Embedded Systems

Ιακωβίδης Ιωάννης

10 Μαΐου 2019

1 Introduction

We know that embedded systems must satisfy a variety of constraints due to the type of applications. One of the most important type of constraints are timing constraints. Very frequently we need for a system to be real time. In this report we will explain a implementation of a periodic sampling task and the results of **execution on the Raspberry pi Zero W**.

There are many instances that we want to use a periodic task in an embedded system to take input for the environment in a timely manner. Ordinary we want sample the input from some peripheral and store it in a volatile variables ,that update every time from memory. In the purposes of this report we sample the gettimeofday function and we analyze the results to quantify the timing performance of the system.

In this document we explain two implementation and their timing results. First an implementation that does not use the time samples to correct handle clock drift and the second does.

For the rest of the document let T be the total time of the sampling task N the number of samples and dt the sampling period.

2 Implementation

The idea of the implementation is that we wake up sample and sleep until the next period. So to make the periodic task we need to use a timer we use the function setitimer.

```
int setitimer(int clk, const struct itimerval *new_value,
              struct itimerval *old_value);
```

For the clk parameter we use ITIMER_REAL. This way we decrement the timer in real time, and deliver SIGALRM upon expiration. Then using the function sigwait we wait until the next period. The following listing shows the code structure.

```
/*Set timer*/
int status =set_timer(period,&sigset);
/*For the number of samples*/
 while(i<N){
        /*Wait for signal*/
        sigwait(&sigset,&dummy );
        /* Do work*/
        /*Sample */
        gettimeofday(&now, NULL);
        /*Compute and store results*/
i++;
 }
```

3 Clock correction

For our second implementation we will use the samples to better synchronize the task. First correction in order to handle clock drift we maintain a moving average (a cyclic buffer) of the last 64 samples and if the moving average is absolutely greater than 0.1 ms we restart the timer. The setitimer has two values an offset and an interval (period) in order to restart the timer right we need to handle the offset appropriately to not add drift ourselves. So we take one more sample (s) and we we set the offset to $(i+1) * dt - s$ for the i'th period.

But through our test runs we have found out that we can have single time signals that are inaccurate and that doesn't mean that they necessarily add gitter (see diagrams). Also resetting the timer can add inaccuracy to our periodic task. So we the condition for restarting the timer that we have chosen is:

$$((|Mean| > 0.1 \&\& number > 10) || |dif| > 10)$$

Where Mean is the moving average that we mention above,number is the number of samples that $i * dt - t > 0.1$ this number is set to 0 every 500 periods and dif is the difference of the current sample time from the desired periodic. The following listing shows the structure of the code:

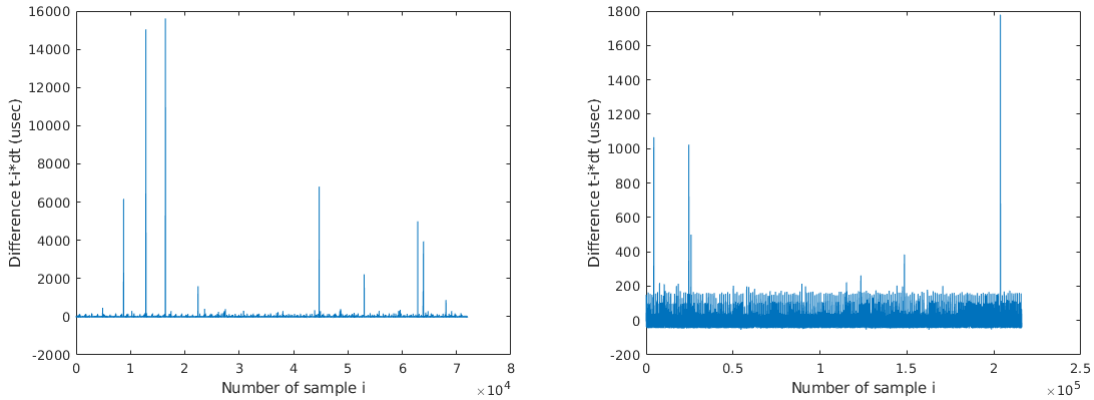
```
/*Set timer*/
int status =set_timer(period,&sigset);
/*For the number of samples*/
 while(i<N){
        /*Wait for signal*/
        sigwait(&sigset,&dummy );
        /* Do work*/
        /*Sample */
        gettimeofday(&now, NULL);

        /*Compute moving average mean*/
         mean=(mean*64-cyclic[i%64]+dif)/64;
           if(fabs(dif)>0.1){
      number++;
    }
    if(i%500==0){
      number=0;
    }
        /*Compute and store results*/
if((fabs(mean)>0.1 && number>10 ) || fabs(dif) >10){
         /*Get one more sample for the offset*/
         /*Restart the timer*/
         setitimer(ITIMER_REAL, &t, NULL);
         mean=0;
         }
i++;
 }
```

So recapping we first maintain a 64 element array called cyclic that has the last 64 samples and with that we compute the moving average,mean variable . If the moving average passes 0.1 ms we restart the timer with offset (it_value) the remaining time on that period and it_interval the period.

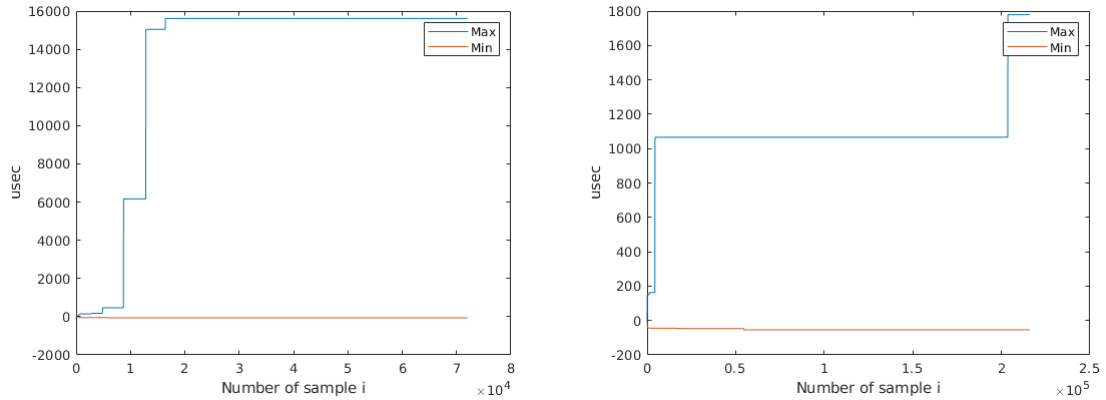
The numbers 0.1ms and 64 were chosen after executions to represent a reasonable value that shows that the clock has drifted and the number of elements to have a local moving average. In our computation of the offset **we also subtract a percent of the last difference to get rid of probable offset**.

4 Results

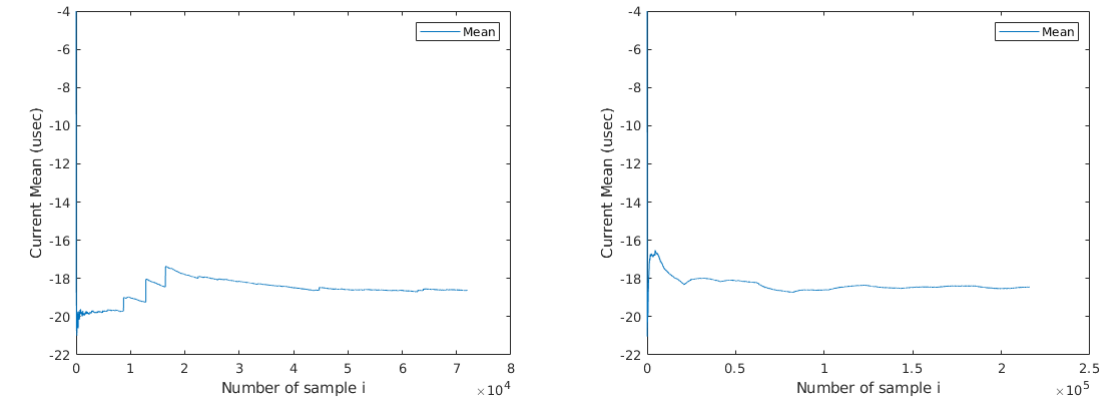
We tested the implementation on many different sizes and we extracted diagrams that show the difference of the sample from the desirable time e.i. $t - i * dt$. Also we will show that we don't lose any sample and that the samples are uniformly spaced.



In the above graphs we see the differences of the signal from the desirable time for 2 and 6 hours in usec . In the left we have the first implementation for 2 hours and we see that we have mainly differences that are between 40 and -40 usecs while we have single samples that are from 6 to 16 msecs. In the right we have the second implementation for 6 hours and we see that we have mainly differences that are between 100 and -100 usecs while we have single samples that are from 0,4 to 2 msecs.

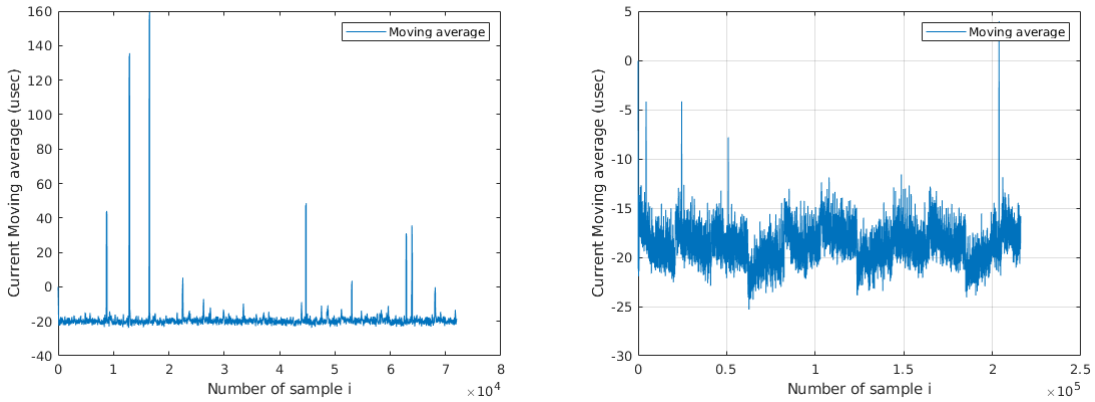


Next in the above we see the graphs from the maximum difference from the desired time. At the left we see the first implementation has a maximum of 16 ms and a relatively insignificant minimum this means that not only the samples are the same number but that in the magnitude of one hundernth of the period all of the time. Same for the right image which is the six hour implementation with timer resetting.

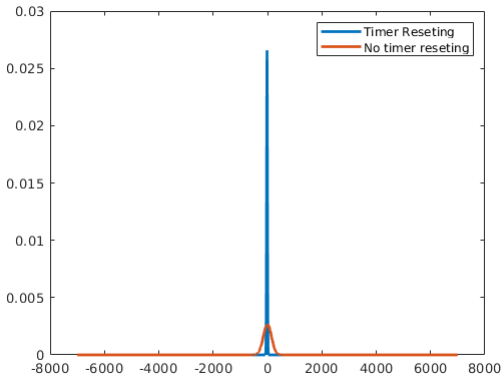


Here we see the graphs of the mean of the difference of the samples form the periodic.We see

that they are roughly the same and that we see no clock drift even for the six hour execution.



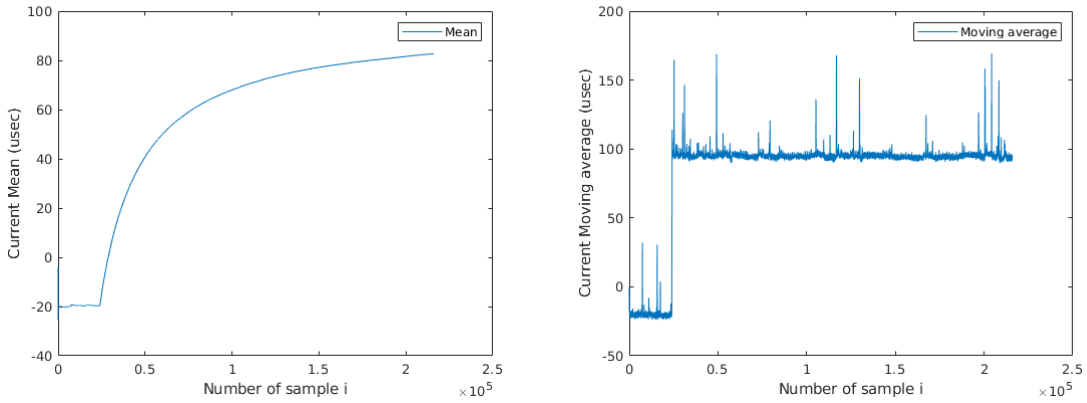
Above we see the moving average diagrams for 100 elements and also we observe that it does not begin to show appearance of clock dift.



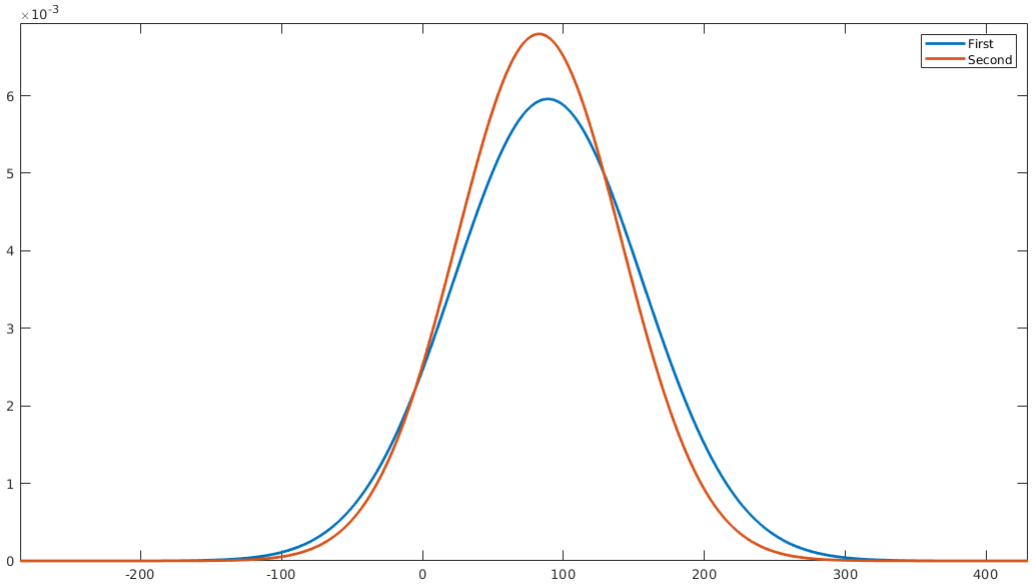
If we fit a normal distribution for the two test runs we see by the above image that they look statistically the similar. The changes that we see are due to the different total times that the two executions took place and the context that the two executions took place (how busy was the Raspberry-pi).

We emphasize that we had 3 resets in the timer resetting run appeared above without adding any additional offset.

Now for a sidenote if **we had not handled the offset** in the timer resetting,we present a 6 hour run, implementation we see that the mean is increasing with time.



This is because we lost accuracy after the first reset. But that mean would stabilize to value around 100usecs, this is apparent by the next diagrams.On the right we see the moving average diagram and we see that we almost have a constant offset of 100usecs.



1: Distribution

But if we fit a normal distribution,above image, for the two test runs (2 hour no timer resetting and 6 hour timer resetting no offset handling) we observe that they look statistically the same.

5 Conclusion

If we had more active tasks in our systems me would certainly had clock drift in our task because of the number of context switches that would take place. So it is necessary to use the samples to reset the timer but also we need to choose the right conditions to reset the timer because we may add some offset to the original timing.

6 CPU Usage

Running the task in the background and using the Linux top command we find out that the CPU usage of the task is 0 according to their metric. And to raise the power usage above 50% we need to use period of 0.1 ms. Here is a table of the Cpu usage related to the period.

CPU usage related to the period on Raspberry pi Zero W	
Period	CPU usage
0.1 sec	0 %
0.01 sec	0 %
0.001 sec	8.6 %
0.0001 sec	50.3 %

7 Code

CodeLink