A) Objectives. In this lab we were tasked with implementing blocking semaphores that worked with a preemptive/cooperative priority scheduler. We also implemented facilities for multiple periodic and aperiodic tasks. Finally, we implemented low-level debugging tools and profiling features to monitor and assess system performance.

B) Hardware Design (none).

C) Software Design.

1) Jitter.

```
long MaxJitter;
unsigned static long time11 = 0;  // time at previous ADC sample
unsigned long time21 = 0;         // time at current ADC sample
long MaxJitter1;             // largest time jitter between interrupts in usec
#define JITTERSIZE 64
unsigned long const JitterSize1=JITTERSIZE;
unsigned long JitterHistogram1[JITTERSIZE]={0,};

unsigned static long time12 = 0;  // time at previous ADC sample
unsigned long time22 = 0;         // time at current ADC sample
long MaxJitter2;             // largest time jitter between interrupts in usec
unsigned long const JitterSize2=JITTERSIZE;
unsigned long JitterHistogram2[JITTERSIZE]={0,};

Sema4Type jitterLock;

void Jitter_Init(void) {
    OS_InitSemaphore(&jitterLock, 0);
}

void Jitter(void) {
    OS_bWait(&jitterLock);
    UART_OutStringCRLF("Periodic thread 1 jitter data:");
    UART_OutString("Max jitter: "); UART_OutUDec(MaxJitter1); UART_OutString(" x 0.1 us");
UART_OutCRLF();
    UART_OutStringCRLF("Jitter distribution: ");
    for(int i = 0; i < JITTERSIZE; ++i) {
        UART_OutString("i="); UART_OutUDec(i); UART_OutString(": ");
        for(int j = 0; j < JitterHistogram1[i] >> 2; ++j)
          UART_OutChar('=');
        UART_OutChar(' '); UART_OutUDec(JitterHistogram1[i]); UART_OutCRLF();
    }

    UART_OutStringCRLF("Periodic thread 2 jitter data:");
    UART_OutString("Max jitter: "); UART_OutUDec(MaxJitter2); UART_OutString(" x 0.1 us");
UART_OutCRLF();
    UART_OutStringCRLF("Jitter distribution: ");
    for(int i = 0; i < JITTERSIZE; ++i) {
        UART_OutString("i="); UART_OutUDec(i); UART_OutString(": ");
        for(int j = 0; j < JitterHistogram2[i] >> 2; ++j)
          UART_OutChar('=');
        UART_OutChar(' '); UART_OutUDec(JitterHistogram2[i]); UART_OutCRLF();
    }
    OS_bSignal(&jitterLock);
}
```

2) Blocking Semaphores.

```
void OS_bWait(Sema4Type *sema) {
    OS_DisableInterrupts();

    if(sema->Value < 0)
        BlockThread(sema);
    else
        sema->Value = -1;

    OS_EnableInterrupts();
}

void OS_Wait(Sema4Type *sema) {
    OS_DisableInterrupts();
```

```
    --sema->Value;

    if(sema->Value < 0)
        BlockThread(sema);

    OS_EnableInterrupts();
}

void BlockThread(Sema4Type *sema) {
    OS_DisableInterrupts();

    tcbType *t = sema->next;
    if(t) {
      while(t->bNext)
        t = t->bNext;
      t->bNext = RunPt;
    } else
      sema->next = RunPt;

    RunPt->blocked = 1;
    RunPt->bNext = 0;

    //2 trigger pendsv
    NVIC_INT_CTRL_R |= 0x10000000;
    OS_EnableInterrupts();
}

void OS_bSignal(Sema4Type *semaPt) {
    OS_DisableInterrupts();
    tcbType *t = semaPt->next;
    if(t != NULL) {
        semaPt->next = semaPt->next->bNext;
        t->bNext = NULL;
        t->blocked = 0;
    } else
      semaPt->Value = 0;

    //2 trigger pendsv
    NVIC_INT_CTRL_R |= 0x10000000;
    OS_EnableInterrupts();
}

void OS_Signal(Sema4Type *semaPt) {
    OS_DisableInterrupts();
    ++semaPt->Value;
    tcbType *t = semaPt->next;
    if(t != NULL) {
        semaPt->next = semaPt->next->bNext;
        t->bNext = NULL;
        t->blocked = 0;
    }

    //2 trigger pendsv
    NVIC_INT_CTRL_R |= 0x10000000;
    OS_EnableInterrupts();
}
```

3) Priority Scheduler.

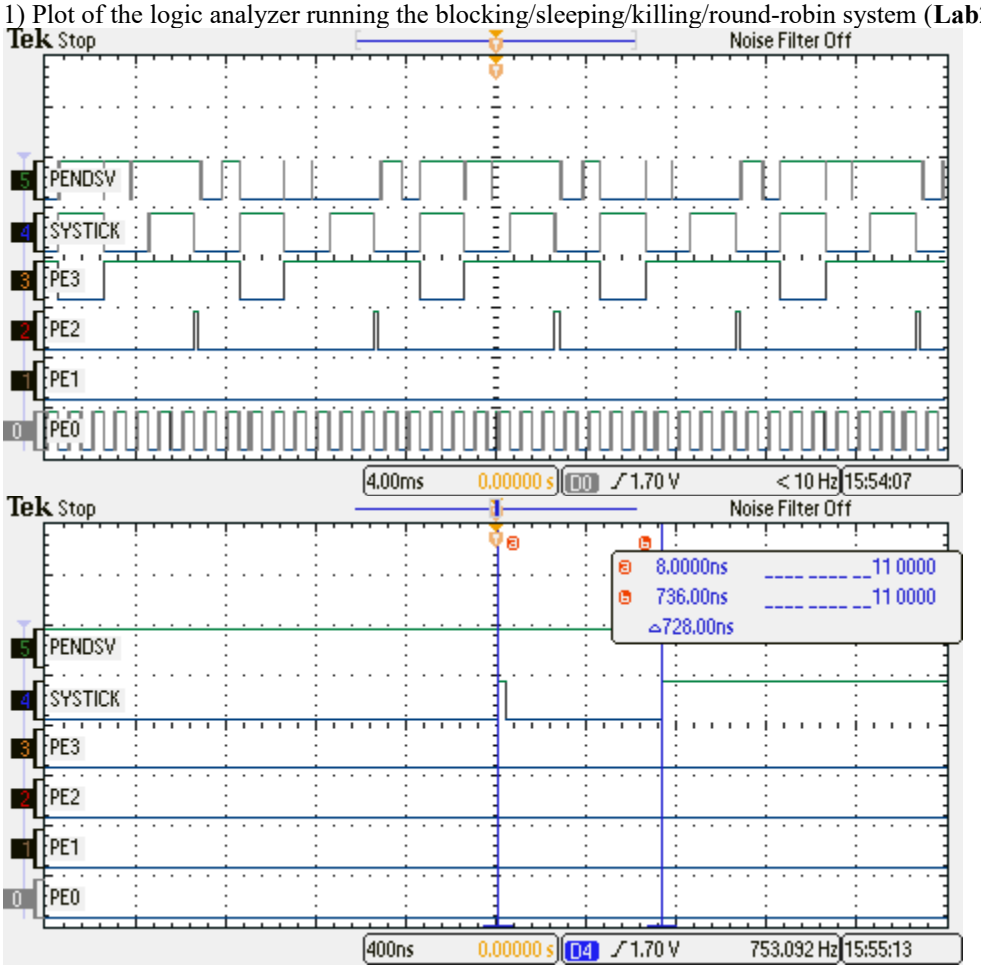
```
PendSV_Handler: .asmfunc
 CPSID   I                  ; 2) Prevent interrupt during switch
  .if DEBUG      ; toggle heartbeat
  LDR     R0, PF2Ptr            ; toggle heartbeat
  LDR     R1, [R0]
  EOR     R1, #0x04
  STR     R1, [R0]
  EOR     R1, #0x04
  STR     R1, [R0]
  .endif
  PUSH    {R4-R11}           ; 3) Save remaining regs r4-11
  MOV     R0, #0xD000D000
  PUSH    {R0}
```

```
  ADD     SP, #4
  LDR     R0, RunPtAddr         ; 4) R0=pointer to RunPt, old thread
  LDR     R1, [R0]           ;    R1 = RunPt
  STR     SP, [R1]           ; 5) Save SP into TCB

  LDR     R3, [R1,#4]        ; R3 is an iterator, R3 = RunPt->next
  MOV     R1, R3
  MOV     R5, R3             ; R5 nextThread
  MOV     R4, #0x7FFF    ; R4 priority

PendSV_Priority:
  LDR     R2, [R1,#16]       ; R3->sleep
  CMP     R2, #0
  BNE     PendSV_Next
  LDR     R2, [R1,#28]       ; R3->blocked
  CMP     R2, #0
  BNE     PendSV_Next
  LDR     R2, [R1,#24]       ; R3->pri
  CMP     R2, R4             ; R2 < R4 ?
  BGE     PendSV_Next
PendSV_Update:
    MOV     R5, R3
  MOV     R4, R2
PendSV_Next:
    LDR     R3, [R3,#4]        ; increment
  CMP     R3, R1
  BNE     PendSV_Priority   ; R3 != RunPt
  STR     R5, [R0]           ; update RunPt
  LDR     SP, [R5]

  POP     {R4-R11}           ; 8) restore regs r4-11
  .if DEBUG
  LDR     R0, PF2Ptr           ; toggle heartbeat
  LDR     R1, [R0]
  EOR     R1, #0x04
  STR     R1, [R0]

  PUSH {LR}
  MOV R0, #0
  BL addTInfo
  POP {LR}
  .endif
  CPSIE   I                  ; 9) tasks run with interrupts enabled
  BX      LR                 ; 10) restore R0-R3,R12,LR,PC,PSR
  .endasmfunc

SysTick_Handler:  .asmfunc     ; 1) Saves R0-R3,R12,LR,PC,PSR
  CPSID   I                  ; 2) Prevent interrupt during switch
  .if DEBUG      ; toggle heartbeat
  LDR     R0, PF1Ptr           ; toggle heartbeat
  LDR     R1, [R0]
  EOR     R1, #0x02
  STR     R1, [R0]
  EOR     R1, #0x02
  STR     R1, [R0]
  .endif
  PUSH    {R4-R11}           ; 3) Save remaining regs r4-11
  MOV     R0, #0xD000D000
  PUSH    {R0}
  ADD     SP, #4
  LDR     R0, RunPtAddr         ; 4) R0=pointer to RunPt, old thread
  LDR     R1, [R0]           ;    R1 = RunPt
  STR     SP, [R1]           ; 5) Save SP into TCB

  LDR     R3, [R1,#4]        ; R3 is an iterator, R3 = RunPt->next
  MOV     R5, R3             ; R5 nextThread
  MOV     R4, #0x7FFF    ; R4 priority

SysTick_Priority:
  LDR     R2, [R1,#16]       ; R3->sleep
  CMP     R2, #0
  BNE     SysTick_Next
```

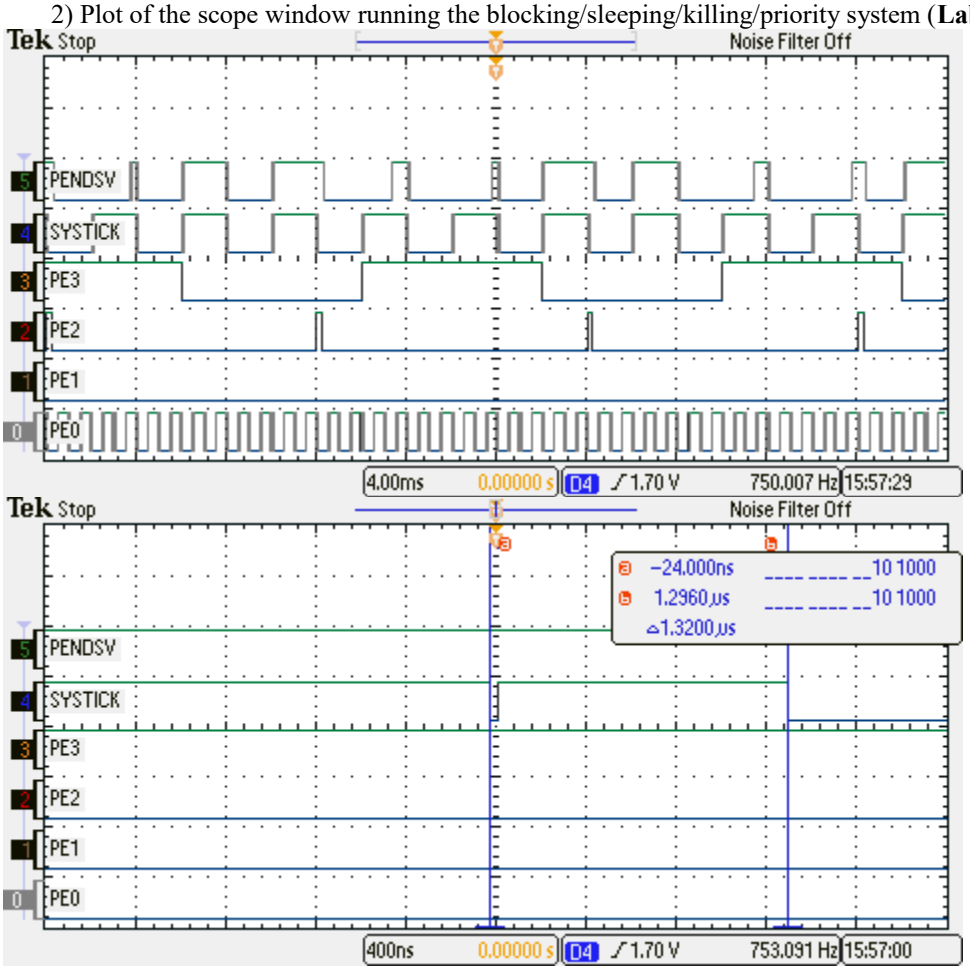
```
  LDR     R2, [R1,#28]        ; R3->blocked
  CMP     R2, #0
  BNE     SysTick_Next
  LDR     R2, [R1,#24]        ; R3->pri
  CMP     R2, R4              ; R2 < R4 ?
  BGE     SysTick_Next
SysTick_Update:
  MOV     R5, R3
  MOV     R4, R2
SysTick_Next:
  LDR     R3, [R3,#4]         ; increment
  CMP     R3, R1
  BNE     SysTick_Priority    ; R3 != RunPt
  STR     R5, [R0]            ; update RunPt
  LDR     SP, [R5]

  POP     {R4-R11}            ; 8) restore regs r4-11
  .if DEBUG
  LDR     R0, PF1Ptr            ; toggle heartbeat
  LDR     R1, [R0]
  EOR     R1, #0x02
  STR     R1, [R0]
  .endif
  CPSIE   I                   ; 9) tasks run with interrupts enabled
  BX      LR                  ; 10) restore R0-R3,R12,LR,PC,PSR
 .endasmfunc
```

D) Measurement Data

1) Plot of the logic analyzer running the blocking/sleeping/killing/round-robin system (**Lab2.c**)



2) Plot of the scope window running the blocking/sleeping/killing/priority system (**Lab2.c**)



In comparison, our priority scheduler does indeed have higher overhead of 1.32us versus 0.728us overhead for Round Robin. A noticeable consequence of the priority scheduler is that the consumer (PE2) and the display (PE3) runs less often.

3) Table like Table 3.1 each showing performance measurements versus sizes of the Fifo and timeslices

Originally, we got really bad data loss due to a bug; the bug was the fifo calling OS_Suspend() since we used a semaphore which blocks. The priority scheduler basically makes the bug more apparent, while it used to work surprisingly well in the spinlock implementation (since the order of running threads is always the same). Furthermore, calling OS_Suspend() during an ISR likely caused another bug where our OS crashes when we run ButtonWork().

		Spinlock / Round Robin			Block / Round Robin			Block / Priority		
FIFO Size	Tslice (ms)	Data Lost	Jitter (us)	PID Work	Data Lost	Jitter (us)	PID Work	Data Lost	Jitter (us)	PID Work
256	2				0	0	6577	3805	0	3912
64	2				66	0	6577	3997	0	3912
16	2				114	0	6577	4045	0	3912
4	2	0	0	3395	126	0	6577	4057	0	3911
256	4	0	0	3396	0	0	6703	4791	0	6345
64	4	0	0	3396	67	0	6704	4983	0	6345
16	4	0	0	3396	115	0	6704	5031	0	6345
4	4	0	0	3396	127	0	6704	5043	0	6345
256	10				0	0	6611	5283	0	6743
64	10				67	0	6611	5475	0	6743
4	10				127	0	6611	5535	0	6743

When we fixed the bug, we got:

		Spinlock / Round Robin			Block / Round Robin			Block / Priority		
FIFO Size	Tslice (ms)	Data Lost	Jitter (us)	PID Work	Data Lost	Jitter (us)	PID Work	Data Lost	Jitter (us)	PID Work
256	2				0	19	4490	0	12	6306
64	2				0	15	4483	0	13	6098
16	2				0	15	4507	0	13	6098
4	2	0	0	3395	0	20	4491	999	13	6098
256	4	0	0	3396	0	22	2331	0	12	6483
64	4	0	0	3396	0	22	2322	0	13	6268
16	4	0	0	3396	0	22	2322	0	13	6268
4	4	0	0	3396	0	22	2323	4497	13	6269
256	10				0	25	8878	0	12	6590
64	10				0	21	8925	0	13	6370
16	10				0	21	8925	2324	13	6371
4	10				0	21	8925	6331	13	6372

We noticed that the jitter increased.

Between Block/RoundRobin and Block/Priority, the Priority scheduler has a more consistent maximum jitter and not very dependent on the time slice.

Furthermore, the Priority Scheduler has a more consistent PID work as the time slice decreases while Round Robin has very high PID work with a large timeslice but decreases with a smaller time slice.

We started getting data loss in the priority scheduler, but as long as the fifo is big enough, we can avoid data loss; or by decreasing the time slice. A possibility of why we get data loss is that the priority scheduler runs all the threads at a different period in comparison to round robin; thus, when a background thread interrupts the consumer, the background thread must finish and the FIFO is full. This likely explains why we get jitter since our threads run atomic OS functions which prevent the background thread from running at the correct time; Spinlock ensures the period between threads to stay constant (ie SysTick timeslice).

E) Analysis and Discussion (2 page maximum). In particular, answer these questions

1) How would your implementation of **`OS_AddPeriodicThread`** be different if there were 10 background threads? (Preparation 1)

We add periodic threads as foreground threads: We can always make the threads higher priority than all foreground threads. Then each ISR will simply add a new thread for the background thread. (+) We have more flexibility with how we can control each thread. (-) This will increase latency as we will need to do a context switch, and some overhead to manage the background threads.

Obviously, if we wanted more background threads, we either have to have more hardware timers or implement the background threads similar to the way foreground threads are implemented. Obviously, we have tradeoffs between each, and which one we implement depends on what we need.

2) How would your implementation of blocking semaphores be different if there were 100 foreground threads? (Preparation 4)

Not much different, our implementation of blocking semaphores is to have both a blocking field in the TCB and each semaphore has a linked list to the corresponding blocked TCBs. When we want to block, we simply set the TCB field to blocked and add the TCB to the tail of the semaphore list. When we want to unblock, we set the head of the semaphore list to unblocked and relink the list again; eventually, the scheduler will run the unblocked thread.

As of now, we don't have a tail pointer for the semaphore. But if we have more threads, we will need a tail pointer to make the block function have constant time.

3) How would your implementation of the priority scheduler be different if there were 100 foreground threads? (Preparation 5)

We can optimize the search function by using a heap which always stay sorted in ascending priority; we will need a heap for sleeping threads or blocked threads (or just add a large constant to keep priority low).

Unless the overhead for a linear search through 100 threads is too high, we can simply use the simplest implementation. *using a heap, we can peek at the top element which costs O(1), any change in priority will need heap insert and remove which cost O(log n). **we will not add or remove using a linked list O(0) but O(n) to peek.

4) What happens to your OS if all the threads are blocked? If your OS would crash, describe exactly what the OS does?

The OS crashes because of a while loop checking for available threads to run.

What happens to your OS if all the threads are sleeping? If your OS would crash, describe exactly what the OS does?

The OS still crashes since interrupts are disabled.
If you answered crash to either or both, explain how you might change your OS to prevent the crash.
Have an idle thread.

5) What happens to your OS if one of the foreground threads returns? E.g., what if you added this foreground

```
void BadThread(void){ int i;
  for(i=0; i<100; i++){};
  return;
}
```

What should your OS have done in this case? Do not implement it, rather, with one sentence, say what the OS should have done? Hint: I asked this question on an exam.
It will jump to wherever LR points to, which should be 0x14141414, This will likely hardfault. The OS should ideally call OS_Kill() when the function returns. We can initialize the LR in the TCB to point to OS_Kill() and return would call BX LR.

6) What are the advantages of spinlock semaphores over blocking semaphores? What are the advantages of blocking semaphores over spinlock?
Interrupts won't have to be disabled for spinlocks, and the scheduler is simpler. We get to implement priority with blocks and IO bound tasks can be blocked instead of wasting the whole time slice.

7) Consider the case where thread T1 interrupts thread T2, and we are experimentally verifying the system operates without critical sections. Let **n** be the number of times T1 interrupts T2. Let **m** be the total number of interruptible locations within T2. Assume the time during which T1 triggers is random with respect to the place (between which two instructions of T2) it gets interrupted. In other words, there are **m** equally-likely places within T2 for the T1 interrupt to occur. What is the probability after **n** interrupts that a particular place in T2 was never selected? Furthermore, what is the probability that all locations were interrupted at least once?

P(one particular spot never interrupted | n interrupts) = ((m-1)/m)^n; This is not the same as the probability of having at least one spot never interrupted.

if n<m
P(every spot interrupted once | n interrupts) = 0
else
http://math.stackexchange.com/questions/124291/whats-the-probability-that-theres-at-least-one-ball-in-every-bin-if-2n-balls-a
The solution does not have a closed form where:

$$P(n,m)=\frac{m!S(n,m)}{m^n}=\frac{1}{m^n}\sum_{j=0}^{m}(-1)^{j+1}{}_mC_j\,j^n$$

S(n,m) is the Stirling relation of the second kind and it has no closed form.
https://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind
The Stirling function describes the number of unique ways to arrange n objects into m "non-empty subsets"

In addition, if we want the probability of having at least one spot never interrupted, we only need to do 1-P(n,m).

We attempted to solve the problem by counting the possible combinations where all buckets are filled, the idea is to count after each trial (interrupt) to see whether that combination fills 1 bucket (time place), 2 buckets, 3 buckets, …, m buckets (at least 1 in every bucket).
So at every trial, if 1 bucket is only filled at the previous trial, then it has 1 (1/m probability) way of still remaining 1 bucket and m-1 ways to become 2 buckets.
This gives us a recursive function which describes every bucket:
(1 bucket)(n+1) = (1 bucket)(n)
(2 buckets)(n+1) = (m-1)*(1 bucket)(n) + 2*(2 bucket)(n)
…
(k buckets)(n+1) = (m+1-k)*(k-1 buckets)(n) + k*(k buckets)(n)
…

(m buckets)(n+1) = (m-1 buckets)(n) + m*(m buckets)(n)

Below is the pattern with 3 buckets and n trials, this matches with the above answer.

n	1 buckets	2 buckets	3 buckets	sum	P(all)
0	0	0	0	0	
1	3	0	0	3	0
2	3	6	0	9	0
3	3	18	6	27	0.2222222222
4	3	42	36	81	0.4444444444
5	3	90	150	243	0.6172839506
6	3	186	540	729	0.7407407407
7	3	378	1806	2187	0.8257887517
8	3	762	5796	6561	0.8834019204
9	3	1530	18150	19683	0.9221155312
10	3	3066	55980	59049	0.9480262155
11	3	6138	171006	177147	0.9653338753
12	3	12282	519156	531441	0.9768836051
13	3	24570	1569750	1594323	0.9845871884
14	3	49146	4733820	4782969	0.9897241651
15	3	98298	14250606	14348907	0.9931492343
16	3	196602	42850116	43046721	0.9954327532
17	3	393210	128746950	129140163	0.9969551455
18	3	786426	386634060	387420489	0.9979700893
19	3	1572858	1160688606	1162261467	0.9986467236
20	3	3145722	3483638676	3486784401	0.9990978149
21	3	6291450	10454061750	10460353203	0.999398543
22	3	12582906	31368476700	31381059609	0.9995990285
23	3	25165818	94118013006	94143178827	0.9997326857
24	3	50331642	2.82379E+11	2.8243E+11	0.9998217904
25	3	100663290	8.47188E+11	8.47289E+11	0.9998811936
26	3	201326586	2.54166E+12	2.54187E+12	0.9999207957
27	3	402653178	7.62519E+12	7.6256E+12	0.9999471972
28	3	805306362	2.2876E+13	2.28768E+13	0.9999647981