ARISTOTLE UNIVERSITY OF THESSALONIKI

PRODUCER-CONSUMER TIMER PROBLEM

Real Time Operating Systems Course

TZOUVARAS EVANGELOS
AEM: 9659
Email: tzouevan@ece.auth.gr

Table of Contents

Introduction

This project developed on Real Time Embedded Systems course at Aristotle University of Thessaloniki. The concept is to create a timer in C/C++ by using the producer-consumer problem. At each “tick” of the timer the timer function must be run, which calculates the sin of ten angles. So at each period time, the producer place a workfunction type object into the queue, and the consumer runs the work function.

Each source code run at the raspberry pi zero, and text files are created that save the period, execution time and the drift error by using timestamps inside the code. The data are visualized into graphs by using the Matlab. You can find the whole project on github.

Source file implementation

Two different source files were created. The first one has only one timer and the period set up by the user (10ms, 100ms, 1sec). The second one has all the timers that run at the same time.

As for the prod_cons_timer.c file (with one timer), a struct Timer is declared which contains all the variables (period, task to execute, Start delay, queue, producer p_thread, workfunction type) and pointers on the function variables.

```c
/* Declare a struct Timer with all the timer parameters and functions (Timer specifications)*/
typedef struct{
  int period;                               // The time between two executions of the TimerFcn (in milliseconds)
  int TasksToExecute;                       // The number of task executions
  int  StartDelay;                          // The delay time (in seconds), before the start of the timer

  // Declare a queue inside the struct
  queue *fifo;

  // Declare the producers
  pthread_t pro;

  // A workfunction struct that contains the userData and the TimerFcn
  workFunction Timer_Work;

  // Declare the struct functions (pointers in functions)
  void *(*StartFcn)(void *);
  void *(*StopFcn)(void *);
  void *(*ErrorFcn)(void *);
}Timer;
```

Figure 1: Timer struct

A Timer_Init function was implemented to create the timer and to initialize all the timer variables. This function has as an argument a queue type to pass the queue, which was created into the main function, into the timer. The timer function return a timer object to be used into the main function.

```c
// Initialize a timer
Timer *Timer_Init (queue *q){
  Timer *t;

  // Allocate memory for the timer
  t = (Timer *)malloc (sizeof (Timer));
  if (!t){
    printf("Failed to allocate memory for the timer!");
    return (NULL);
  }

  // Initialize the timer variables
  t->period = PERIOD;
  t->TasksToExecute = (TIME_TO_EXECUTE*1000)/PERIOD;
  t->StartDelay = START_DELAY;

  // Copy the external queue into the timer queue
  t->fifo = q;

  // Initialize the functions
  t->StartFcn = &start_Fcn;
  t->StopFcn = &stop_Fcn;
  t->ErrorFcn = &error_Fcn;

  // Initialize the array with the angles
  int *p = (int *)malloc(sizeof(int)*10);
  int k = 0;
  for(k = 0; k < 10; k++)
    p[k] = k;
  t->Timer_Work.arg = p;
  t->Timer_Work.work = &work;
  return t;
}
```

Figure 2: Timer Init function

The struct Timer has three different functions, the StartFcn, the StopFcn and the ErrorFcn, which print the appropriate messages at the start and at the end of the timer or if an error appeared. Also, the start function is responsible to start the timer by creating the producer p-thread and the startat function is responsible to start the timer at specific date by inserting an extra delay time at the beginning of the producer thread.

```
411  // Function to start the timer
412  void start(Timer *t){
413    (t-> StartFcn)(NULL);
414    pthread_create (&(t->pro), NULL, producer, t);
415  }
416
417  // Function to start the timer at specific time
418  void startat(Timer *t, int y, int m, int d, int h, int min, int sec){
419    // Start the timer by calling the StartFcn
420    (t-> StartFcn)(NULL);
421
422    // Convert the specific date into a struct timeval type
423    struct tm specificDateTime = {0};
424    specificDateTime.tm_year = y - 1900;
425    specificDateTime.tm_mon = m - 1;
426    specificDateTime.tm_mday = d;
427    specificDateTime.tm_hour = h;
428    specificDateTime.tm_min = min;
429    specificDateTime.tm_sec = sec;
430
431    // Convert the specificDateTime to seconds by using the mktime function
432    time_t DelayTime = mktime(&specificDateTime);
433
434    // Take the current time by using the gettimeofday() function
435    gettimeofday(&currentTime, NULL);
436
437    // Calculate the time difference
438    int Delay = DelayTime - (currentTime.tv_sec);// + 0.000001*currentTime.tv_usec);
439    if (Delay > 0)
440      t->StartDelay = Delay;
441    else
442      t->StartDelay = 0;
443
444    // Add an extra delay before the producer creation
445    usleep((t->StartDelay)*1000000);
446
447    // Create the producer
448    pthread_create (&(t->pro), NULL, producer, t);
449  }
450
```

Figure 3: Start and startat functions

For each timer one producer is created which is responsible to insert inside the queue workfunction data types (a struct which contains the “timer” function). The function now has as an argument the timer t, to have access on the timer queue, on the producer p-thread and on other timer variables (periods, tasksToExecute etc.).The producer has a “for” loop which runs for TasksToExecute times. Each loop has a delay equals with the period time of the timer (the delay is implemented by using the usleep function). The tasksToExecute are calculated by the following formula:

$$TasksToExecute = \frac{TimeToExecute(ms)}{Period(ms)}$$

```
212
213  void *producer (void *t)
214  {
215    // Create a timer and copy the *t pointer after typecasting
216    Timer *timer;
217    timer = (Timer *)t;
218
219    int i;
220    for (i = 0; i < (timer->TasksToExecute); i++) {                // Loop to run the code multiple times
221      pthread_mutex_lock ((timer->fifo)->mut);                     // Mutex lock
222      while ((timer->fifo)->full) {                                // While fifo is full
223        printf ("producer: queue FULL.\n");                        // Print fifo is full
224        pthread_cond_wait ((timer->fifo)->notFull, (timer->fifo)->mut); // producer thread wait until the fifo is notFull, then unlock the mutex
225      }
226      printf("\nTask to Execute: %d\n", i);
227
228      // Measure each time that the producer place an element to the queue
229      gettimeofday(&prod_times[i], NULL);
230
231      // The producer will place inside the queue a workfunction type structs
232      queueAdd ((timer->fifo), (timer->Timer_Work));
233
234      pthread_mutex_unlock ((timer->fifo)->mut);                   // Unlock the mutex
235      pthread_cond_signal ((timer->fifo)->notEmpty);               // The consumer thread waits for the notEmpty signal
236
237      usleep ((timer->period)*1000);
238    }
239    return (NULL);
240  }
241
```

Figure 4: Producer function

The consumer function does not have any change. It just takes the queue as an argument and runs the function of each workfunction element of the queue.

```
241
242  void *consumer (void *q)
243  {
244    queue *fifo;
245    int i, d;
246
247    /* A workFunction type */
248    workFunction out;
249    fifo = (queue *)q;
250
251    while(1){                                                    // Consumer will always run
252      pthread_mutex_lock (fifo->mut);                            // Lock the mutex
253      while (fifo->empty) {                                      // While fifo is empty
254        printf ("consumer: queue EMPTY.\n");                     // Print the queue is empty
255        pthread_cond_wait (fifo->notEmpty, fifo->mut);           // Mutex will remain lock until the fifo is empty. If FIFO not empty unlock the mutex
256      }
257
258      /* The consumer will delete each fifo element and will run the function */
259      queueDel (fifo, &out);                                     // The out workFunction will take the final workFunction queue elemen
260      out.work(out.arg);                                         // Here the consumer calls the work function
261      free(out.arg);
262
263      /* Get the time that the consumer run the function */
264      gettimeofday(&con_times[executed_tasks], NULL);
265
266      pthread_mutex_unlock (fifo->mut);
267      pthread_cond_signal (fifo->notFull);                       // The producer thread waits for the notFull signal
268      printf ("consumer: recieved %d.\n", d);
269
270      /* Stop the consumer */
271      executed_tasks = executed_tasks + 1;
272      if(executed_tasks == ((TIME_TO_EXECUTE*1000)/PERIOD))
273          break;
274    }
```

Figure 5: Consumer function

Three different files are created to save the results by using timestamps. The first one “Periods.txt”, saves the periods of the timer (the time difference between each “for” loop iteration inside the producer function). The second file (“Execution_times.txt”), saves waiting time of each function inside the queue. The third file (“Drift_error.txt”), saves the delay time between the theoretical period of the timer and the actual period of the timer.

As for the second source file, “prod_cons_all_timers.c”, the difference is that all the timers run at the same time (the main function creates three timers by using the Timer_Init function, with different periods) and three consumers are used to execute the functions.

Results

1. Timer at 10ms

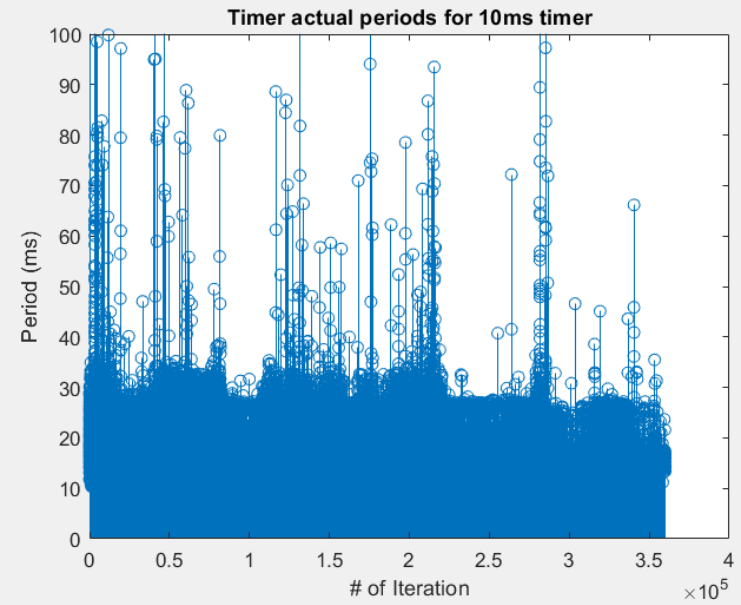


Figure 6: Actual periods on 10ms timer

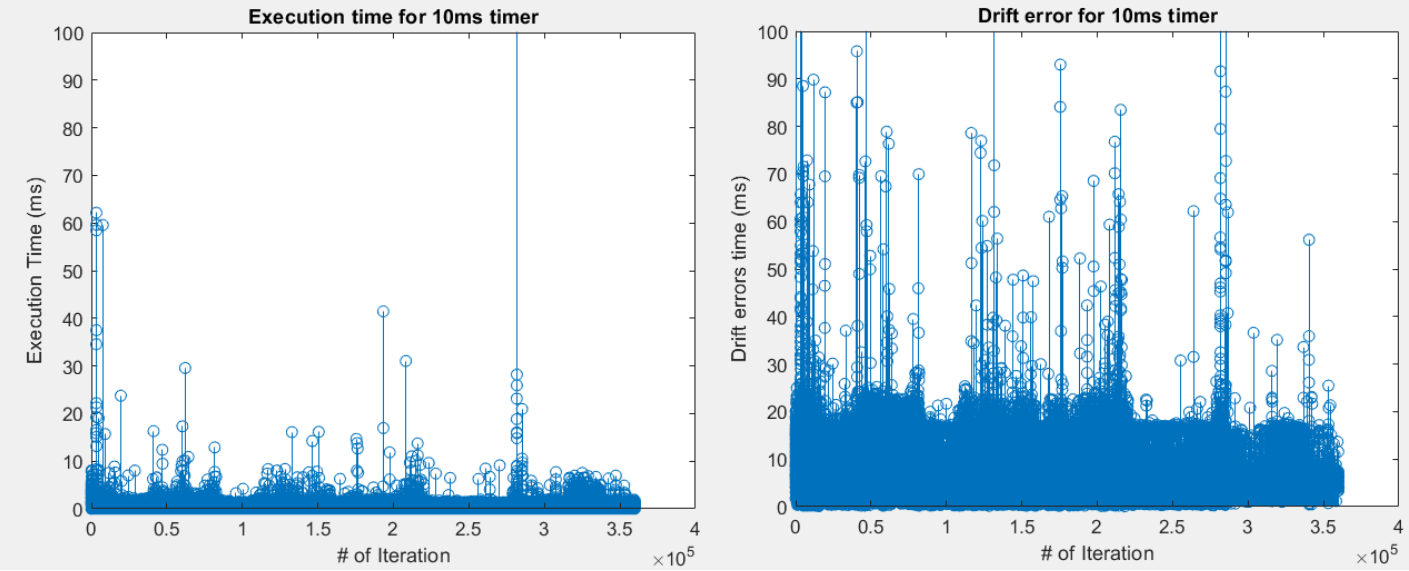


Figure 7: Execution time and drift error for 10ms timer

```
============================================================
Mean Period(ms):1.517888e+01
Mean Execution Time(ms):4.623055e-01
Mean Drift Error(ms):5.178840e+00
============================================================
```

Figure 8: Mean results for 10ms timer

It is observed that on the 10msec timer the drift error is over 50% of the theoretical period. As a result the mean actual period value is 15.17msec, significantly greater than the 10msec which is the objective. The timer has also some extreme values up to 100msec.

2. Timer at 100ms

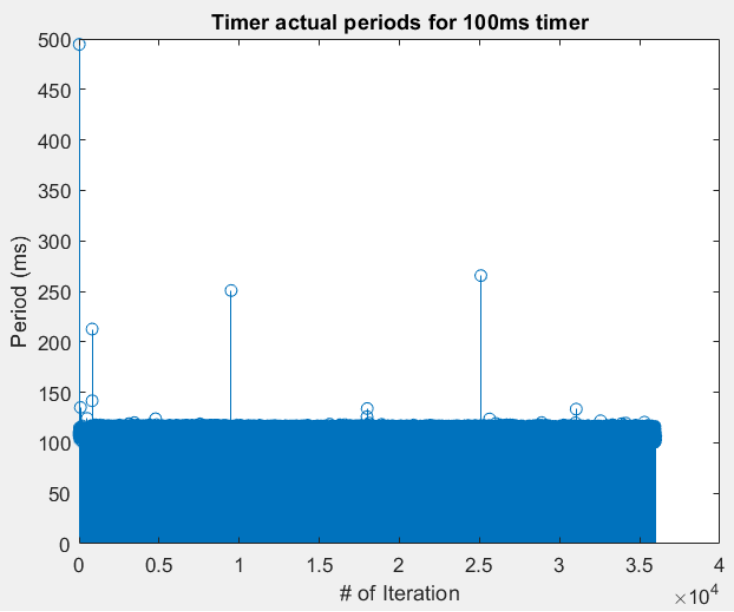


Figure 9: Actual periods on 100ms timer

```
============================================================
Mean Period(ms):1.081053e+02
Mean Execution Time(ms):7.188975e-01
Mean Drift Error(ms):8.102283e+00
============================================================
```

Figure 10: Mean results for 100ms timer

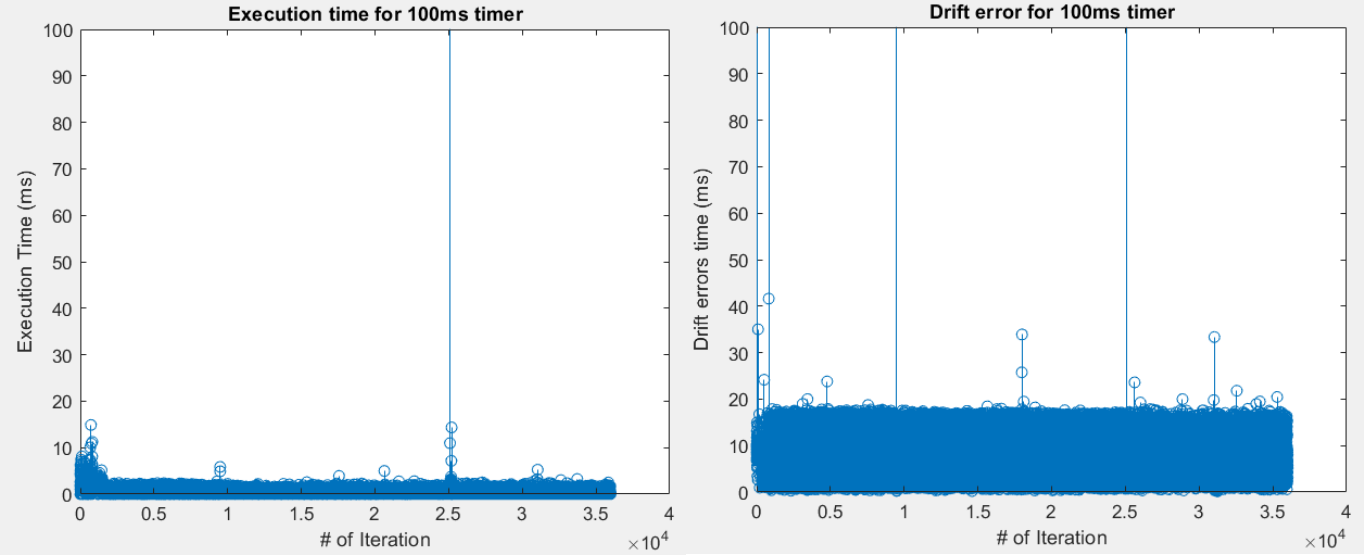


Figure 11: Execution time and drift error for 100ms timer

On the 100msec timer the mean drift error is 8.1msec, smaller than the drift time on 10msec timer. Also, the percentage of the drift error on relation with the timer period is 8.1%, measurable value but not that big.

3. Timer at 1sec

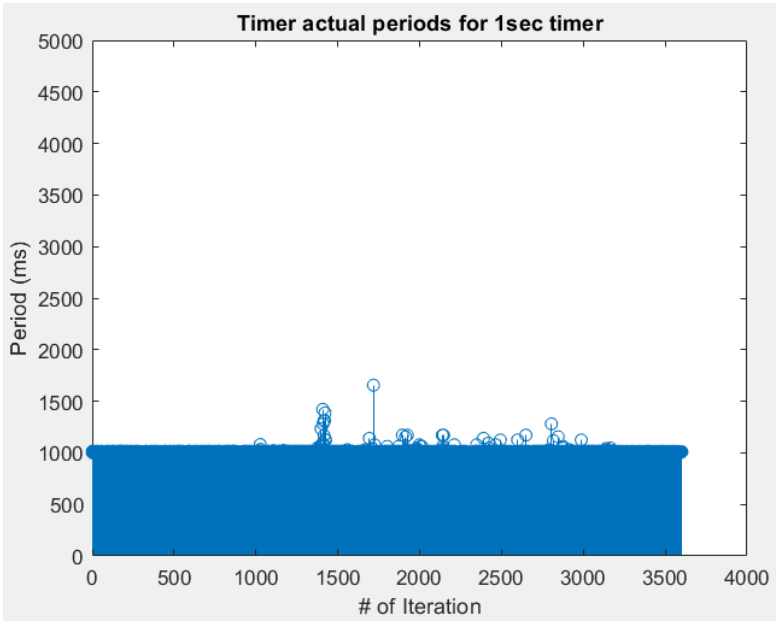


Figure 12: Actual periods on 1sec timer

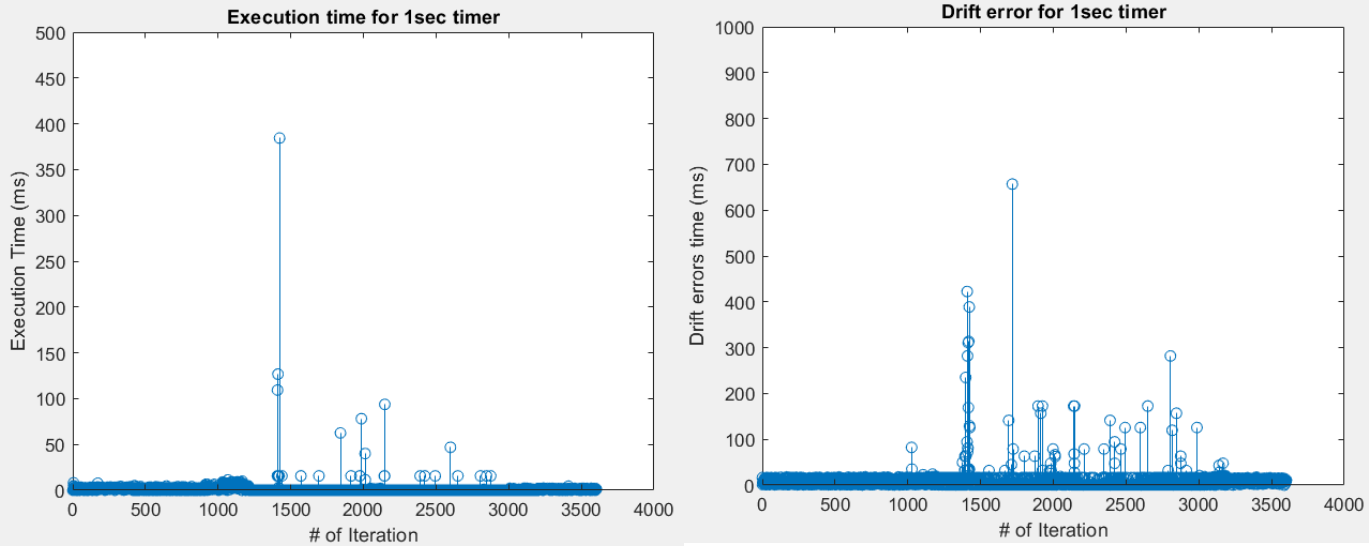


Figure 13: Execution time and drift error for 1sec timer

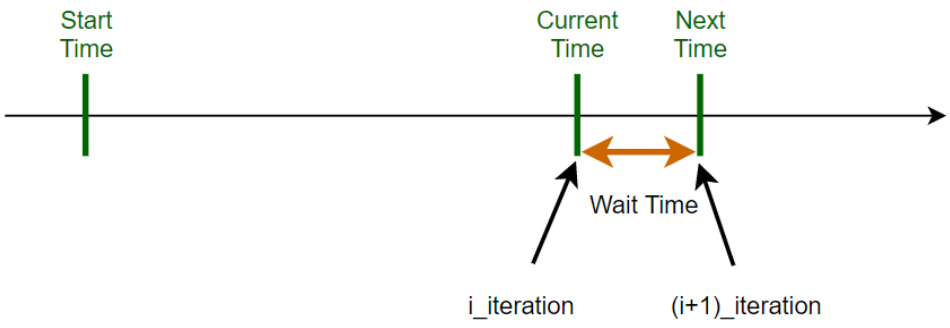
```
==========================================================
Mean Period(ms):1.007767e+03
Mean Execution Time(ms):1.192878e+00
Mean Drift Error(ms):8.487498e+00
==========================================================
```

Figure 14: Mean results for 1sec timer

The mean drift time on the 1sec timer in about 8.5msec, similar with the drift time on the 100msec. But as a percent, it is only the 0.8% of the theoretical period value. The actual period of the timer is close enough with the theoretical period.

Drift error solution

To solve the constant drift error that was appeared on the timers above, the start time must be taken into consideration. Based on the image above the new wait time must be calculated.



$$Wait\ Time = Next\ Time - Current\ Time$$

$$Next\ Time = Start\ Time + (i + 1) * (Timer\ period)$$

So the wait time is calculated on each iteration inside the producer by the following formula and is used as an argument on the usleep function.

$$Wait\ Time = Start\ Time + (i + 1) * (Timer\ period) - Current\ Time$$

With this update on the producer's function, the drift error tents to zero and the period of the timer is constant, as the following graphs are shown below.

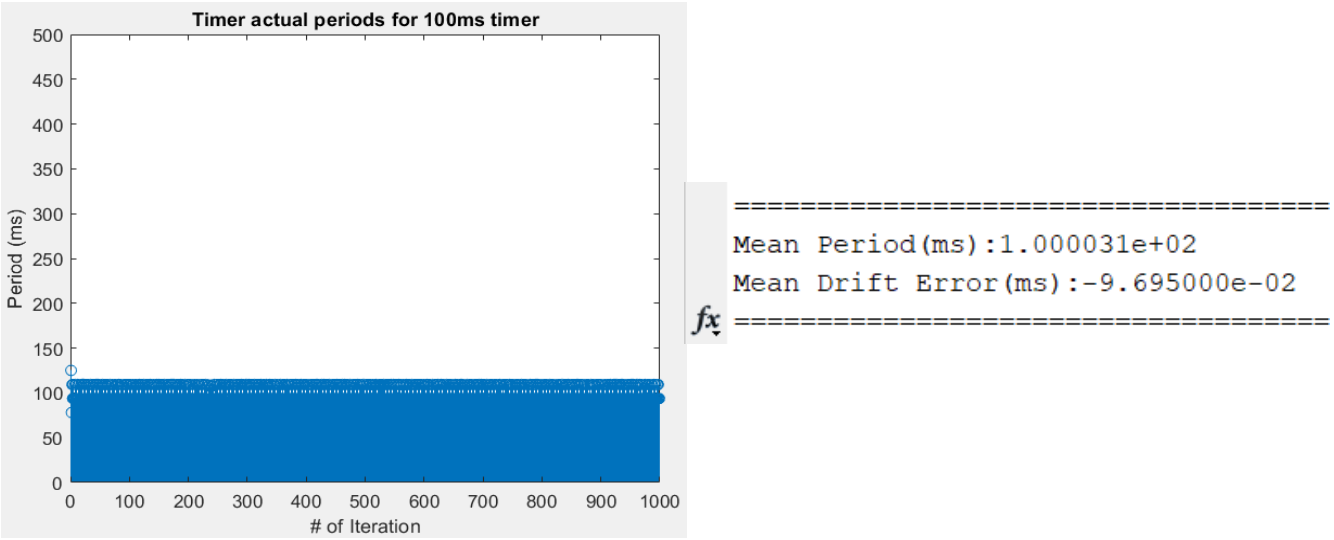


Figure 15: Timer actual periods, for 100ms timer, without drift error

Bibliography

- Real Time Embedded Systems course, e-Learning auth, https://elearning.auth.gr/course/view.php?id=9933

- POSIX Threads Programming, https://hpc-tutorials.llnl.gov/posix/