# CSE522 Real-time Embedded Systems

# Assignment 3 Report

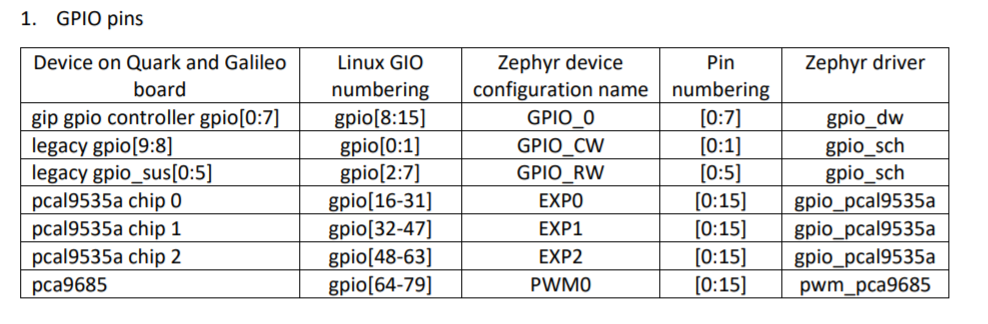
Group No. 2

Hena Shah- 1213348511

Nidhi Dubey-1213031246

Our program performs 3 measurements in sequence: interrupt latency without background computing, interrupt latency with background computing, and context switching overhead RTOS (Zephyr version 1.10.0) : interrupt latency (with and without back ground computation) and context switching overhead. Interrupt latency is the total delay between the interrupt signal being asserted and the start of the interrupt service routine execution. This delay may be extended if an interrupt arrives when the RTOS is in a non-preemptive critical region. As for context switching overhead, it is the delay of context switching process of saving the context of the executing thread, restoring the context of the new thread, and starting the execution of the new thread. We measure 500 samples, for each measurement which is stored in separate buffers. We have also registered a shell module to print data randomly from the buffer after the program ends.

We have used the table below, to configure and write data pins,

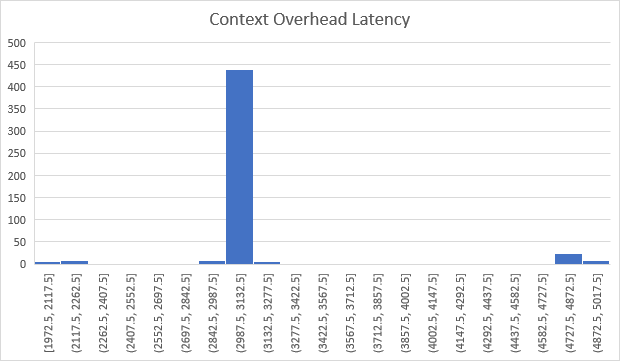


We have also referred to the pinmux.c file to properly configure the multiplexing and pull up pins. We have referred ZEPHYR documentation and other Zephyr samples for function definitions and their implementations.

CASE 1:

For the context switch, we have created two threads, one of priority higher than the other. First the higher priority thread tries to acquire the mutex but cannot acquire as the lower priority thread has acquired the lock and we find the context switch overhead by taking the time stamp after the higher priority thread acquires the locks and before the lower priority thread releases the lock. Here, we are also subtracting the overhead for k\_mutex\_unlock which we have calculated intially in the main process. The overhead for mutex\_unlock varies between 850-1350 so the context switch latency varies a bit(when we run the program multiple times) but the distribution in the histogram graph comes to be same as the context switch varies accordingly. Here the overhead for k\_thread\_mutex is not taken into consideration as the higher priority thread is in the waiting state after it tries to acquire the lock and it simply goes to the ready state when the lower priority thread releases the lock. We are using x86’s Time Stamp Counter (TSC) to read instances.

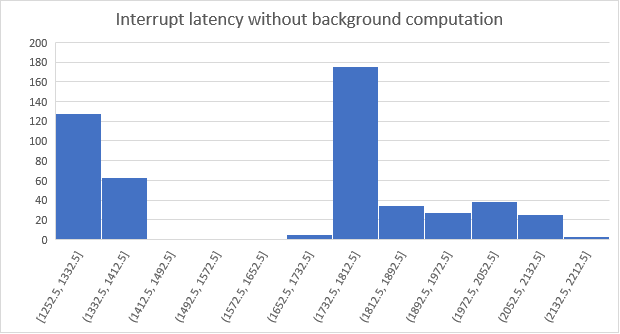
In this graph 90% of our samples are concentrated in one range since the unlocking mutex time and the computation time in lower priority thread will almost remain constant.



CASE 2:

Over here interrupt latency without background computation is found. In this case we constantly measure time stamp as we don’t know precisely the arrival of interrupt on IO0. From the graph we can see there is a bit variation in the interrupt latency. We have used PWM to generate interrupts which will further call a callback function which is the start of the interrupt service routine. We are measuring time between when the interrupt arrives and when it enters the call back function.

In the graph below, we can see that approximately 80% of our values are centered across 2 ranges. The time computed on this case has to be less than when we measure it with computation.



CASE 3:

In the program PWM is implemented to trigger the interrupt periodically. The time stamp is taken after each instruction in the background computing threads because the PWM is in background and we don’t know the exact arrival of the rising edge on the input pin IO0. We first used the approach to find the number of ticks during each cycle and then use it to estimate the arrival of the interrupt on pin IO0. But the estimation could not be done properly, and it effected the interrupt latency with background computing. So, we used the approach to get the time stamp after every instruction and when the interrupt is called the threads are preempted and we get the time stamp for instruction at which has been preempted. Here from the graph we see that the overhead is greater than the one without the background computing as the message queue while putting and getting data in the buffer disables the interrupt.

As we can see from the above graph, the interrupt latency has a larger range from 1240 to 3600 nanoseconds approximately depending on when the interrupt arrives. The peak value is greater than the one we measured in case of without background computation since the delay increases in the presence of a non-pre-emptible critical region. Around 80% of the samples have values greater than the maximum time range we computed in case 2.

