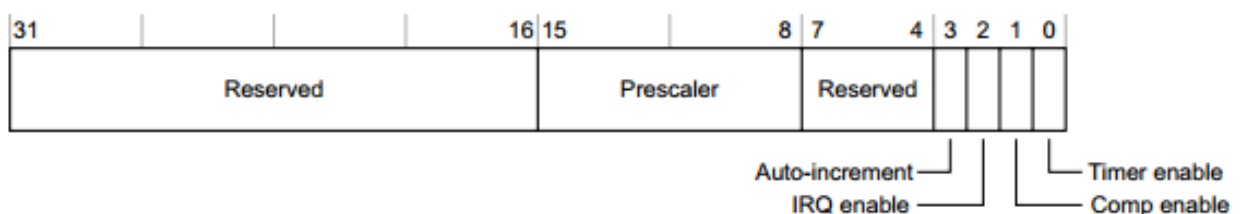


Figure 2.2 (a): Global Timer Control Register [2]

As seen from the picture above the first bit of this register labeled “Timer enable” is used to start the global timer. When this bit is set the counter counts up.

- **Comp Enable:** The timer comes with a feature of comparing the counter values with the values of a comparator register. This is used along with the interrupt option in the global timer. The idea here is that an interrupt is triggered when the counter values are above the values of the comparator registers. In the previous versions the interrupt was triggered when the value was equal to the comparator values. This utility has been extensively used in this project. Besides all the benefits this provides, the usage is prone to errors. The programmer should make sure to increment the values of the comparator once the interrupt is triggered. A failure to do this results in interrupt being fired continuously, as long as the IRQ bit is set or as long as the counter values are higher than the comparator values. This is again a banked bit indicating that comparison is enabled only on that particular register which executes the set command.
- **IRQ Enable:** This bit determines if the interrupt ID27 is enabled or disabled. It is the same interrupt that has been discussed in the above section when the counter value exceeds the comparator value. This is again a banked bit, so the interrupt is fired only on that particular processor in which this bit is set. The other processors are unaffected as long the banked copy of this bit is not set.
- **Auto Increment:** This bit decides the status of the auto increment function of the values. When set the values of comparator are auto-incremented by a preset value which is stored in a different register.
- **Prescaler:** The Prescaler is the means by which we can change the frequency of clock. A higher frequency means that the counter reaches an over flow state quickly. By using the Prescaler functionality we can reduce the frequency at which the counter counts up. A Prescaler is an electronic counting circuit used to reduce a high frequency electrical signal to a lower frequency by integer division.

**Timer Interrupt Status Register:** This is a banked register for all Cortex-A9 processors present. The first bit of this register is an event flag. The event flag is a sticky bit that is automatically set when the Counter Register reaches the Comparator Register value. If the timer interrupt is enabled, Interrupt ID 27 is set as pending in the Interrupt Distributor after the event flag is set. The event flag is cleared when written to 1.

**Comparator registers:** There are 2 comparator registers which hold values against which the counter values are compared to fire an interrupt. Similar to the counter registers there is a comparator low and comparator high register each of which is 32 bits. The comparison is performed only if the Comp Enable bit is set in the timer control register.

**Auto-Increment Register:** While explaining the bits of the Timer control register it was mentioned that care should be exercised when dealing with comparator registers. If the IRQ Enable and Comp Enable bit are set in the Timer Control register then an interrupt is triggered every time the counter value is more than the comparator value. A failure to update the value comparator register results in interrupts being continuously fired. One way to get over this problem is to use the Auto Increment feature provided by the manufacturer. When the Auto-

Increment bit in the Timer Control register is set, the comparator values are incremented by the values of auto-increment register. This operation is performed automatically every time the interrupt occurs. This is a 32 bit register and each processor has a separate copy of Auto-Increment register. This register should be set before starting the timer.

Note: The value of this Auto Increment register cannot be modified in the Interrupt Service Routine. As explained later in this document, a work around had to be implemented to handle this Auto-Increment feature.

## **2.3 Memory:**

The Microzed provides the programmer with the option of DDR (Double Data Rate) memory and OCM (On Chip Memory) to save the data and instructions. It provides us with 1Gb of DDR RAM and the on-chip memory (OCM) module contains 256 KB of RAM and 128 KB of ROM (BootROM). It supports two 64-bit AXI slave interface ports, one dedicated for CPU/ACP access via the APU snoop control unit (SCU), and the other shared by all other bus masters within the processing system (PS) and programmable logic (PL). The BootROM memory is used exclusively by the boot process and is not visible to the user. The access to OCM is faster than the access to DDR memory. The programmer has the flexibility to choose where he wants to place his data by modifying the script file generated along with the project in the Xilinx SDK. According to the Xilinx data sheets the access to OCM is much faster as compared to the DDR memory access. But due to the limited size OCM is used for placing synchronization variables.

All these memory regions are associated with attributes which decide how and who are allowed access to these regions. The board support package (BSP) that is generated with the project has assigned attributes to the memory regions. These can be changed in the project by using the MMU header file and associated functions. Some of the main attributes which play a part in the implementation of this project are

- Shareable and non shareable: As the name suggests, when the attribute is set to shareable, the memory region in question can be accessed by both the cores.
- Cacheable and non cacheable: Setting the attribute as cacheable to a memory region enables the L2 cache to access this region. The Xilinx documentation provided online suggests making the memory region attribute as non-cacheable for synchronization variables. This is performed to address cache coherency issues.

## **CHAPTER 3**

### **Internship process:**

#### **3.1 Step 1: Application development for Linux implementation:**

As mentioned above the existing implementations of tasking framework are implemented on a wide variety of platforms like Linux, RTEMS etc. The initial days of this internship was spent trying to understand the existing implementations. In order to better understand the framework there was a need to work with some example applications which can help a great deal in

understanding. Although each implementation already had a set of basic example these were insufficient.

So as the first step I developed some applications which used the Tasking framework for computations. I had selected the tasking framework implementation using the Linux platform to develop these applications. To date the number of examples/applications developed stands at somewhere around 15. Working with these gave me a better understanding about the API aspect of the Tasking framework and how each of the functionalities included can be utilized for maximizing the efficiency.

We make use of Cmake for compiling these examples on the linux implementation. These examples were written with the idea of making them available and easily understandable for the next person taking this project forward.

## **3.2 Step 2: Getting acquainted to the hardware:**

The hardware used as mentioned above is one that uses the Xilinx's popular Zynq architecture. The presence of a Programmable Logic block in the hardware gives additional flexibility to the designer to migrate some functionality into PL. In this project however it was decided that the PL will not be used to sharing the work load. When starting with a Microzed board the Xilinx provides a set of basic tests and exercises which help in testing as well as giving a brief understanding about the capabilities of the product. These were performed in the initial days of the internship to make sure all the aspects of the board are tested and verified.

One of the first steps involves generating a bitstream file. Now although we are not using the FPGA on the board for any implementation, before we can use the peripherals available on the board we need to configure the FPGA. The configuration is achieved by mapping a bitstream file on the FPGA. This file is generated by using a software package provided by Xilinx called the Vivado. Using the software package we can select the appropriate evaluation board we are working with and then add a block representing the Processing System.

NOTE: Issues were faced using the vivado software package for the SLED 12 systems. It was observed that the vivado version 2015.4 had unresolved bugs for its SLED 12 implementation which did not allow successful creation of the bitstream files.

An image showing the Zynq processing system in the Vivado software is shown below.

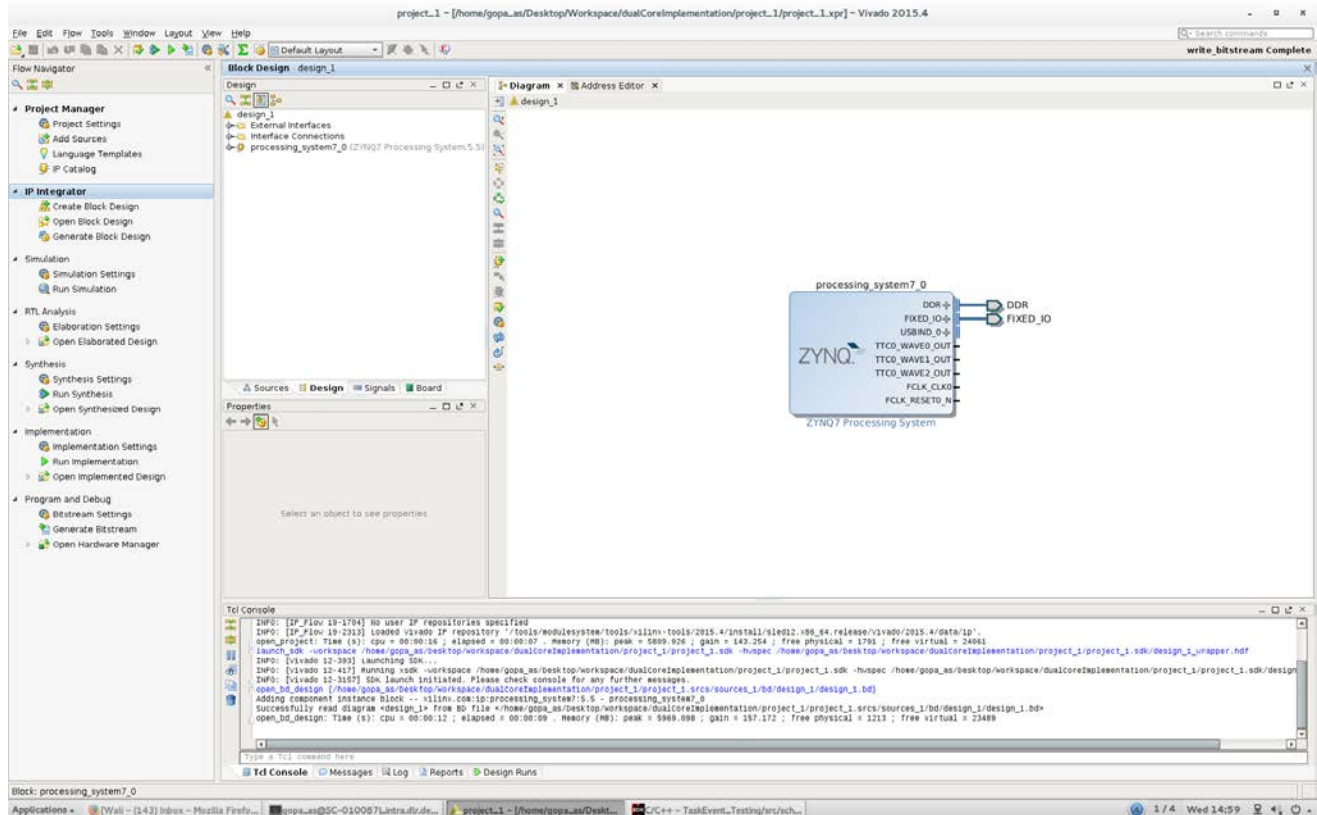


Figure 3.2 (a): GUI of vivado with Zynq PS7

The next step involved exporting the hardware to be made available for software development. Once exported, the Xilinx SDK provides many sample programs to run and explore the device. But before we can run an application on the board we have to set up the First stage Boot loader and a Board support package for our projects. In a normal computer the boot loaders are used to load the operating system into the memory once the system is powered ON. Ideally BIOS runs some set of tests called the Power on Self Tests (POST) after which the control is handed over to a certain region on ROM memory called the Master Boot Record where the boot loader is placed. Now in case of the Microzed board and the Zynq architecture the FSBL is responsible for loading the bitstream and configuring the Zynq processing system at boot time. The procedure to create an FSBL is simple and well documented on the Xilinx manual.

**Board Support Package (BSP):** In a generic definition the Board support package is a piece of software that provides support to the hardware to work with the operating system installed. It provides device drivers, initializes the processor and the RAM etc. Once again Xilinx provides a BSP for standalone applications as standard offering. This BSP can be tweaked and customized for suiting different needs. The BSP provided by Xilinx has seen many changes over the years in the form of enhancements and bug fixes.

Every application project we create will require an FSBL and a BSP to be associated with it. Once these two are done we can commence testing different peripherals of the evaluation board.

One of the first tests Xilinx suggests is the memory tests. The objective is to check the different hardware memory regions that are available. For this purpose the Xilinx already provides an application in their standard set. All the programmer has to do is make an application project and select Memory Test as the template in the sdk. Further details regarding the same can be found on the Xilinx page for microzed. [20]

Once the memory tests are finished and we have obtained a PASS for them, we move ahead to try and using the multiple boot options that are available on the Microzed board. On a practical scenario when working with embedded systems for various application, we may not have a JTAG cable connected with the system in all scenarios. In such case we need to have other mechanism to configure and start the system and run the applications. With the Microzed 7020 we get 2 ways to boot without using the JTAG cable. These form the next steps.

The board provides microSD card and a QSPI (Quad Serial Peripheral Interface) which can be used for booting the system in cases where JTAG is not available. In order to boot from either of these sources we need to prepare a boot image first. The process is more length and cumbersome when compared to using a JTAG cable. When creating the boot image we need to select the following files in the order mentioned:

- FSBL elf
- Bitstream
- Application elf

The order is important and need to be respected strictly. The boot image is made with a “.bin” extension for booting from Micro-SD and a “.mcs” extension for booting from the QSPI option. The position of the jump switches present on the board decide which boot mode is being used. [20]

To boot from the QSPI, keep the Jump switches to cascaded JTAG position, i.e JP1, JP2 and JP3 at 1-2 position. Select the Program Flash option and select the boot image with the appropriate extension. Once the flashing is completed, power off the board and set JP3 to the position 2-3 and leave JP1 and JP2 in 1-2 position. The blue LED should be lit when powered on and hitting the Reset button should start the boot process from QSPI.

For booting from the Micro-SD card, insert the card into a suitable adapter and transfer the boot image with “.bin” extension into the card. It is mandatory that this boot image is named “boot.bin” or “BOOT.bin”. Any other name is unacceptable. So if the file is named something else, it is necessary to change the name to one of these in the card. Once this is done, change the jump switches JP2 and JP3 to 2-3 position and JP1 at 1-2 position. Plugging in the USB cable and hitting the Reset button should start the boot process from the Micro-SD.

### **3.3 Getting acquainted to BareMetal and Clock in the Zynq:**

In the early days of this project, it was decided that the hardware clock present in the board would be used for porting the clocking functionalities. The question however remained as to which clock to be used. In case of the Microzed each processor has its own 32 bit private timer



and a 64 bit global timer. The issue with using private clock is that given only 32 bit length for counting and the high frequency of the clock, an overflow would be reached in 12 seconds with the standard frequency unless a pre-scaling is applied. Another issue with using a private clock is that, the value of the private clock is only visible to the associated processor and not to the other processors. In such conditions it was feared using private timers might create issues and unwanted firing of events when both the processors are employed for load sharing. After a brief deliberation it was decided to use the Global Timer. Although the private timers are more easy to use with readily available header file and associated function, it was decided that the global timer is a much better option.

To understand different aspects associated with the clock and how to use them, simple bare metal applications were developed to exploit various functionalities of the clock. The programs varied from reading the timer values to triggering interrupts when the comparator value was reached. These exercises made the porting of clock class easier. The BSP provides us the header files which are extremely useful when dealing with the hardware resources. Some of the header files which are used as part of the implementation involving global timer are

- xil\_exception.h
- xparameters.h
- xscugic.h

These header files provide us access to certain functions which are used to associate an interrupt service routine to the interrupt associated with the Global timer. These functions tell the processor which service routine is to be executed for which interrupt. They also give us the option of specifying which processor has to run the service routines. Since this is a global timer interrupt, this interrupt can be shared by both the processors. The decision as to which processor handles the interrupt is taken by looking at which processor id is passed to the functions. We also need to initialize the GIC before we start the clock. All the initializations and configurations are to be done before modifying the values in the Global Timer Control register.

The idea behind working with the timer was to try out the different features available on the Global timer. The examples included setting different values for the Prescaler bits in the Global Timer Control register to see the behavior of the timer. One obvious disadvantage of reducing the frequency of the clock ticks is we will have to make a compromise on the resolution of the timer. Working with the standard PERIPHCLK frequency we can measure time up to nanosecond precision, this value will keep deteriorating as we reduce the frequency. After computing the number of days the 64 bit timer would take to reach an overflow when working with the PERIPHCLK frequency it was decided that a prescaling would be unnecessary at this instant.

All these implementations were being performed with only a single core. These also served as first steps in bare metal application development.



### **3.4 Porting on single core:**

Now that I had some experience working with the board and getting familiarized with the functionalities, it was time to move on to performing the necessary changes to the existing implementations to make it possible to have them running on the Microzed board. Some of the instructions to be kept in mind and respected were:

- As mentioned in the previous sections, it was decided not to make any changes to the API of the tasking framework and all associated changes will have to be managed by changing the Scheduler and hardware part of the framework.
- It was also decided not to use the “pthreads” library.
- Initially a semaphore implementation was also not advised when making use multi core.
- The possibility of a lock free implementation was to be considered.

From the perspective of porting almost all the classes implemented in the hardware section of the framework had to be modified. In order to make things more simple and easy to achieve the entire porting was carried out in steps. The idea was to get the entire framework working with a single ARM core on the board and then think of how to utilize the second core and how can the work sharing be achieved.

Some of the above mentioned points which serve as requirements for porting are significant, especially the one with using threads. The existing implementations both on linux and other platforms make use of the POSIX API to achieve multi threading and synchronization between threads. All these implementations were exploiting the context switching utility very efficiently and extensively to achieve parallel operation. At any time of operation a minimum of three threads were being executed with appropriate context switching. It also used synchronization primitives provided by the POSIX API like mutexes and condition variable to wake up a thread and provide access to resources. With the removal of POSIX API the entire idea of thread and its structure would undergo changes. The principle becomes “one core- one thread”.

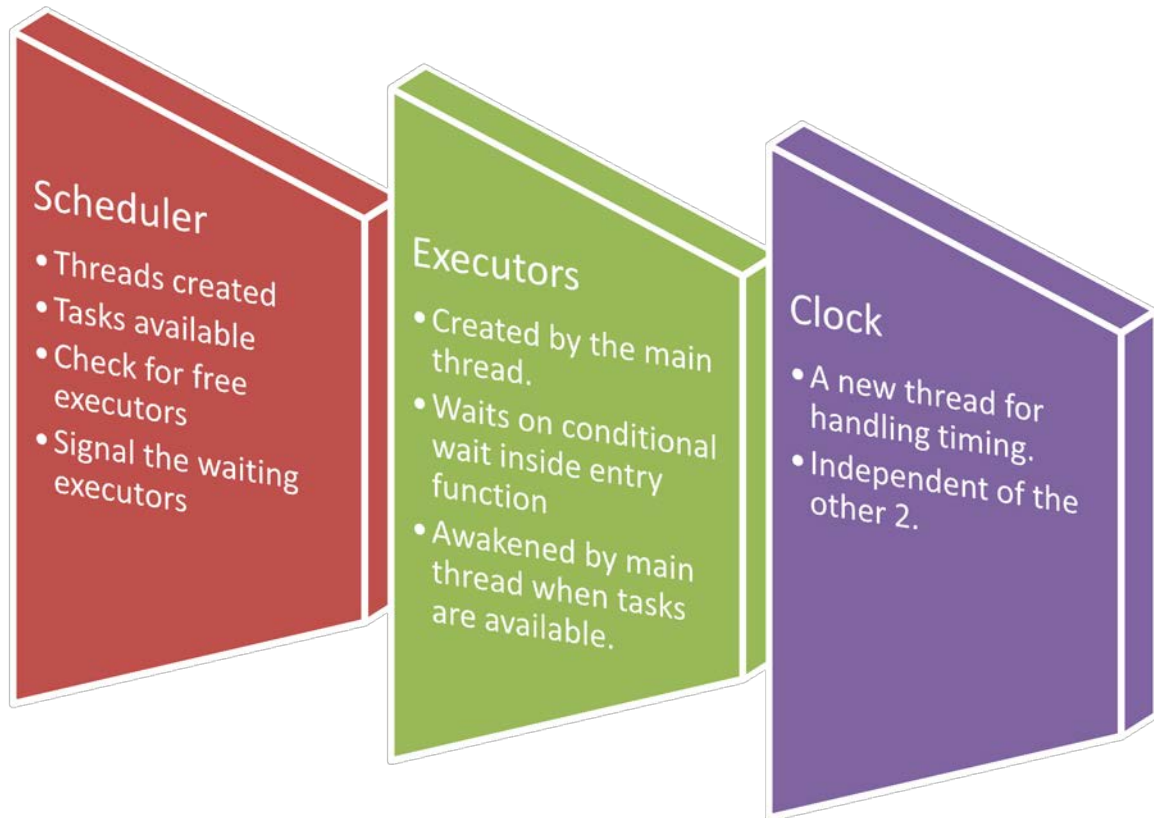


Figure 3.4 (a): Graphical representation of existing implementation

The context switching can be achieved rather easily when we have an operating system running on our hardware to allocate the threads access to the resources like processor time, memory etc. But in the absence of an operating system, having context switches is a painstaking activity and is not a very good idea. A context switching generally involves

1. Suspension of the current thread/process and storing the context of CPU in memory.
2. Reading the context of the next thread/process from the memory and loading them in the CPU registers.
3. Returning to the address at which the process was interrupted and resuming the execution.

Any operating system that supports multitasking can perform these functions rather effortlessly with minimum effort from the application programmer. But performing all the steps mentioned above in case of a bare metal implementation is an expensive affair, and could possibly result in unwanted overheads. [21]

So in the bare metal implementation we have just 1 thread. The one main thread that then invokes the scheduler object. The idea of the executor has been retained and the number of executor objects depends on the number of cores available. The entry function for the thread in the Linux implementation has been retained without the mutexes and condition variable. So

every time there is a task or event which needs to be scheduled this function of the executor class is called.

The Clock is now ported to the Hardware timer. The Global timer set up functions and associated configurations are implemented in the constructor of the clock class. Because the project will have only 1 clock, when the clock object is called the Global Timer is configured to start the counting up sequence. Now since we are using the interrupt and comparator registers of the Global timer, we need to populate the comparator registers to make sure unwanted interrupts are not triggered. So while initializing we also set the Auto Increment register to certain value so that every time the interrupt occurs the comparator registers are updated. Every time we have a TaskEvent, we overwrite the comparator registers by the time of occurrence of the TaskEvent. Once the time has elapsed for the Event the interrupt is fired and the corresponding task associated to that event gets executed. Now one disadvantage of this approach is any attempt to disable or undo the changes of auto increment in the interrupt service routine is futile. This is not supported by the hardware. Hence to undo the changes of auto increment a flag has been introduced which indicates the comparator high register has been modified due to auto incrementing. And this flag is checked later in the program and appropriate actions are taken to address this.

So in the bare metal implementation every TaskEvent whether periodic or onetime event are treated as interrupts. The “clockThread” which was a separate thread in Linux implementation is the service routine for this interrupt.

At this moment in the project we are working with a single core and only one main thread is existent. Given the non-preemptive nature of the framework synchronization is not an issue in the scheduler now. Some classes which were present in the Linux implementation dedicated for synchronization has been omitted thus far and their inclusion will be considered later.

### **3.5 Starting the second core:**

As mentioned in the earlier sections this particular hardware has 2 cores. So far in the project we have used and hosted the application only on a single core, now let us look at how to use both the cores in this project and how to get share the workload for a more efficient performance. When starting up the board the BootROM is the first piece of code to run. The BootROM executes on CPU 0 and CPU 1 executes the wait-for-event (WFE) instruction [1]. The main tasks of the BootROM are to configure the system, copy the Boot Image FSBL/User code from the boot device to the OCM, and then branch the code execution to the OCM. Optionally, the FSBL/User code can be executed directly from a Quad-SPI or NOR device in a non-secure environment. In the context of getting the core1 execute a piece of code it is very much fair to say that the Core0 acts as the master core. CPU 0 is in charge of starting code execution on CPU 1. The BootROM puts CPU 1 into the Wait for Event mode. Nothing has been enabled and only a few general purpose registers have been modified to place it in a state where it is waiting at the WFE instruction. There is a small amount of protocol required for CPU 0 to start an application on CPU1. When CPU 1 receives a system event, it immediately reads the contents of address 0xFFFFFFF0 and jumps to that address. If the SEV is issued prior to updating the destination

address location (0xFFFFFFFF0), CPU 1 continues in the WFE state because 0xFFFFFFFF0 has the address of the WFE instruction as a safety net. If the software that is written to address 0xFFFFFFFF0 is invalid or points to uninitialized memory, results are unpredictable.[1]

The steps for CPU 0 to start an application on CPU 1 are as follows:

1. Write the address of the application for CPU 1 to 0xFFFFFFFF0.
2. Execute the SEV instruction to cause CPU 1 to wake up and jump to the application.

The address range 0xFFFFFE00 to 0xFFFFFFFF0 is reserved and not available for use until the stage 1 or above application is fully functional. Any access to these regions prior to the successful start-up of the second CPU causes unpredictable results.

In the data sheet provided by Xilinx, they mention that in order to make CPU1 jump to the address of its instruction the address must be 32 bit aligned and it must be a valid ARM-32 instruction. For this reason we use special function provided by the “Xil\_io.h” file for writing the address of the code to the address 0xFFFFFFFF0.

To make sure that the “sev” command is executed only after the address is updated we make use of memory barrier statements like “dmb”. This is an additional measure taken to avoid the scenario where the event is fired before the memory is updated. This way we can write a function to be executed on the core 1 and store its address at 0xFFFFFFFF0 for the core1 to start execution.

### **3.6 Multi-Core Processing:**

Now, we know how to get the second core started, we can decide on the type of architecture and implementation we would use for this project. When we have a multi core arrangement there are plenty of options one has to choose from. One of the major points is the kind of multi processing system you want to implement. From the hardware perspective the multi core system are categorized as homogenous system when the cores available have the same architecture and heterogeneous system when they are of different architecture. In the case of this project this has already been decided by the manufacturer of the Microzed board. Both the cores are based on Cortex a9 architecture there by making this a homogenous system.

From the software perspective there are two ways a programmer can make use of the multiple cores available. These are called

- Asymmetric Multi Processing
- Symmetric Multi Processing

Since the names of these shed very little light on the 2 options, the following section describes what does these terms mean from an implementation perspective.

#### **3.6.1 Asymmetric Multi Processing:**

In case of AMP the CPU architecture could be either heterogeneous or homogenous. Each core involved in the AMP arrangement may or may not have operating systems and these need not be

the same. AMP is most likely to be used when different CPU architectures are optimal for specific activities – like a DSP and an MCU [17]. In an AMP system, there is the opportunity to deploy a different OS on each core – e.g. an RTOS and Bare metal (no OS) as befits the required functionality.

So the AMP set up is beneficial and useful when we have a scenario where more than 1 application needs to be processed in parallel. Each core can work on their respective application. A pictorial representation of AMP based systems are shown below.

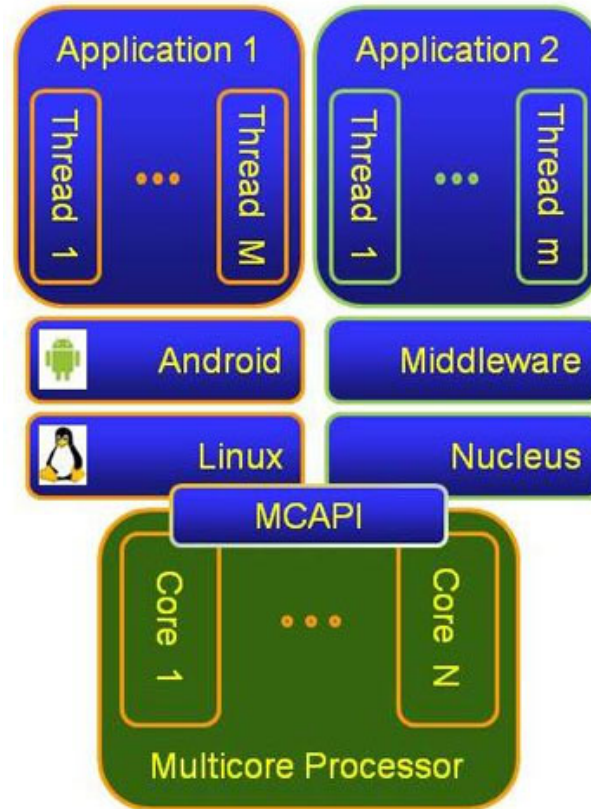


Figure 3.6.1 (a): Pictorial representation of AMP [17]

### **3.6.2 Symmetric Multi-Processing (SMP):**

In case of an SMP architecture it is required that all the cores included have a similar architecture i.e. a homogenous system. SMP architecture is used where more processing power is required to complete a task [5].

In case of Symmetric Multi Processing we have only one copy of the operating system (if we use one) for all the cores which are part of the SMP arrangement. It is the processing of programs by multiple processors that share a common operating system and memory. In symmetric (or "tightly coupled") multiprocessing, the processors share memory and the I/O bus or data path. A single copy of the operating system is in charge of all the processors. SMP, also known as a

"shared everything" system, does not usually exceed 16 processors. A pictorial representation for SMP is shown below.

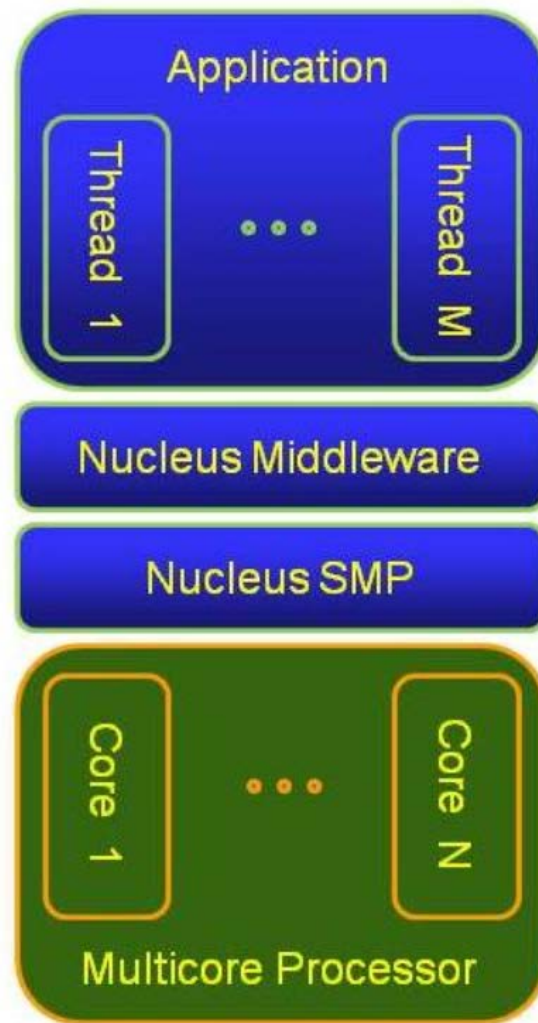


Figure 3.6.2 (a): SMP architecture [17]

As can be seen from the picture above a single application is distributed among multiple cores. All the cores work in parallel to ensure the deadlines are met and the processing is completed on time. Since the cores in SMP access a single image of the operating system, the OS is responsible for achieving parallelism of application. The task partition among different cores is performed by the OS and it also takes care of the resource sharing and task completion and other associated challenges.



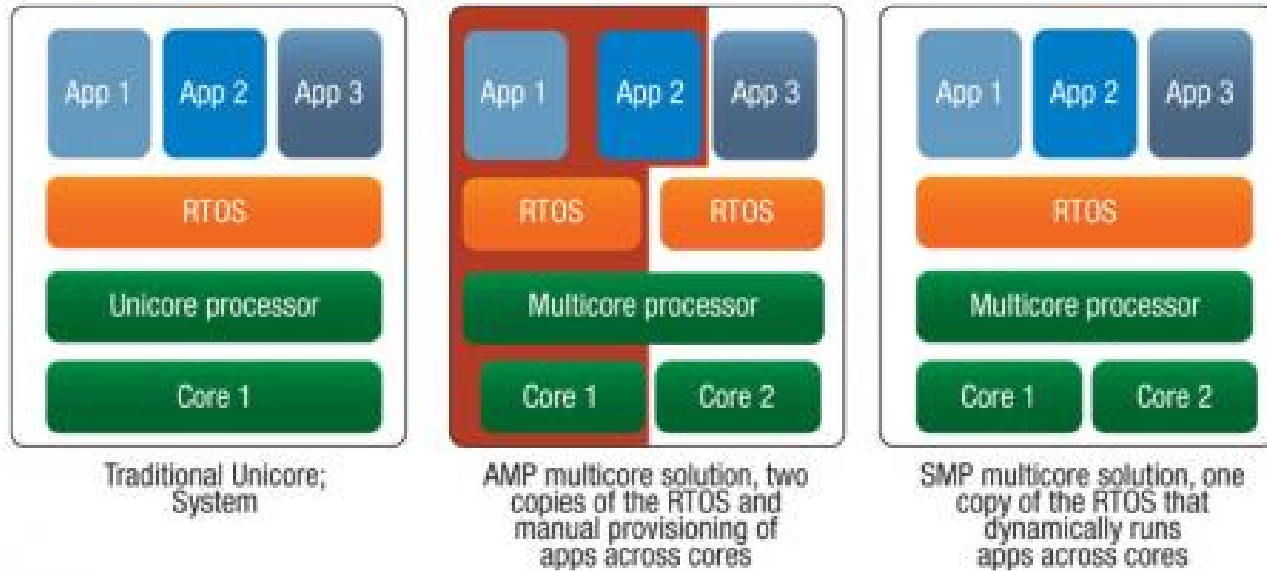


Figure 3.6.2 (b): Comparison of SMP and AMP [9]

The above figure gives a brief comparison between SMP and AMP architectures.

### **3.7 SMP Implementation:**

In the perspective of this project we have homogenous hardware architecture. We also know that the 2 cores have to host only the tasking framework and no other application is decided to be run on them at the moment. So from the definition, the SMP architecture better suits our requirement. So the design is to have both the cores working without any operating system and hosting a single instance of the tasking framework.

The series of projects titled “Microzed Chronicles” by Mr. Adam Taylor [4] was extensively consulted for the implementation. It was discovered that an SMP bare metal was not very widely used on these kinds of boards. Most of the examples were dealing with AMP with 1 OS and bare metal on the other or SMP using LINUX or RTEMS. So in order for the successful completion of the project the examples using AMP with one bare metal and SMP using linux were studied for understanding how to achieve the parallelism and how to implement a synchronization primitive between the 2 cores.

So we have a single binary file for both the processors. Once booted the FSBL starts up core 0 and core 1 is waiting on WFE as explained in the previous section. The idea is to have “one executor object for each core”. So 2 cores account for 2 executor objects which are used while processing a task. Before the core 1 is started there are certain environment settings to be completed. The core 0 is already running and it does all the configurations to start the core 1. These configurations include setting the attributes for certain memory regions which are to be used for implementing synchronization (discussed later), setting up the platform, initialize the variables for inter-processor communication and set up a register for SMP operation.



In this implementation everything related to initializing the Tasking framework and queuing the tasks in the FIFO are done by the core 0. All the requests by tasks to be queued for processing are handled by core 0. Once the initializations are done the core 0 writes the address of the function “cpu1App (void)” to a pointer which is then used to copy the address to the location 0xFFFFFFF0. This function is just a copy of the member function in the executor class. One of the variables used for inter-processor communication is used to indicate if the second core has any task assigned to it. When no task is assigned the core 1 remains in a “while (1)” loop continuously checking for a new task. Each time a new process request is made by a task, core 0 starts looking starts the process. When it reaches a point where an executor is to be assigned to the task, it first reads the value of the variable indicating the status of the second core. If it is busy it proceeds to check if the other executor is free, and if this other executor is found to be free it assigns this executor to core 0 and finishes the computation. If no other executor is available at the moment, the task goes into the list of waiting tasks. If in the first attempt core 1 is found to be available the executor gets assigned to core 1.

Another observation to be made here is the fact that the sharing of workload happens only in the case of tasks which are not associated to any events. This is because only the core 0 is configured to address any interrupt that is generated by the Global Timer. For keeping things simple and easier any task that is associated to an event is scheduled only on core 0. Since tasks associated to events have the highest priority they are at the top of the list and are scheduled before any others.

### **3.8 Data sharing between processor:**

This is one of the biggest challenges faced during the implementation. SMP is a data sharing based architecture. All the processors involved in an SMP should be able to see the same copy of data. Although sharing everything at the first sight looks the easy way, it does not work that way. When multiple processors have access to the same data race conditions can occur. This calls for the use of synchronization between processors when accessing shared data. Cache coherency issues are another common problem that show up when dealing with multiple processors with independent L1 and shared L2 cache. In shared memory architecture, there are multiple copies of the same data. One copy rests in the shared memory region which is seen by all the processors and other in the private cache of each processor. Cache coherency is the policy by which the data consistency is observed across various copies[10]. Any change in the shared copy of the data should be transmitted to the individual caches of the processors in question.

While the synchronization can be achieved by using mutexes or other synchronization primitives, addressing the cache coherency issue was a hurdle. Multiple ways are provided on the Xilinx website to address this. One such way was to set the attributes of the memory region where the data is stored. These memory attributes are already decided and fixed in the board support package (BSP). In order for a programmer to change these attributes, Xilinx provides certain MMU function implementations. Using these function call we can change the attributes of certain memory regions. By using the “lscript” file generated for every project we can place all the data of the project in a particular memory location. Once we know this memory location we can set the attribute of this location to be “shared”. Hardware cache coherency is assured

when the memory is marked as shared. The linker file categorizes instructions and data into different sections depending upon a number of parameters. All the variables are under “.data” section. The global variables are usually placed under a “.bss” section. There are other section names like “.sbss”, “.sdata1” etc. In this project all the data are placed on a region in the DDR RAM and this region is indicated as shared. A snapshot of a sample lscript file is included below.

#### Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_ddr_0_S_AXI_BASEADDR
.init	ps7_ddr_0_S_AXI_BASEADDR
.fini	ps7_ddr_0_S_AXI_BASEADDR
.rodata	ps7_ddr_0_S_AXI_BASEADDR
.rodata1	ps7_ddr_0_S_AXI_BASEADDR
.sdata2	ps7_ram_0_S_AXI_BASEADDR
.sbss2	ps7_ram_0_S_AXI_BASEADDR
.data	ps7_ram_0_S_AXI_BASEADDR
.data1	ps7_ram_0_S_AXI_BASEADDR
.got	ps7_ddr_0_S_AXI_BASEADDR
.ctors	ps7_ddr_0_S_AXI_BASEADDR
.dtors	ps7_ddr_0_S_AXI_BASEADDR
.fixup	ps7_ddr_0_S_AXI_BASEADDR
.eh_frame	ps7_ddr_0_S_AXI_BASEADDR
.eh_frame_hdr	ps7_ddr_0_S_AXI_BASEADDR

Figure 3.8 (a): Snapshot of Configuration file generated in Xilinx SDK.

Another method suggested to address the cache coherency issue was to disable the cache access to a particular part of the memory. This is again achieved by setting the attributes of the memory region. This implementation was tried as well and seemed to be working satisfactorily.

NOTE: Setting the memory attributes is one of those initializations that need to be done by the core0 before starting core1.

### **3.9 Writing SMP configuration to register:**

This was one step that was not explicitly mentioned in the data sheets. This may be because of the fact that SMP in bare metal is not commonly used and when used with an OS, the OS takes care of updating the register in question.

In order to make the two cores on the board work in an SMP configuration we have to configure certain co-processor registers. These are protected registers and the addresses of them are not made available to the programmer. Instead the ARM website provides an assembly statement to modify the value of these co-processor registers. To achieve this inline assembly statements had to be used.

This is part of the initialization performed by core0.

### **3.10 Synchronization primitives:**

This was perhaps the most challenging part of the entire project. As mentioned earlier because this is a “shared everything” approach synchronization is very important to protect the data and prevent any race condition. There are certain projects implemented by Xilinx to demonstrate different features. The “XAPP1078” [7] and “XAPP1079” [6] are 2 such projects which demonstrate the AMP implementation using the Microzed board. These examples were referred for implementing synchronization. These suggest placing a variable on the on chip memory (OCM) and implementing a mutex on it. The region on the OCM where this variable is placed should be marked as non cacheable. Multiple approaches were tried to use which are documented below:

1. **Mutex:** As suggested the variable was placed on the OCM and the memory was made non cacheable. Using the header file “mutex.h” locking and unlocking functions were called. It was observed that when the mutex variable was placed in a non cached memory the lock was never achieved. When the attribute was changed back to cacheable the lock was achieved but cache coherency issues were observed, where in each processor was working with its private copy of the mutex variable.
2. **Atomic instructions:** In an alternate approach atomic instructions were considered to atomically check and store certain variables for implementing a spin lock. Yet again for this to be successful the variable should not be cached. This method also ran into similar trouble as in the previous case where when not cached the atomic statements would never work.
3. **Mutex implementation in Assembly [11]:** The idea of implementing a mutex in assembly language is not a popular option but it was still considered for this project. A simple in line assembly code was developed for mutex implementation. Two of the main instructions in assembly language for implementing exclusive access are the load exclusive LDREX and store exclusive STREX. Yet again LDREX and STREX instructions will work on normal idempotent memory regions and not on un cached memory.
4. **Compiler Intrinsic functions:** This is the currently implemented synchronization primitive and works as expected in uncached and shared memory regions. The function “\_\_sync\_bool\_compare\_and\_swap”. The function compares and swaps the values of a variable as an atomic operation and returns a boolean value depending on the completion of the operation. If the variable had the same value indicated by the second parameter and it was able to swap this value with the third parameter that was passed while calling the

function, the function returns a true else it returns a false. A simple locking mechanism has been implemented using this intrinsic statement.

5. Inter processor interrupts: This is another mechanism that is provided for communication between the two cores. The ARM processors provide a software interrupt for establishing synchronism between the 2 cores. This interrupt can be generated by merely writing into a particular register in the system. It was however unclear how expensive this would be to implement a synchronism mechanism by using interrupts and hence was not considered.

### **3.11 Conclusion:**

The main objective of porting the framework to a bare metal implementation was achieved according to the conditions laid out. The testing of the framework was performed under numerous test cases to understand the behavior in case multiple events occur at the same instant. It was observed the implementation did manage to schedule all the events effectively on both the cores even under such circumstances.

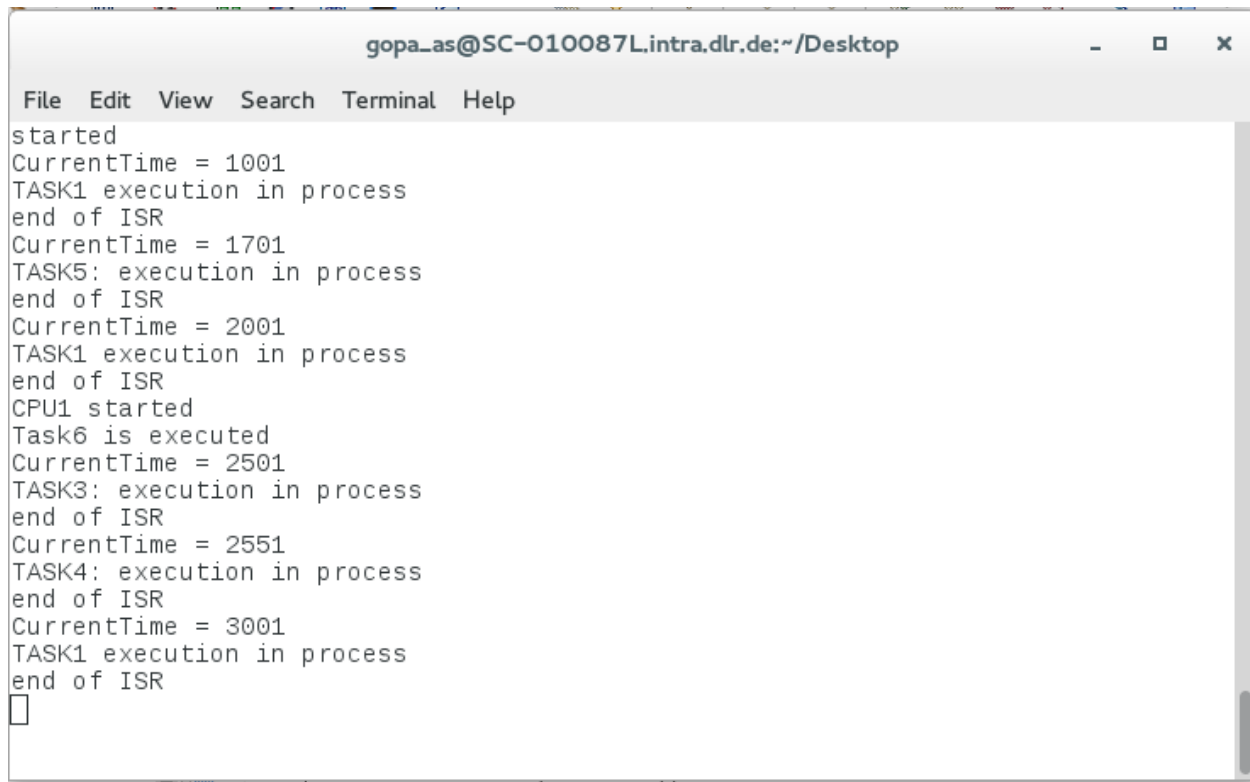
An alternate implementation which involves a non blocking algorithm [13] was proposed, where in the first core does not execute any tasks and only keeps the framework active and all the tasks are scheduled on the second core. The core 1 would simply push all the incoming tasks into the queue and return to address further requests. However a question regarding the effective use of resources was raised for this implementation and hence was not carried out.

Multiple attempts were made to try a different implementation where only one processor had access to the data and the second processor would only execute the tasks as and when a request is raised. But in all these situations cache coherency issues were faced and it was decided that both the processors would have access to the data and it should be synchronized where ever the need arises.

An optional work of implementing a serial interface was initially discussed. It was agreed upon that this would be an additional work and would be completed if there was any time after the porting was completed. This task could not be completed and hence could be included in the list of future tasks.

### **3.12 Results:**

A picture showing a terminal window with the outputs from the bare metal implementation is included below. When each task is executed it prints its completion status to the console indicating the completion.



```

gopa_as@SC-010087L.intra.dlr.de:~/Desktop
File Edit View Search Terminal Help
started
CurrentTime = 1001
TASK1 execution in process
end of ISR
CurrentTime = 1701
TASK5: execution in process
end of ISR
CurrentTime = 2001
TASK1 execution in process
end of ISR
CPU1 started
Task6 is executed
CurrentTime = 2501
TASK3: execution in process
end of ISR
CurrentTime = 2551
TASK4: execution in process
end of ISR
CurrentTime = 3001
TASK1 execution in process
end of ISR

```

Figure3.12 (a): Minicom window with output

The output above shows a test scenario where 2 periodic tasks, 3 one time trigger events and 1 recurring task not associated to an event where scheduled. The task named “Task 6” is the independent task not associated to any event all the other tasks are associated to an event. As explained in the implementation part all the tasks associated with an event are processed in the ISR and scheduled on Core 0. The core 1 executes only the independent tasks.

Multiple test cases were tried with a different combination of tasks and events in order to check for race conditions and data consistency. All of them tried so far have returned good results. The snapshot included above is from the more recent test scenarios.

## **REFERENCES:**

- [1] Technical Manual of Zynq 7000 All programmable SoC  
[http://www.xilinx.com/support/documentation/user\\_guides/ug585-Zynq-7000-TRM.pdf](http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf).
- [2] ARM-Cortex A9 MP-core Technical Reference Manual  
[http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407g/DDI0407G\\_cortex\\_a9\\_mpcore\\_r3p0\\_trm.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407g/DDI0407G_cortex_a9_mpcore_r3p0_trm.pdf)
- [3] ARM-Cortex A9 Technical Reference Manual  
[http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388g/DDI0388G\\_cortex\\_a9\\_r3p0\\_trm.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388g/DDI0388G_cortex_a9_r3p0_trm.pdf)
- [4] Microzed Chronicles by Adam Taylor  
Website: <http://zedboard.org/content/microzed-chronicles>
- [5] SMP description  
[https://en.wikipedia.org/wiki/Symmetric\\_multiprocessing](https://en.wikipedia.org/wiki/Symmetric_multiprocessing)
- [6] AMP on bare metal system  
[https://www.xilinx.com/support/documentation/application\\_notes/xapp1079-amp-bare-metal-cortex-a9.pdf](https://www.xilinx.com/support/documentation/application_notes/xapp1079-amp-bare-metal-cortex-a9.pdf)
- [7] AMP using Linux and bare metal  
[https://www.xilinx.com/support/documentation/application\\_notes/xapp1078-amp-linux-bare-metal.pdf](https://www.xilinx.com/support/documentation/application_notes/xapp1078-amp-linux-bare-metal.pdf)
- [8] Zedboard support forums  
<http://zedboard.org/forums/zed-english-forum>
- [9] AMP and SMP  
<http://rtcmagazine.com/articles/view/101663>
- [10] Cache coherency description  
[http://www.webopedia.com/TERM/C/cache\\_coherence.html](http://www.webopedia.com/TERM/C/cache_coherence.html)
- [11] Assembly Mutex  
<http://stackoverflow.com/questions/29783951/can-i-use-ldrex-strex-to-implement-a-spin-lock-without-enabling-scu-in-a-multico>
- [12] Vivado support  
<https://forums.xilinx.com/t5/Embedded-Development-Tools/XAPP1079-in-Vivado-2015-4/td-p/678257>
- [13] Gabor Drescher, Wolfgang Schroder-Preikschat: Guarded Sections: Structuring Aid for Wait-Free Synchronization. 2015 IEEE 18th International Symposium on Real-Time Distributed Computing

[14] Using the Interrupts on Zynq

[https://www.xilinx.com/support/documentation/xcell\\_articles/how-to-use-interrupts-on-zynqsoc.pdf](https://www.xilinx.com/support/documentation/xcell_articles/how-to-use-interrupts-on-zynqsoc.pdf)

[15] Embedded System timers.

<http://www.scriptoriumdesigns.com/embedded/timers.php>

[16] Critical sections on ARM cortex A9

<http://electronics.stackexchange.com/questions/141673/how-to-implement-critical-sections-on-arm-cortex-a9>

[17] Multi-Processing

<http://www.embedded.com/design/mcus-processors-and-socs/4429496/Multicore-basics>

[18] Microzed 7020

<http://www.cnx-software.com/wp-content/uploads/2013/08/microZED.jpg>

[19] DLR-SC

[http://www.dlr.de/sc/en/desktopdefault.aspx/tabid-1185/1634\\_read-3062/](http://www.dlr.de/sc/en/desktopdefault.aspx/tabid-1185/1634_read-3062/)

[20] Microzed using Vivado

<http://zedboard.org/support/design/1519/10>

[21] Context switching

[http://www.lininfo.org/context\\_switch.html](http://www.lininfo.org/context_switch.html)



## APPENDIX

The code snippets for the hardware layer are provided below for reference.

The Scheduler Class:

```
/*
 * scheduler.cpp
 *
 * Author: Gopal Ashish
 * Date: 05 September, 2016
 *
 * \brief
 * Implementation of the execution model for XILINX board
 */

#include <stdio.h>
#include "scheduler.h"
#include "chain.h"
#include "task.h"
#include "taskEvent.h"
#include "clock.h"
#include <iostream>
#include <cstdint>
//directives for hardware and dual core implementations

#include "sleep.h"
#include "xil_io.h"
#include "xparameters.h"
#include "xil_mmu.h"
#include "platform.h"
#include "xreg_cortexa9.h"

#define sev() __asm__("sev")

//-----Synchronisation Variables-----//

#define SMP_CONFIGURE (*(volatile uint32_t*) 0xFFFF0000) //for configuring
the 2nd core for SMP
#define UNLOCKED 0
#define LOCKED 1

namespace Tasking {
    Scheduler scheduler;
    Scheduler::ExecutorThread Scheduler::m_executors[USED_CORES];
    extern Clock clock;
}
```

```
volatile int* mutex = (volatile int*) 0x4000000; //ending address of DDR
bool Lock(volatile int* lock);
```

```
//forward declaring the global functions
void core2Initialisation();
```

```
//-----end of forward declaration-----//
```

```
void (*ptrCPU1APP)(void);
typedef struct{
    int secondCoreBusy;
    void* executoraddrs;
} commBlock;
```

```
commBlock* communication;
```

```
//-----Memory attributes-----//
#define SHAREABLE (1<<16)
#define SECTION(attributes) ((attributes) | 0x2)
```

```
u32 properties = SECTION(SHAREABLE);
```

```
//-----
```

```
void Tasking::Scheduler::ExecutorThread::run(void) {
    running = true;

    ExecutorThread* data = this;

    if((communication->secondCoreBusy == 0) && scheduler.eventHandling ==
false)
    {
        communication->secondCoreBusy = 1;
        secCore = true;
        communication->executoraddrs = this;
    }
    else
    {
        do{

            scheduler.callExecution(data);
            while(!Lock(mutex));
            scheduler.putToPool(static_cast<ChainObject*>(this-
>executableObject));
            this->executableObject =
static_cast<Scheduler::ExecutionObject*>(scheduler.getNext());
```

```

        *mutex = UNLOCKED;

        } while(executableObject != NULL && running);

    }

}

//-----

void Tasking::Scheduler::callExecution(Tasking::Scheduler::ExecutorThread*
threadDetails)
{
    switch (threadDetails->executableObject->type) {    //changed here
        case Scheduler::EXECUTION_TYPE_TASK: {
            Task* task = static_cast<Task*>(threadDetails->executableObject-
>executableObject);
            task->setPriority(task->m_priority);
            task->synchronizeStart();
            task->execute();
            task->synchronizeEnd();
            task->markAsExecuted();
            task->setPriority(EMPTY_EXECUTOR_PRIORITY);
        }
        break;
        case Scheduler::EXECUTION_TYPE_TASK_APERIODIC: {
            Task* task = static_cast<Task*>(threadDetails->executableObject-
>executableObject);
            task->setPriority(task->m_priority);
            task->synchronizeStart();
            task->execute();
            task->synchronizeEnd();
            task->reset();
            //task->setPriority(EMPTY_EXECUTOR_PRIORITY);

        }
        break;
        case Scheduler::EXECUTION_TYPE_TASK_EVENT:{
            TaskEvent* event = static_cast<TaskEvent*>(threadDetails-
>executableObject->executableObject);
            event->handle();
        }
        break;
    }
}

//-----

```

```

Tasking::Scheduler::ExecutorThread::ExecutorThread(void) :
    running(false), executableObject(NULL), secCore(false) {
    // Nothing else to do
}

//-----

Tasking::Scheduler::ExecutorThread::~~ExecutorThread(void) {
    // Nothing else to do
}

//-----

Tasking::Scheduler::Scheduler(void) :
    m_currentThread(0), m_stopp(true), m_queue(), eventHandling(false) {
    // Fill the object pool
    for (unsigned int i = 0; i < USED_CORES + MAX_WAIT_QUEUE_LENGTH; i++) {
        m_objectPool.enqueue(&(m_objectMemory[i]));
    }

    Xil_SetTlbAttributes((INTPTR)(mutex), properties);
    *mutex = UNLOCKED;
}

//-----

void Tasking::Scheduler::startExecution(void) {
    m_stopp = false;
    //for core 2
    core2Initialisation();
}

//-----

void Tasking::Scheduler::terminateExecution(void) {
    m_stopp = true;
    while (m_queue.getHead() != NULL) {
        m_queue.dequeue();
    }
}

//-----

void Tasking::Scheduler::initialize(void) {
    Task* head = getAndSetTaskList(NULL);
    getAndSetTaskList(head);
    for (Task* i = head; i != NULL; i = i->m_nextTask) {
        i->initialize();
    }
}

```

```

}

//-----

void Tasking::Scheduler::reset(void) {
    Task* head = getAndSetTaskList(NULL);
    getAndSetTaskList(head);
    for (Task* i = head; i != NULL; i = i->m_nextTask) {
        i->reset();
    }
}

//-----

Tasking::Task* Tasking::Scheduler::getAndSetTaskList(Task* task) {
    static Task* head = NULL;
    Task* result = head;
    head = task;
    return result;
}

//-----

void Tasking::Scheduler::perform(ExecutionObject* p_object) {
    // Check if not stopped
    if (!m_stopp) {
        //for synchronism purposes only
        if (communication->secondCoreBusy == 1)
        {
            if (communication->executoraddrs ==
&m_executors[m_currentThread])
                m_currentThread = (m_currentThread + 1) % USED_CORES;

        }

        else{

            for(int Search = USED_CORES; Search > 0;)
            {
                if(m_executors[m_currentThread].executableObject ==
NULL && m_executors[m_currentThread].secCore == false)
                    Search = 0;
                else
                {
                    m_currentThread = (m_currentThread + 1) %
USED_CORES;
                    Search--;
                }
            }
        }
    }
}

```

```

    }
}

    if (m_executors[m_currentThread].executableObject == NULL
&& m_executors[m_currentThread].secCore == false) {
        m_executors[m_currentThread].executableObject =
p_object;
        m_executors[m_currentThread].run();
    }
    else {
        while(!Lock(mutex));
        m_queue.enqueue(p_object);
        *mutex = UNLOCKED;
    }

}

}

//-----

void Tasking::Scheduler::perform(Tasking::Task* p_task) {
    while(!Lock(mutex));
    ExecutionObject* executionObject =
static_cast<ExecutionObject*>(m_objectPool.getHead());
    if (executionObject == NULL) {
        executionObject = new ExecutionObject;
    } else {
        m_objectPool.dequeue();
    }
    *mutex = UNLOCKED;
    executionObject->executableObject = p_task;
    if (p_task->isAperiodic()) {
        executionObject->type = EXECUTION_TYPE_TASK_APERIODIC;
    } else {
        executionObject->type = EXECUTION_TYPE_TASK;
    }
    executionObject->setPriority(p_task->m_priority);
    perform(executionObject);
}

//-----

void Tasking::Scheduler::perform(Tasking::TaskEvent* p_event) {
    uint64_t time = 0;
    do
    {

```

```

        while(!Lock(mutex));
        time = (uint64_t)clock.getHead()->m_start;
        p_event = clock.consumeHead()->event;

        ExecutionObject* executionObject =
static_cast<ExecutionObject*>(m_objectPool.getHead());
        if (executionObject == NULL) {
            executionObject = new ExecutionObject;
        } else {
            m_objectPool.dequeue();
        }
        *mutex = UNLOCKED;
        executionObject->executableObject = p_event;
        executionObject->type = EXECUTION_TYPE_TASK_EVENT;
        executionObject->setPriority(999u);
        perform(executionObject);

    }while(time == (uint64_t)clock.getHead()->m_start);
}

//-----

Tasking::ChainObject* Tasking::Scheduler::getNext(void) {
    ChainObject* result = NULL;
    if (m_queue.getHead() != NULL) {
        result = m_queue.getHead();
        m_queue.dequeue();
    }
    return result;
}

//-----

void Tasking::Scheduler::putToPool(Tasking::ChainObject* p_object) {
    m_objectPool.enqueue(p_object);
}

//-----

void Tasking::Scheduler::waitUntilEmpty(void) {
    bool someoneRunning = false;
    do {
        // Check if at least one executor thread is running or someone is waiting
        while(!Lock(mutex));
        someoneRunning = (m_queue.getHead() != NULL); // Someone is waiting
        // Check on running executor threads
        for (unsigned int i = 0; (i < USED_CORES) && !someoneRunning; i++) {

```



```

        if (m_executors[i].executableObject != NULL) {
            someoneRunning = true;
        }
    }
    *mutex = UNLOCKED;
    // If one of them is running sleep until wake up

} while (someoneRunning);

}

//-----
uint8_t Tasking::Scheduler::getCpuAffinity(void)
{
    asm volatile("MRC " XREG_CP15_MULTI_PROC_AFFINITY " ;"
                 : "=r" (this->cpuAffin)
                 );
    return this->cpuAffin;
    return 0;
}
//-----

void Tasking::Scheduler::cpu1App(void)
{
    init_platform();
    Tasking::Scheduler::ExecutorThread* instance;
    while(1)
    {
        if(communication->secondCoreBusy == 1 )
        {
            printf("CPU1 started\n");
            instance =
reinterpret_cast<Tasking::Scheduler::ExecutorThread*>(communication-
>executoraddr);
            while(instance->executableObject != NULL and instance-
>running)
            {

                scheduler.callExecution(instance);
                while(!Lock(mutex));

                scheduler.putToPool(reinterpret_cast<ChainObject*>(instance-
>executableObject));
                instance->executableObject =
reinterpret_cast<Scheduler::ExecutionObject*>(scheduler.getNext());
                *mutex = UNLOCKED;
            }
            instance->secCore = false;
            communication->executoraddr = NULL;
        }
    }
}

```

```

    }
    communication->secondCoreBusy = 0;

    sleep(1);
}

cleanup_platform();
}

//-----

void core2Initialisation()
{
    Xil_SetTlbAttributes(0xFFFF0000,0x14de2);
    dmb();
    Xil_SetTlbAttributes((INTPTR)(communication),properties);
    Xil_SetTlbAttributes(0x100000,properties);
    communication->secondCoreBusy = 0;
    communication->executoraddrs = NULL;
    ptrCPU1APP = Tasking::Scheduler::cpu1App;
    Xil_Out32(0xFFFFFFFF0,(u32)ptrCPU1APP);
    dmb();
    sev();
    SMP_CONFIGURE = 71;
    asm volatile("MCR p15, 0,%0, c1, c0, 1;"
                :
                : "r"(SMP_CONFIGURE)
                );
}

//-----

bool Lock(volatile int* lock)
{
    bool result;
    result = __sync_bool_compare_and_swap(mutex, UNLOCKED, LOCKED);
    return result;
}

//-----

*****End of Scheduler Class*****

```

## The Clock Class:

```

/*
 * XilinxClock.cpp
 *
 * Created on: Aug 5, 2016
 * Author: Gopal Ashish
 */

#include <stdio.h>
#include <time.h>
#include <inttypes.h>
#include "stdint.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xtime_l.h"
#include <math.h>    //for fmod used in startAt and startIn

#include "clock.h"
#include "scheduler.h"
#include "taskEvent.h"

// #define GLOBAL_TIMER_START ((volatile uint32_t*) 0xF8F00200)
#define GLOBAL_TIMER_CTRL ((volatile uint32_t*) 0xF8F00208)
#define GLOBAL_TIMER_LOW ((volatile uint32_t*) 0xF8F00200)
#define GLOBAL_TIMER_HIGH ((volatile uint32_t*) 0xF8F00204)
#define GLOBAL_TIMER_COMPLow ((volatile uint32_t*) 0xF8F00210)
#define GLOBAL_TIMER_COMPHigh ((volatile uint32_t*) 0xF8F00214)
#define GLOBAL_TIMER_AUTOINCR ((volatile uint32_t*) 0xF8F00218)
#define GLOBAL_TIMER_IRQ ((volatile uint32_t*) 0xF8F0020C)

#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#define GT_INTERRUPT_ID XPS_GLOBAL_TMR_INT_ID
#define GT_DEVICE_ID XPAR_GLOBAL_TMR_DEVICE_ID

#define GPTIMER_ENABLE    (1<<0)
#define GPTIMER_COMP_EN   (1<<1)
#define GPTIMER_IRQ_EN    (1<<2)
#define GPTIMER_AUTO_INC  (1<<3)

XScuGic_Config *IntcConfig;
static XScuGic Intc;

uint32_t prevCompHi = 0 ;
uint32_t prevCompLow = UINT32_MAX;

```

```

int preDiv= 0;
int div = 0;
bool comphiReplaced = false;

double frequency = 333000.0 ; //when time is expressed in milliSec frequency
value is 333000

namespace Tasking {
Clock clock;
extern Scheduler scheduler;
}

//****ISR for handling the clock interrupts****//

Xil_ExceptionHandler Tasking::clockThread(void* p_clock) {
    Clock* clock = (Clock*) p_clock;
    scheduler.eventHandling = true;

    //to account for the update by autoincrement
    if (prevCompHi != *GLOBAL_TIMER_COMPHIGH)
    {
        comphiReplaced = true;
    }
    //end of autoincrement handling

    printf("CurrentTime = %d\n", (int)clock->getTime());
    *GLOBAL_TIMER_IRQ = 1;
    uint64_t timeOfNext = clock->queue->next->m_start;

    scheduler.perform(clock->getHead()->event);

    if(clock->queue->m_start == timeOfNext)
    {
        Clock::EventQueueElement* next = clock->queue;
        *GLOBAL_TIMER_COMPLow= (uint32_t)(fmod((next->m_start) *
(uint32_t)frequency, UINT32_MAX));

        div = ((next->m_start) * (int)frequency)/UINT32_MAX;

        if((div > preDiv) && (*GLOBAL_TIMER_COMPLow <= prevCompLow))
        {
            ++(*GLOBAL_TIMER_COMPHIGH);
        }

        if(comphiReplaced)
        {
            (*GLOBAL_TIMER_COMPHIGH)--;
            comphiReplaced = false;
        }
    }
}

```

```

    }

    preDiv = div;

    prevCompLow = *GLOBAL_TIMER_COMPLow;
    prevCompHi = *GLOBAL_TIMER_COMPHIGH;
}

scheduler.eventHandling = false;
printf("end of ISR\n");
return NULL;

}

//*****end of ISR*****//

//*****interrupt initialisation and configuration*****//

void interrupt_init(void* clockObject)
{
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);

    XScuGic_CfgInitialize(&Intc,IntcConfig, IntcConfig->CpuBaseAddress);

    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
(Xil_ExceptionHandler) XScuGic_InterruptHandler, &Intc);
    Xil_ExceptionEnable();

    XScuGic_Connect(&Intc,GT_INTERRUPT_ID
,(Xil_InterruptHandler)Tasking::clockThread,(void*)clockObject);

    XScuGic_Enable(&Intc, GT_INTERRUPT_ID);
}

//-----

Tasking::Clock::Clock(void) :running(false), queue(NULL), empty(NULL) {
    *GLOBAL_TIMER_COMPLow = UINT32_MAX;
    *GLOBAL_TIMER_COMPHIGH = 0;

    interrupt_init(this);
    *GLOBAL_TIMER_CTRL =
GPTIMER_ENABLE|GPTIMER_COMP_EN|GPTIMER_AUTO_INC|GPTIMER_IRQ_EN; //STARTING
THE TIMER
}

```

```
//-----

Tasking::Clock::~Clock(void) {

    running = false;
    while (queue != NULL) {
        EventQueueElement* element = queue;
        queue = queue->next;
        delete element;
    }

    while (empty != NULL) {
        EventQueueElement* element = empty;
        empty = empty->next;
        delete element;
    }
}

//-----

void Tasking::Clock::startAt(TaskEvent* p_event, const uint64_t time) {
    uint32_t compLowVal = 0;
    uint64_t timeInMs;

    EventQueueElement* element = NULL;
    if (empty != NULL) {
        element = empty;
        empty = empty->next;
    } else {
        element = new EventQueueElement;
    }

    timeInMs = (uint64_t)getTime();
    element->m_start = time;

    element->event = p_event;
    if (enqueue(element)) {

        compLowVal =(uint32_t) (fmod((element->m_start) *
(uint32_t)frequency, UINT32_MAX));
        *GLOBAL_TIMER_COMPLow = compLowVal;
        div = ((element->m_start) * frequency)/UINT32_MAX;
        if(div > preDiv && (*GLOBAL_TIMER_COMPLow < prevCompLow))
        {
            //if(((div == preDiv) && (*GLOBAL_TIMER_COMPLow >
prevCompLow)) || ((div > preDiv) && (*GLOBAL_TIMER_COMPLow < prevCompLow)) )
                ++(*GLOBAL_TIMER_COMPHIGH);
        }
    }
}
```

```

        if(comphiReplaced)
        {
            (*GLOBAL_TIMER_COMPHIGH)--;
            comphiReplaced = false;
        }

        *GLOBAL_TIMER_AUTOINCR = (uint32_t)((element->m_start -
timeInMs) * (uint32_t)frequency);

        prevCompHi = *GLOBAL_TIMER_COMPHIGH;
        preDiv = div;
        prevCompLow = *GLOBAL_TIMER_COMPLow;

    }

}

//-----

void Tasking::Clock::startIn(TaskEvent* p_event, const uint64_t time) {
    EventQueueElement* element = NULL;
    uint32_t compvalStartIn;
    if (empty != NULL) {
        element = empty;
        empty = empty->next;
    } else {
        element = new EventQueueElement;
    }

    element->m_start = getTime(); //time in mseconds
    element->m_start += time ; //time in mseconds

    element->event = p_event;
    if (enqueue(element)) {
        compvalStartIn = (uint32_t)((element->m_start *
(uint32_t)frequency) % UINT32_MAX);
        *GLOBAL_TIMER_COMPLow = compvalStartIn;
        div = (element->m_start * frequency)/UINT32_MAX;
        if (div >= preDiv && compvalStartIn > prevCompLow)
            ++(*GLOBAL_TIMER_COMPHIGH);

    }
    preDiv = div;
    prevCompLow = *GLOBAL_TIMER_COMPLow;
    prevCompHi = *GLOBAL_TIMER_COMPHIGH;
}

```



//-----

```
bool Tasking::Clock::enqueue(EventQueueElement* p_element) {
    EventQueueElement* previous = NULL;
    EventQueueElement* current = NULL;
    current = queue;

    bool headReplaced = false;
    bool search = true;

    if (queue == NULL) {
        queue = p_element;
        p_element->next = NULL;
        headReplaced = true;
    } else {
        while ((search) && (current != NULL)) {
            if (current->m_start >= p_element->m_start) {
                search = false;
            }

            if (search) {
                previous = current;
                current = current->next;
            }
        }

        if (previous != NULL) {
            previous->next = p_element;
            p_element->next = current;
        } else {
            headReplaced = true;
            p_element->next = current;
            queue = p_element;
        }
    }

    return headReplaced;
}
```

//-----

```
void Tasking::Clock::dequeue(const TaskEvent* p_event) {
    EventQueueElement* current;
    if (queue != NULL) {
        if (p_event == NULL) {
            while (queue != NULL) {
                current = queue;
                queue = queue->next;
            }
        }
    }
}
```

```

        current->next = empty;
        empty = current;
    }

} else {
    if (queue->event == p_event) {
        current = queue;
        current->next = empty;
        empty = current;
        queue = queue->next;

    } else {
        current = queue;
        while (current != NULL) {
            if (current->next != NULL) {
                if (current->next->event == p_event) {
                    EventQueueElement* toBeDequeued =
current->next;

                    current->next = toBeDequeued->next;
                    toBeDequeued->next = empty;
                    empty = toBeDequeued;
                    current = NULL;
                } else {
                    current = current->next;
                }
            } else {
                current = NULL;
            }
        }
    }
}

}

}

}

}

}

}

//-----
Tasking::Clock::EventQueueElement* Tasking::Clock::getHead(void) {
    return queue;
}

//-----

Tasking::Clock::EventQueueElement* Tasking::Clock::consumeHead(void) {
    EventQueueElement* result = queue;
    if (result != NULL) {
        queue = queue->next;
        result->next = empty;
        empty = result;
    }
    return result;
}

```

```
//-----
```

```
double Tasking::Clock::getTime(void) const {  
    uint32_t timerHigh = *GLOBAL_TIMER_HIGH;  
    uint32_t timerLow = *GLOBAL_TIMER_LOW;  
    uint64_t currentTime = (uint64_t) timerHigh<<32|timerLow;  
    return (double)currentTime/frequency; //returns time in mS  
}
```

```
*****End of Clock Class*****
```