SUMMER PROJECT
(Semester - 4)
DRDO, Bangalore
Real Time Executive for Avionics System

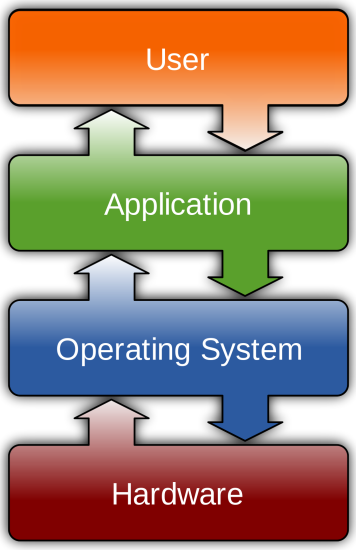
Shubhangi Ghosh
EE15B129
Department of Electrical Engineering

October 31, 2018

1 ABSTRACT:

This report presents the development of a Real Time Executive for a Avionics System, which comprises of functionality for Interrupt Handling, Task Scheduling and Context Switching. The developed scheduler will be further used as a part of an indigeneos RTOS(Real Time Operating Software), for a DRDO(Defence Research & Development Organisation) avionics project with criticality level A. The objective of this effort has been to reduce reliance on foreign RTOS, such as Integrity and VXWorks, and also processor and memory overheads due to extraneous functions provided by these softwares. We have been working to develop a tailored Operating Software to cater to the specific needs of the above-mentioned project.

2 INTRODUCTION:



The project involved working in the Operating System layer of code, implementing some Kernel programming functionalities.
A basic task scheduler for e500v2 core was built, which:

1. Switches context from a continuously running **Background Task** to a short **Foreground Task**, through an **ISR(Interrupt Service Routine)**, which services periodic **external FPGA timer interrupts**.

2. Restores context after completion of **Foreground Task**, back to **Background Task**.

A Real Time Executive for a Avionics System was implemented, which includes functionality for Interrupt Handling, Task Scheduling and Context Switching. Some specifics about the functionality include:

1. It was tested on a p1015 processor based custom board.

2. It works as a Priority based **preemptive** scheduler, which context of the current task is saved when interrupted, and restored after interrupt is handled.

3. Task switching occurs in the event of a specific(**FPGA timer**) external interrupt.

4. Core should be halted if none of the tasks are executing.

3 PROJECT DETAILS:

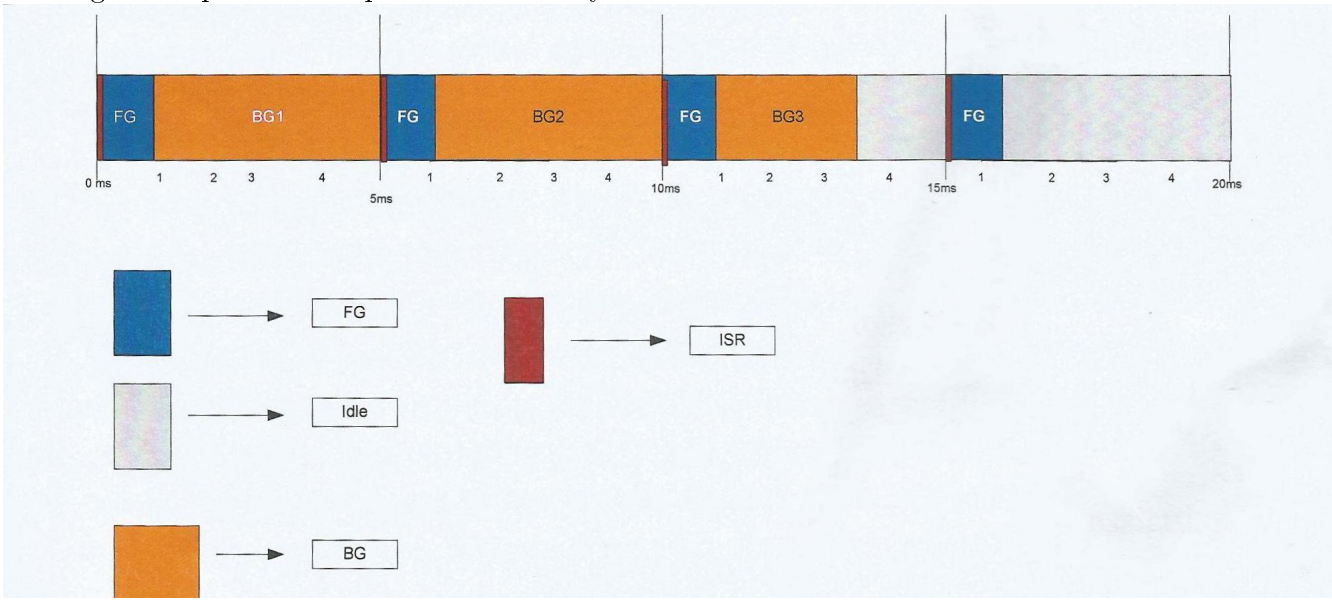
3.1 PROJECT REQUIREMENTS:

Basic requirement of the project framework is to provide support for:

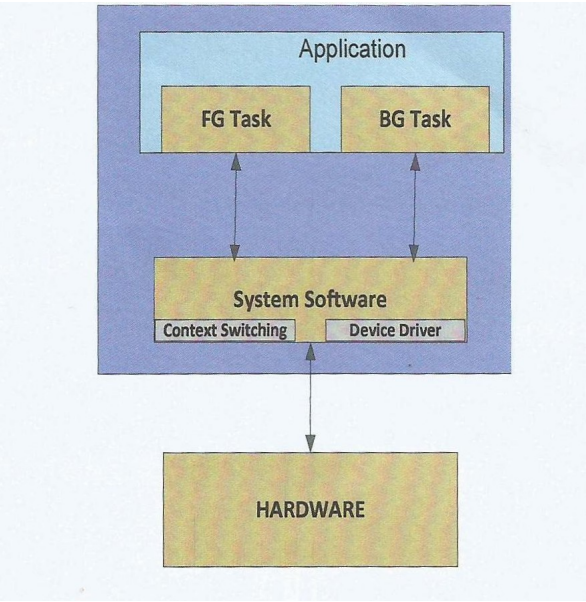
1. Perform low frequency performance-checking functions at 20ms(Background Task).

2. Perform high frequency performance-checking functions at 5ms(Foreground Task).

Below diagram depicts the requirements clearly.

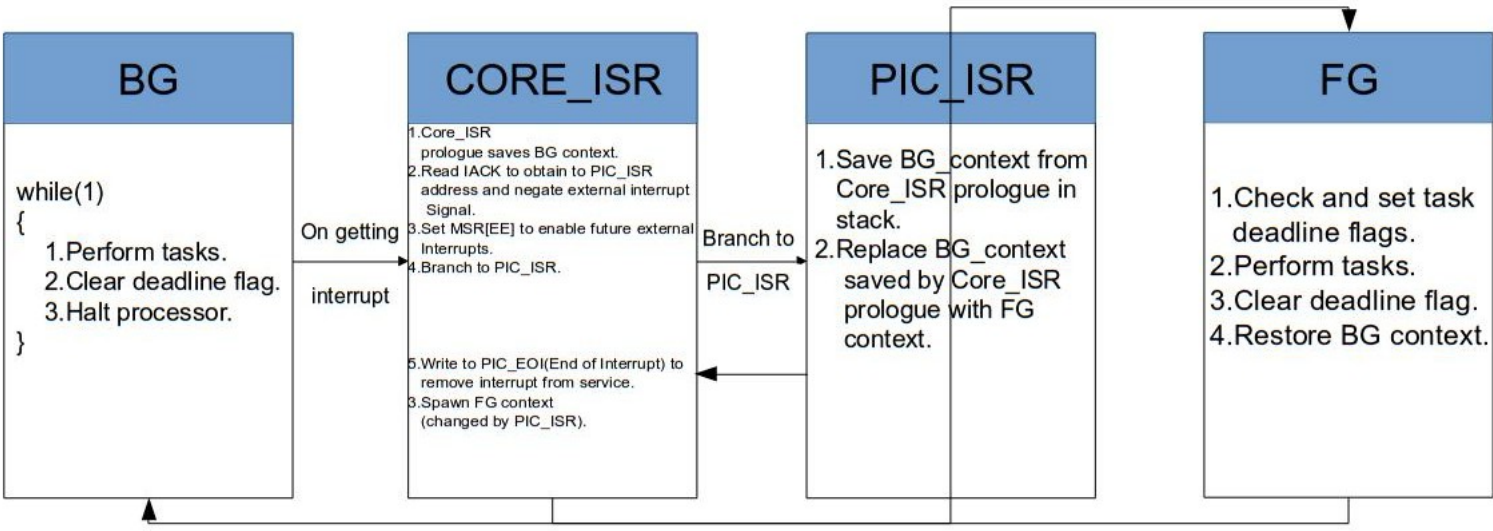


Timing Diagram



End-level architechture

3.2 IMPLEMENTATION:



Task Flow Diagram

As explained in the above diagram, the program structure consists of two tasks, Foreground Task(FG) and Background Task(BG). These are the important points:

1. Priority based preemptive scheduling has been followed to schedule the above mentioned tasks.
2. Foreground task has a higher priority than background, which implies that background task will be pre-empted be foreground task at the moment when foreground task is ready. This implies that the context(describe what context inlcudes in future) of the Background task will be saved, and the Foreground task will be spawned.
3. Since FG has higher priority than BG, FG will not be pre-empted by BG at any point of execution. Although, FG may stopped if it misses its task deadline.
4. FG changes its state to ready every 5ms. It changes its state to waiting after finishing its job.
5. BG task state is ready at all execution time.
6. At 5ms interrupt, CORE_ISR present at a designated memory location is invoked by processor.

7. The CORE_ISR acknowledges the interrupt and calls PIC_ISR.
8. PIC_ISR changes the current execution context to FG context and transfers control to CORE_ISR.
9. CORE_ISR writes to PIC_EOI(End of Interrupt) Register. Exiting ISR returns control to FG.
10. After FG finishes its activities, BG context is restored.
11. Design is catered for minimum context switching.
12. Task Starvation of low priority task has been taken care by assigning deadline to high priority task.
13. Context switch is performed in two conditions:
    - Involutary Switch (through ISR)
    - Voluntary Switch (FG restores BG context)

This is analogous to thread switching, since the above mentioned tasks(FG and BG) coexist in the same virtual address space.

3.3 TASK DESCRIPRTION:

3.3.1 TASK CONTEXT:

The task context comprises:

1. **Non-volatile General Purpose Registers(GPR)**, i.e. r0, r3-r12.

   They are 32 bit registers, which need to be saved while exiting a subroutine and restored while resuming the subroutine.

2. **SRR0 (Save/Restore Register 0)**

   This is a 32-bit register stores the address of the instruction to return to after a non-critical interrupt is serviced. After *rfi* (return from interrupt) instruction is executed, the **PC(PROGRAM COUNTER)** is loaded with this value by processor core.

3. **SRR1 (Save/Restore Register 1)**

   This is a 32-bit register. When a non-critical interrupt is taken, the contents of the **MSR(MACHINE STATE REGISTER)** are placed into **SRR1**. After *rfi* (return from interrupt) instruction is executed, the **MSR** is loaded with this value by processor core.

4. **SP (Stack Pointer)**

   This is a 32-bit register. The stack pointer where task(BG) context is saved.

5. **SPEFSCR (Signal Processing and Embedded Floating-Point Status and Control Register)**

   This is a 32-bit register.
   The SPEFSCR is used by the SPE(Signal Processing Enable) and embedded floating-point APUs (Auxiliary Processing Unit). Bits of this register are affected by:

   (a) Vector floating point instructions.

   (b) Single and double precision floating point instructions.

6. **XER (Integer Exception Register)**

   This is a 64-bit register, which stores carry and borrow bits for addition and subtraction operations and other integer exceptions.

7. **CTR (Count Register)**

   This is a 64-bit register.

   (a) Bits 32:63 of the Count Register can be used to hold a loop count that can be decremented during loop execution.

   (b) The entire 64-bit Count Register can also be used to provide the branch target address for the *Branch Conditional to Count Register* instruction.

8. **LR (Link Register)**

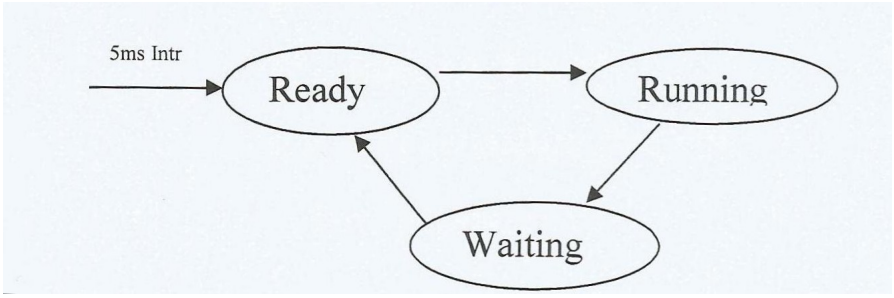
   This is a 64-bit register, which can be used to provide the branch target address for the *Branch Conditional to Link Register* instruction.

9. **CR (Condition Register)**

   This is a 32-bit register, which reflects the result of comparison operations.

3.3.2 FOREGROUND TASK:

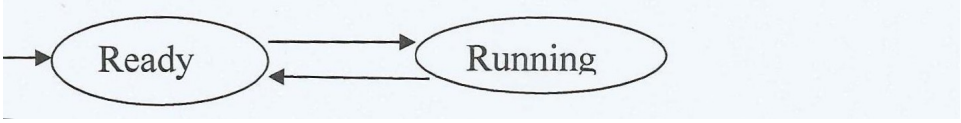
This is a periodic task with *period 5ms* and *deadline 5ms.* The FG task logs previous deadline misses of FG and BG and then performs various activities sequentially and cannot be pre-empted by background task, since it has higher priority.



Task State diagram of Foreground Task

3.3.3 BACKGROUND TASK:

This is a aperiodic task which performs a defined set of activities sequentially and then halts the processor(can be brought to active state only when interrupted). This is an infinite task. The same set of tasks in an infinite while loop are performed again after roughly 20ms (after the FG task is executed 4 times).



Task State diagram of Background Task

3.3.4 CORE INTERRUPT SERVICE ROUTINE(ISR):

The CORE_ISR:

1. Saves BG context in stack.
2. Acknowledges Interrupt.
3. Calls PIC_ISR.

4. Writes PIC_EOI(End of Interrupt).
5. Spawns modified context(by PIC_ISR; context modified to FG context) and thereby lauches FG.

3.3.5 PIC INTERRUPT SERVICE ROUTINE(ISR):

The PIC_ISR:

1. Saves the BG context saved by ISR prologue elsewhere in stack.
2. Modifies the stack where BG context was originally saved to FG context.

3.4 PSEUDO-CODE:

```
//since FG and BG are pending initially setting both flags to 1
FG_deadline_flag = 1
BG_deadline_flag = 1
//For counting FG/ISR execution
int counter = 0;
//Foreground Task
FG_Task()
{
```

- Save initial stack pointer of fg task
- Move to a new stack pointer so as to not disrupt stack where BG context would be saved before executing FG

```
    //setting deadline flags appropriately
    ++counter;

    //Logging FG deadline miss


    //If it is not the first time ISR is executed and deadline flag is set

    if((FG_deadline_flag == 1)&&(counter>1))
printf("FG deadline missed.\n");


    //If ISR has run 4 times, i.e. BG deadline of 20ms has expired

    if(counter==5)
    {
if(BG_deadline_flag == 1)
    printf("BG deadline missed.\n");
counter=1;
BG_deadline_flag=1;
    }
```

- Perform FG activities

```
    //FG completed within deadline
    FG_deadline_flag = 0
```

- Move to ISR stack pointer where BG context is saved
- Restore BG context

```
}
//Background Task
BG_Task()
{
    while(1)
    {
      • Perform BG activities
        //BG completed within deadline
        BG_deadline_flag = 0
        while(BG_deadline_flag == 0)
          • Halt Processor
    }
}

//CORE_ISR(Interrupt Service Routine)
core_isr()
{
  • Acknowledge interrupt
  • Set MSR[EE] to re-enable external interrupts
  • Call PIC_ISR
  • Write PIC_EOI
}


//PIC_ISR(Interrupt Service Routine)
pic_isr()
{
  • Save the BG context saved by ISR prologue elsewhere in stack.
  • Modify the stack where BG context was originally saved to FG context.
}

//Main Function
int main()
{
  • Startup code for halting processor, i.e.  enabling HID0[DOZE] bit
  • Linking ISR to appropriate interrupt through PIC(Programmable Interrupt
    Controller)
  • Genering PIC_IPI(Interprocessor) interrupt.
}
```

Currently, BG deadline is logged after 4 executions of ISR, and thereby FG. To maintain an exact 20ms deadline, TCR(Timer Control Register) interrupt can be used.

1. TCR_{FP}[38:39] concatenated with TCR_{FPEXT}[47-50], selects one of the 64-bit locations of TB(Time Base) to use as Fixed Interval Interrupt Period.

2. Set TCR_{FIE} and MSR_{EE} to enable this interrupt.
   Read section 2.6 in E500CORERM datasheet for further details.

3. BG_deadline_flag should be set to 1 in the handler for the Fixed-interval timer interrupt, i.e. IVOR11. Since IVOR11 is by default written to 0xb00 by the start-up code, the interrupt vector for this handler should be placed at 0xb00.

3.5 SOME IMPORTANT DETAILS:

3.5.1 PROGRAMMABLE INTERRUPT CONTROLLER(PIC):

The CPU core, in this case e500v2 usually has only one pin for external interrupt. Thus, multiple external interrupts must be routed through Programmable Interrupt Controller(PIC).
Handling external interrupts through PIC:

1. The CCSRBAR register contains the start address of the CCSR(Configuration, Control and Status Registers) space.

2. The CCSR space contains various configuration registers including PIC registers and is located in the local address space.

3. The CCSRBAR has been configured to 0xffe0_0000 currently. PIC is at an offset of 0x4_0000 from the start of CCSR space, as mentioned in P1024RM datasheet. Thus, PIC registers are located in local memory starting from address, 0xffe4_0000.

4. For testing purpose of this project, interprocessor interrupts have been generated and handled.

5. The interrupt vector address and priority is updated in PIC_IPIVPR0(0xffe4_10a0).

6. The IPI interrupt is generated through Core 0 IPI Dispatch register - PIC_IPIDR_CPU00(0xffe6_00400) / PIC_IPIDR0(0xffe4_00400).

7. Following sequence has to be followed to initialise PIC registers for interrupt handling:

   (a) Write the vector, priority, and polarity values in each interrupt's vector/priority register, leaving their MSK (mask) bit set. PIC_IPIVPR0(0xffe4_10a0) to 0x800f0c00, where f is the priority and 0x0c00 is the interrupt vector address. The mask bit is also set.

   (b) Clear CTPR - Current Task Priority Register - 0xffe4_0080 / CTPR_CPU0 - 0xffe6_0080 (CTPR = all zeros).

   (c) Program the PIC to mixed mode by setting GCR[M], so that all external interrupts, IRQ0-11, can be processed.

   (d) Clear the MSK bit in the vector/priority registers to be used.

   (e) Perform a software loop to clear all pending interrupts. This has been ignored at the moment since we have manually ensured that there are no pending interrupts. If required, this can be done as follows:

      i. Load counter with FRR[NIRQ].

      ii. While counter > 0, read IACK and write EOI to guarantee all the IPR and ISR bits are cleared.

   (f) Set the processor core CTPR values to the desired values.

8. Set Message enable register (PIC_MER) to 0x0000_000F.

The interrupt routing has been checked as follows:

1. The activity bit in the interrupt's vector/priority register is set.

2. The IACK register now contains the interrupt vector address. (0x0c00)

In the actual applications, an FPGA based 5ms timer will be used to provide external interrupt. In that case, PIC_EIVPR0(0xffe5_0000) can be used as the interrupt vector/priority register in place of PIC_IPIVPR0.
The external interrupt dispatch register, PIC_EIDR0 should be automatically set by PIC on arrival of FPGA external interrupt.

The PIC now passes on the interrupt vector address to the core register IVOR4, for external interrupt. Read E500CORERM datasheet for further information.

PIC initialisation is to be done in start-up code.

3.5.2 CORE HALTING:

DOZE power-saving mode was used for CORE-HALTING.
To enable DOZE mode:

1. Enable core register, HID0[DOZE] bit, in start-up code.
2. When we need to halt core:
   (a) Clear TCR register, to disable timer interrupts, which can bring core out of halt state. Decrementer interrupt by default gets enabled in a while loop, which has to be disabled.
   (b) Enable MSR[WE], preceded by *msync* and succeded by *isync* instruction.
      i. *msync* is to ensure core does not contain cached copies of instructions from the old address space.
      ii. *isync* is to ensure that the instruction to enable MSR[WE] is definitely executed.

DOZE MODE:
Core complex clocks continue running, and bus snooping continues to maintain L1 cache coherency. The core complex is in core-halted state when the integrated device is in doze state.

3.5.3 TASK SWITCHING:

1.TASK SWITCHING FROM ISR TO FG:

1. When the external interrupt switches control from BG to ISR, current BG program counter and MSR are saved by processor core in SRR0 and SRR1 respectively.
2. The ISR prologue generated by the compiler saves the current BG context, which includes SRR0 and SRR1 in stack.
3. In ISR, we save the stack locations containing SRR0 and SRR1 in a different location in stack.
4. Then, we modify the original locations of SRR0 and SRR1 to now contain the starting instruction address of FG and the MSR value with SPE bit enabled(allows use of vector operations in FG) respectively.
5. The ISR epilogue restores these new values to SRR0 and SRR1.
6. When *rfi*(return from interrupt) is called at the end of ISR epilogue, the values of SRR0 and SRR1 are restored to PC and MSR by processor core.

2.TASK SWITCHING FROM FG TO BG:

1. The BG context saved in stack by ISR prologue, is now restored by FG.
2. SRR0 and SRR1 are restored from the locations where BG SRR0 and SRR1were separately saved by us.
3. *rfi* is explicitly called eventhough FG is not an ISR so that the values of SRR0 and SRR1 are restored to PC and MSR by processor core.

3.5.4 VECTOR OPERATIONS:

SPE APU instructions treat 64-bit GPRs as being composed of a vector of two 32-bit elements. This allows for some vector instructions for loading and storing, adding, subtracting and multiplying GPRs. Thus MSR[SPE] bit has been enabled before using vector instructions. The vector instructions used in this program are:

1. evldd - Vector load double word into double word.
2. evstdd - Vector store double word into double word.
   This has been used in ISR prologue generated by compiler, hence not visible in my code. This has been generated the __interrupt keyword prefixing my ISR.

4 DATASHETS USED:

1. BOOK_E
2. E500CORERM
3. P1024RM

5 CONCLUSION:

1. The RTE(Real Time Executive) developed by me has been optimised for a particular project. But it can be used as general RTOS in all future avionics systems.
2. The above report has been read and verified by Mrs. Alka Soni, Scientist 'D', DARE(Defence Avionics Research Establishment), DRDO, Bangalore.