ECE 560: EMB. SYS. ARCHITECTURES
PROJECT 1 REPORT

Arpad Voros

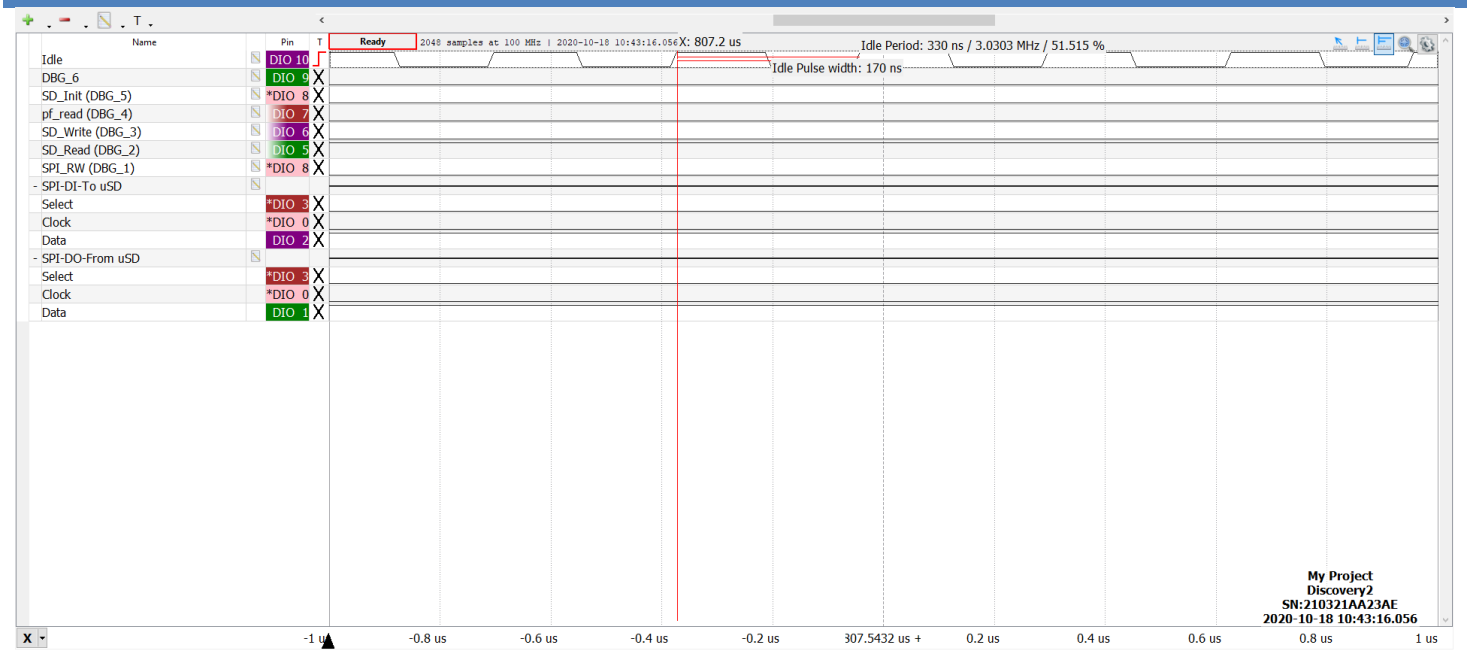
Unity ID: aavoros

INTRODUCTION

In this project, we are exploring how to recover idle time using a real time operating system. This is important for the same reason why interrupts exist: we don't want to continuously poll a system when the MCU can be performing some other function. For systems that require waiting time, it would be ideal to recover idle time during this waiting period.

Project 1 decides to read data from an SD Card, which communicates over a SPI interface. The SD Card follows a specific protocol when being read and written to. After sending a read block command, the SD Card gathers itself and responds with a falling pulse on the SPI channel to indicate it's ready to be read followed by the corresponding data block. During this waiting period, we can create use osDelay from RTOS to wait for a certain amount of 'ticks' (occurring every ms) to pass to perform other operations before returning to the thread. However, osDelay is inconsistent with precision timing as the number of 'ticks' independently occur to the time at which osDelay is called. Meaning, osDelay(2) returns to the thread anywhere between 1ms – 2ms, osDelay(3) is 3ms – 4ms, so on and so forth. For more precise delays, we can utilize the periodic interrupt timer (PIT) to count down from a value and generate an interrupt when its done. With this precision, we can block the thread and recover idle time and significantly more accurate rate.

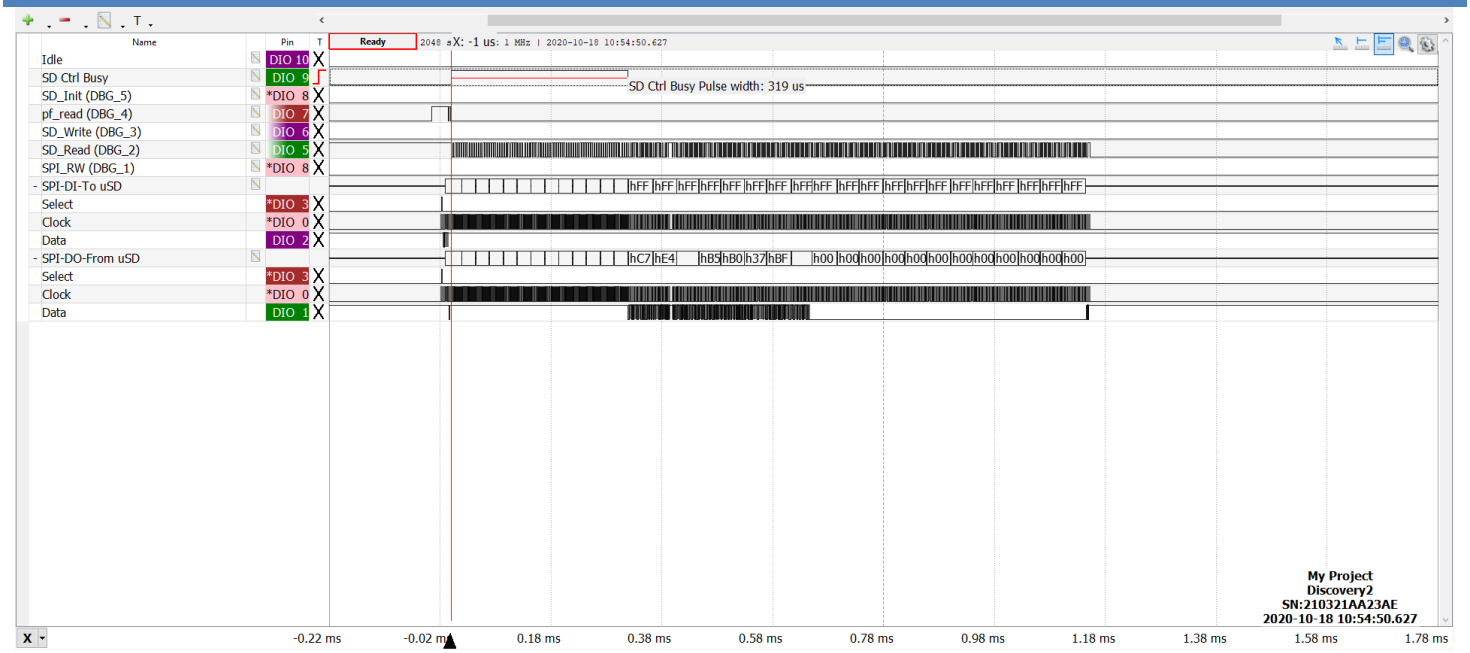
ANALYZE IDLE THREAD TIMING



- **How much time does it take for the loop in osRtxIdleThread to execute one iteration?**
    - Since DEBUG_TOGGLE is used (plugged into DIO10, as seen in the figure above), every time the loop repeats the channel is toggled, meaning one half period of our signal is the time for each loop. In this case, it fluctuated around 160-180ns, being highly consistent at 170ns.

- **How many iterations would happen in one millisecond if only the idle thread ran, and nothing else?**
    - 1 ms / 170 ns = (1 * 10 ^ -3) / (170 * 10 ^ -9) = around 5882 times per ms

ANALYZE SD TIMING

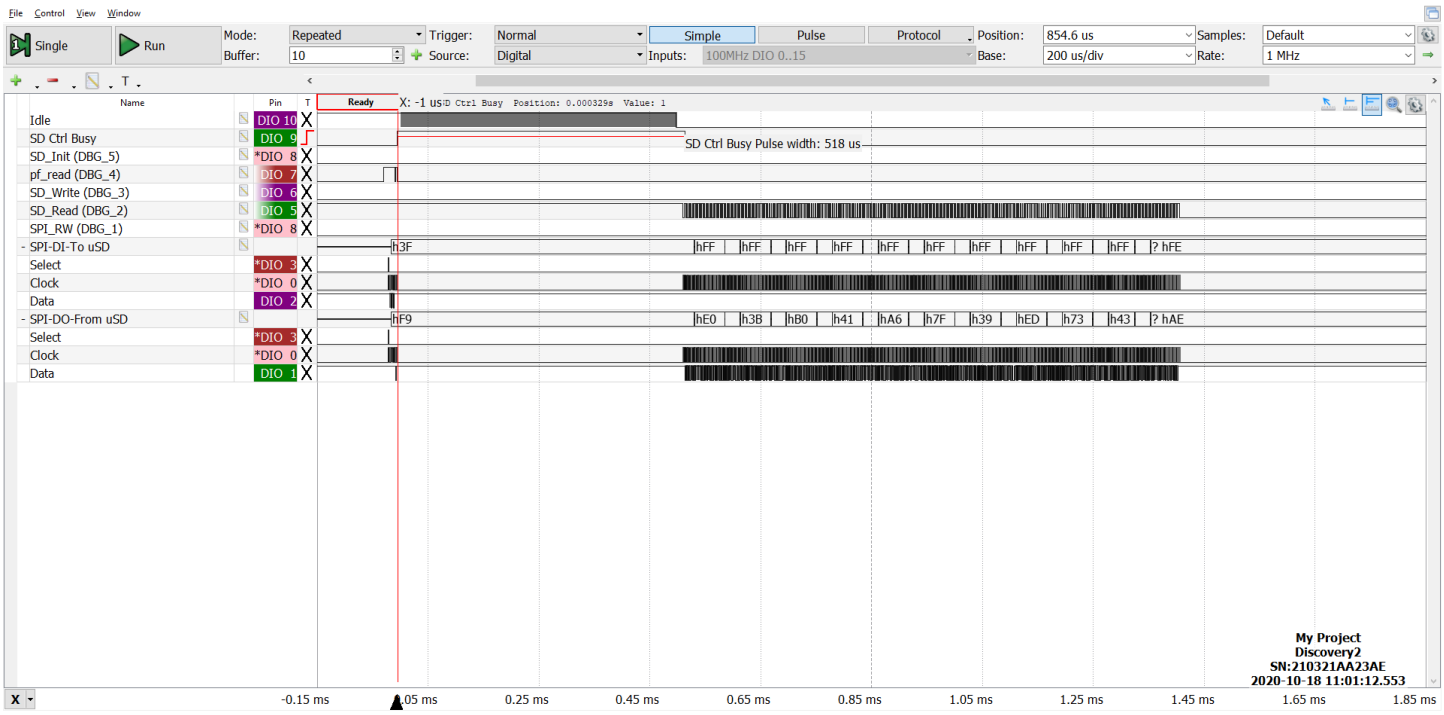


After adding the SD Ctrl Busy channel visible on the AD2, we observe that the pulse switches erratically between 300 us and ~460 us without any sort of idle time consideration.

- **What are the values of the statistics reported on the display? What do they indicate? How do they relate to the length of SD Ctlr Busy?**
    - Blocks: 1159, Total Time: 11554 -> These indicate the time it took to read as well as the number of blocks read from the SD card
    - Idle time and loops are both 0 since we are not blocking on this thread.

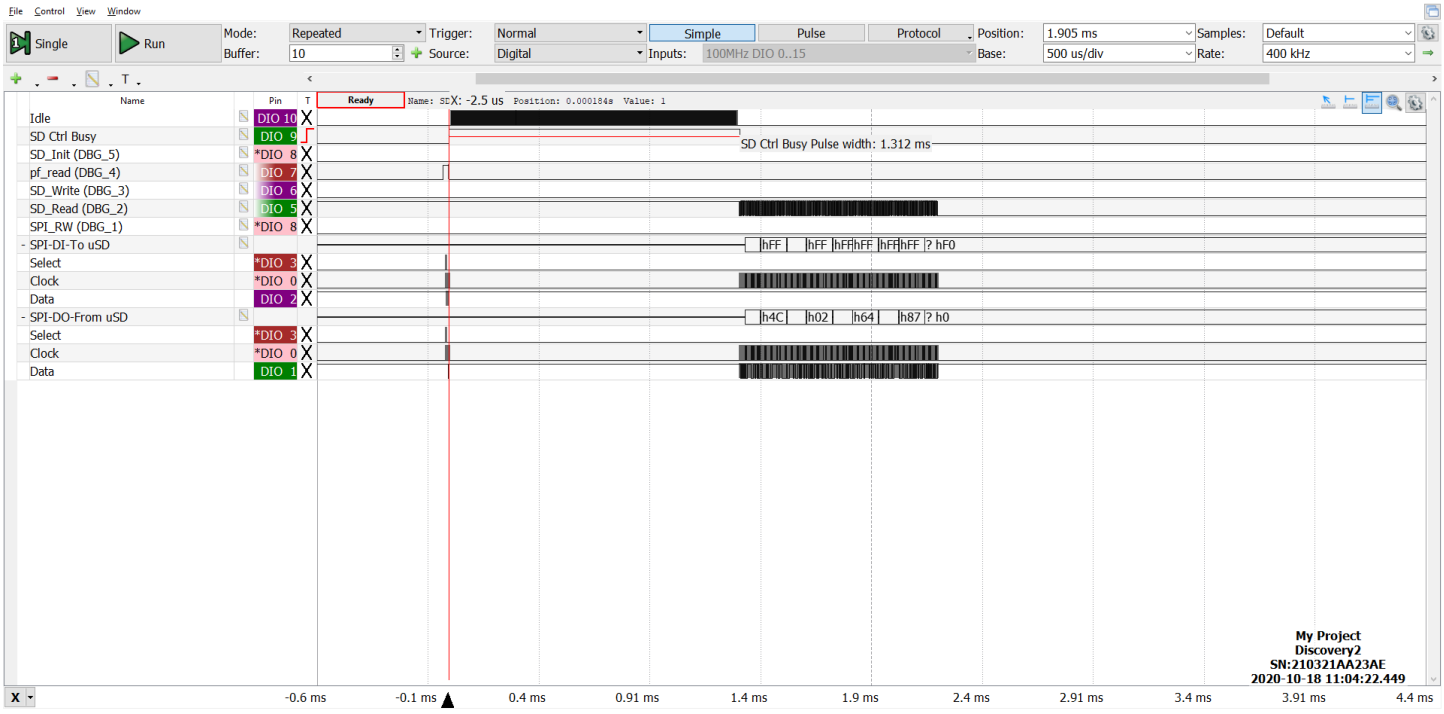
ANALYZE OSDELAY TIMING

- **What range of delay times do you see, and how do they compare with the requested delay?**
    - The delay ranges from ~300 – 1000 us. The higher end of the value occurs occurring because the SD card is still reading (minimum being ~300 us), which means osDelay(1) return anywhere between 0 ms – 0.3 ms, but the SD card still keeps busy. The higher end of this occurs because osDelay(1)'s upper value of return is 1 ms.



osDelay(1)

- **What are the values of the statistics reported on the LCD? What is the fraction of time the idle thread executes? What do they indicate has changed from the previous case?**
    - Blocks Read: 1159, Total Time: 11870 -> we can notice that the average time takes a little bit longer than with no osDelay, since osDelay(1) blocks (on average) slightly higher than the average SD card read time (~300 us – ~460us)
    - Idle loops: ~3494000 (variable), Idle time: 594 -> these non-zero values indicate that we recovered idle time while waiting for the reading operation.
    - The fraction of the time the idle thread executes lasts approximately the same amount as the SD Ctrl Busy thread, ~300 us – 1 ms

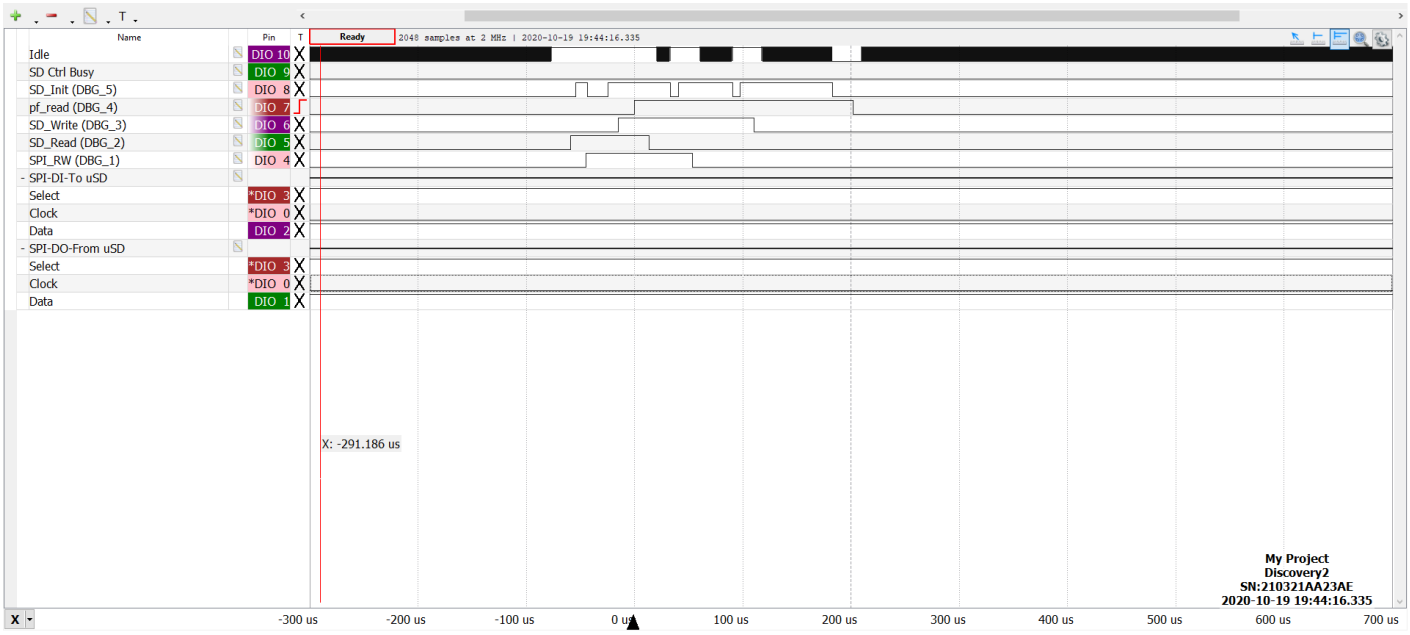


osDelay(2)

- **What are the values of the statistics reported on the LCD? What is the fraction of time the idle thread executes? What do they indicate has changed from the previous case?**
    - 1 - 1.99 ms now, since osDelay(2) is used
    - Blocks Read: 1159, Total Time: 12954 -> significantly more time is used because of the larger delay
    - Idle loops: ~10461000, Idle time: 1778 -> recovered more idle time, obviously, due to the larger osDelay
    - There is on average 3 times the number of idle time/loops as previously.

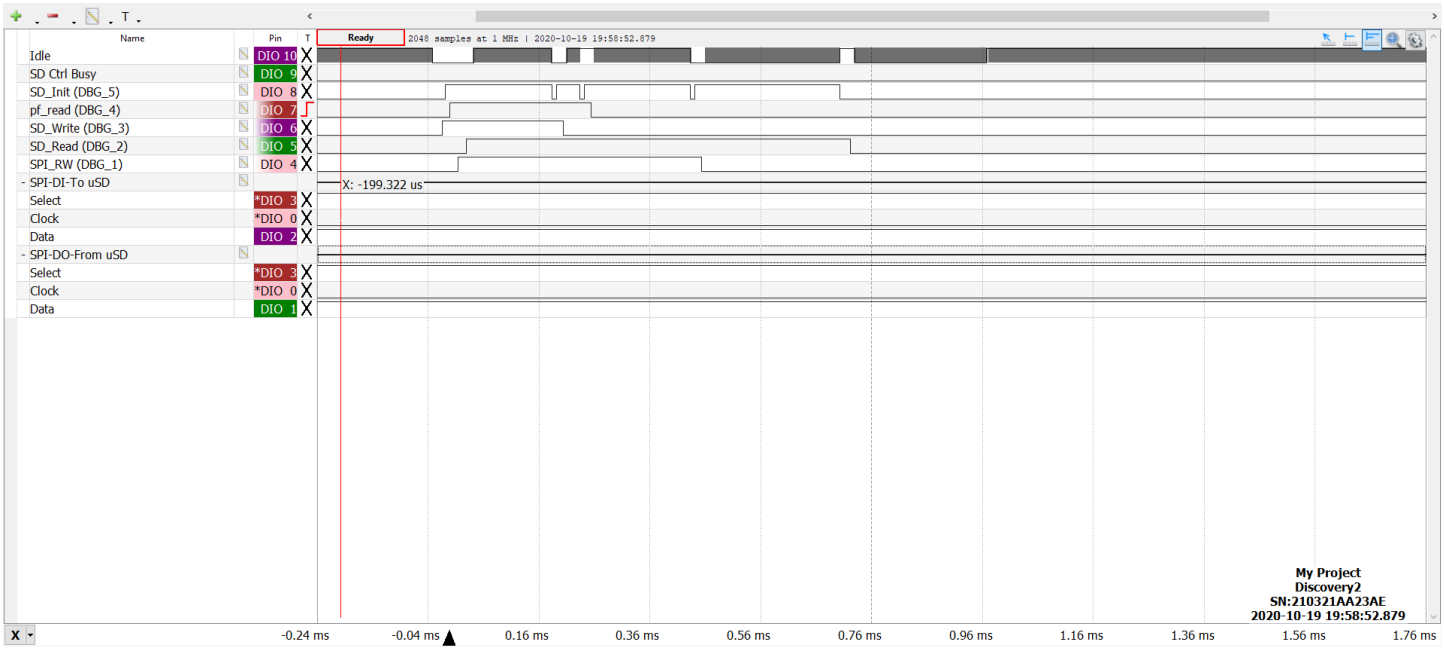
ECE 560: EXPLAIN PRECISION DELAY SOFTWARE DESIGN AND ANALYZE TIMING

- **Provide logic analyzer screenshots showing overlapping delays which prove that your system actually works.**



My precision delay implementation, using the delay values provided of 28 us, 10 us, 86 us, and 179 us

- **How much time overhead does your precision delay mechanism add to the requested delay?**
    - The pulse widths above are 98us for 28us, 72us for 10us, 125us for 86us, and 202us for 179us, which results in an average value of 48.5us delay. However, with larger delays, this number improves slightly.
    - The top signal (DIO 8, DBG_5) is set when PIT_Start is called and is reset upon entering the interrupt for PIT->CHANNEL[0]. This was used for debugging purposes, as well as a visual to see how long my PIT timers last. In the figure above, the first pulse is exactly 10us (since the first thread to execute here is the one using DBG_2, thread T2 = 10us ideal delay.
- **Does the overhead vary with the number of times a precision delay is interrupted by another channel? If so, explain why.**
    - Yes, but only in some circumstances. After fiddling around, I noticed when values reach sub average-overhead levels (i.e. below 50us) and multiple of them are called in a row, then the overhead varies significantly higher than the average overhead value. In addition, the first delay that is called takes approximately 10-20us to complete than the rest:



Here, we see the following delays (larger than the test one), showing the set and the real value:

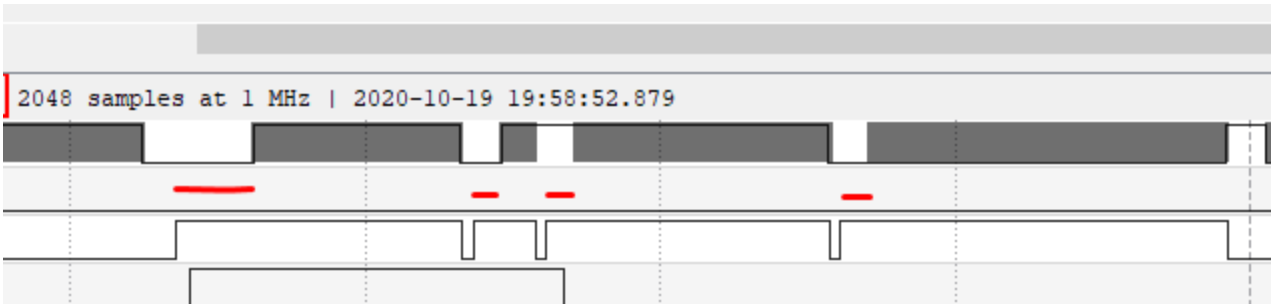
DIO Channel	Set Delay	Actual Delay	Order Called	Difference
DIO 4	400us	440us	3	40us
DIO 5	670us	693us	4	23us
DIO 6	150us	218us	1	68us
DIO 7	200us	254us	2	54us

We make two evident observations with this algorithm I made:

1. The order being called corresponds to the length of the delay. Since the shortest requested delay returns first, it calls osDelay(10) first, meaning on average it will come out of osDelay sooner than its thread counterparts
2. The overhead of the delays are significantly off at the beginning and get increasingly accurate as time goes on.

Here, the average overhead was 46.5us, so a 2 us improvement from last time. Not much, however, you can see a significant improvement with the percentage of idle time recovered.

In the figure directly above, I showed larger delays to show the delay corresponding between the moment PIT_Start is called versus when the idle thread begins running. I realised that for the first call of PIT_Start (shown by DIO 8), there is a consistent 50us delay before the idle thread starts running. This is precisely why in the shorter delays (shown in the previous figure), there is absolutely no idle time recovered for the first delays of 10 & 28us. Once this 50us passes, then each subsequent PIT_Start will be followed by an idle thread in 20us. This can be shown in the figure below:



First line representing the first delay of 50us before idling, then subsequently 20us each time

- **How consistent are the time delays? Use your logic analyzer or oscilloscope to find the minimum and maximum timing errors.**
    - Though there are varying delays, the delays themselves are consistent. They do not oscillate, grow, decrease, etc., but stay consistent down to 0.2 us. This slight fluctuation, I'm assuming, comes from the PIT interrupt itself.

I tried hard to decrease the average overhead, but with no avail. Due to this large overhead, it's safe to say that my algorithm is not optimal. But I did take into consideration the following:

1. If multiple threads end on the same tick of PIT (I know it's rare), I set both of their osFlags immediately rather than one after another
2. Some function delays can be calculated, so there is one instance where I consider the time it takes to operate.

It also took a significant amount of time to return from the IRQ Handler to the thread with the raised flag. If that had happened earlier, the DEBUG_STOP within each of those threads could have potentially ended earlier. That is why I included DIO 8, so we can see when the IRQ Handler is actually entered and exited and can observe much more precise delays. The RTOS with it's flag checking and thread returning is adding to the delay upon return.

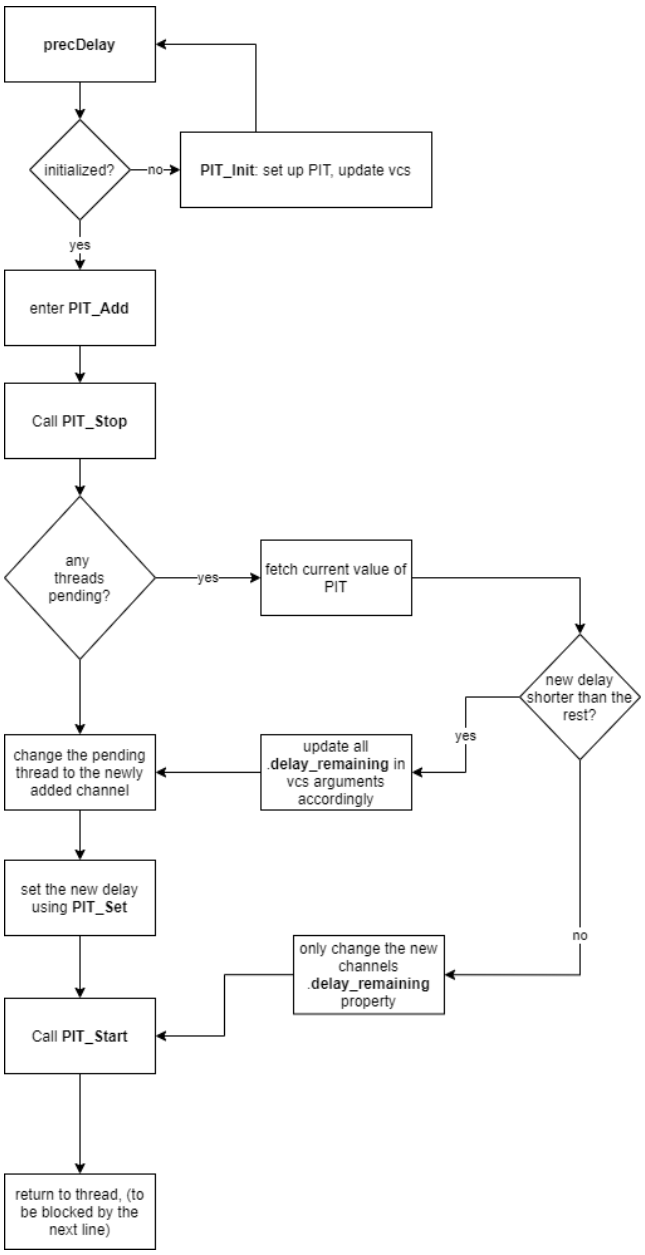
The functions used, with descriptions, are:

1. **precDelay**: the function to be called when wanting a precision delay
2. **PIT_Init**: initialize the first instance of each virtual PIT channel
3. **PIT_Add**: adds a new instance of the timer for a specified channel
4. **PIT_Set**: sets the value of the PIT (loads value)
5. **PIT_Start:** enables the counter
6. **PIT_Stop**: disables the counter
7. **PIT_Handover**: hands over the timer to the next virtual channel
8. **PIT_IRQHandler**: the IRQ handler for the PIT

Some important globals used include:

1. **virtualChannel vcs[4]**: struct array of virtualChannel for each channel, with the following properties
    a. thread_flag – the flag to be set, once this channel finishes
    b. thread_id – the ID of the thread to return to, once this channel finishes
    c. delay_remaining – the amount of delay (in PIT increments) remaining for this virtual channel before returning to its thread
2. **pending**: an unsigned char which sets values 1, 2, 4, 8 (4 LSB bits) to 1 if the corresponding channel is pending its return. Meaning, if pending = 0b0001, then upon entering the IRQ handler, the first channel, channel 0, will set its corresponding osFlag. If there are multiple pending bits, like 0b0110, then channels 1 and 2 will both set their osFlags once the timer expires.
3. **initialized**: an unsigned char, not particularly useful. But each time that precDelay is called, it checks to see if PIT_Init has been called. If so, it skips PIT_Init and immediately goes to add the timer using PIT_Add.

A brief idea on how the flow works:



Adding a channel

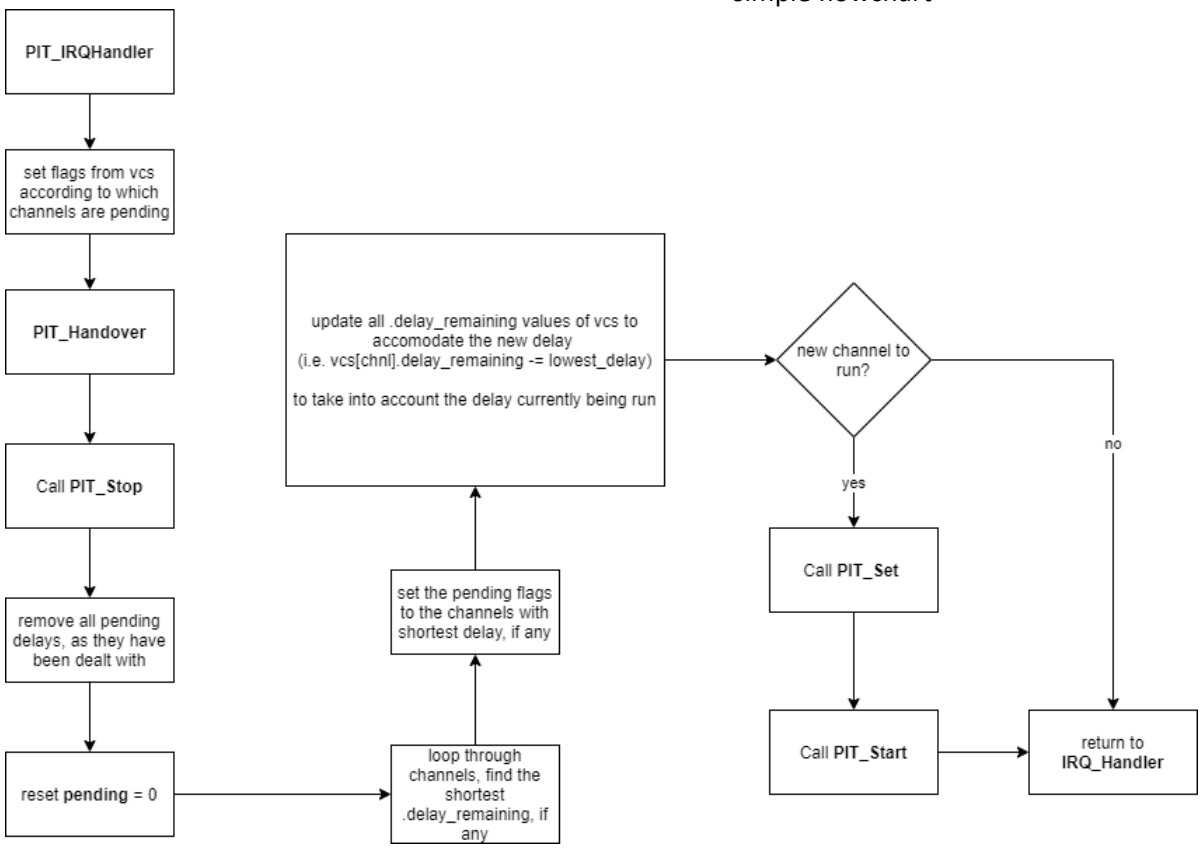
The only aspect to elaborate on, is when I say “update all .delay_remaining vcs values accordingly” I mean there I consider the current value of the PIT, the delay being added, etc.

The .delay_remaining value simply holds the value which the PIT should be loaded with when it’s that channels time to run. Example:

- Channel 0 is initialized with 50us
- PIT begins counting down
- Channel 1 is initialized to 30us
- The PIT has counted to 2us, so its current value is 48us. Channel 1 needs to be returned to sooner, so:
- **pending** = 0b0001 -> 0b0010
  vcs[0].delay_rem = 50us -> (48 – 30) = 18us
  vcs[1].delay_rem = (not set) -> 30us
- After 30us when the IRQ Handler is called, it sees that the pending channel is 0b0010 = channel 1, so it sets the flags for channel 1 and enters PIT_Handover
- PIT_Handover (seen below) finds the shortest delay, which in this case is only 18us from channel 0
- The delay is set to 18us and the timer starts again
- After 18us, channel 0 would have experienced a ‘time out’ by 2us (beginning) + 30us (channel 1 waiting) + 18us (calculated delay remaining) for a total of 50us, which is what we wanted

PIT_Handover

Briefly described in the paragraph above, as well as this simple flowchart



Listings/Screenshots of the program:

```
130 void PIT_Add(uint32_t virtual_channel, uint32_t delay){
131   uint32_t pit_val;
132
133   // stop the timer for quick calculation
134   PIT_Stop();
135   // fetch the current value in the PIT
136   pit_val = PIT->CHANNEL[0].CVAL;
137
138   // if there is currently a PIT running
139   if (pending) {
140     // if the delay needs precedence, then load delay into PIT and update .delay_remaining
141     if (pit_val > delay && pit_val) {
142       // update .delay_remaining of all CURRENT waiting threads (aka not the one being added)
143       // faster than for loop
144       if (vcs[T1_CHNL].delay_remaining) vcs[T1_CHNL].delay_remaining = (pending & T1_MASK) ? pit_val - delay : vcs[T1_CHNL].delay_remaining + pit_val - delay;
145       if (vcs[T2_CHNL].delay_remaining) vcs[T2_CHNL].delay_remaining = (pending & T2_MASK) ? pit_val - delay : vcs[T2_CHNL].delay_remaining + pit_val - delay;
146       if (vcs[T3_CHNL].delay_remaining) vcs[T3_CHNL].delay_remaining = (pending & T3_MASK) ? pit_val - delay : vcs[T3_CHNL].delay_remaining + pit_val - delay;
147       if (vcs[T4_CHNL].delay_remaining) vcs[T4_CHNL].delay_remaining = (pending & T4_MASK) ? pit_val - delay : vcs[T4_CHNL].delay_remaining + pit_val - delay;
148       // if there is a delay remaining, then update appropriately whether or not the thread was the
149       // first pending or whether there was still delay time
150
151       // set this channels .delay_remaining
152       vcs[virtual_channel].delay_remaining = delay;
153
154       // set pending channel
155       HARD_SET_THREAD(virtual_channel);
156       // set timer to delay
157       PIT_Set(delay);
158       // PIT->CHANNEL[0].LDVAL = PIT_LDVAL_TSV(delay);
159     } else if (pit_val == delay) {
160       // rare, but if the delays are the equal
161       SOFT_SET_THREAD(virtual_channel);
162       // set this channels .delay_remaining
163       vcs[virtual_channel].delay_remaining = delay - pit_val;
164       PIT_Set(pit_val);
165     } else { vcs[virtual_channel].delay_remaining = delay - pit_val; PIT_Set(pit_val); }
166   } else {
167   // if there is NO PIT running
168     // set pending channel
169     HARD_SET_THREAD(virtual_channel);
170     // set this channels .delay_remaining
171     vcs[virtual_channel].delay_remaining = delay;
172     // set timer to delay
173     PIT_Set(delay);
174   }
```

```c
void PIT_Handover(void) {
  uint32_t lowest_delay = 0;
  //unsigned char lowest_idx = NUM_CHANNELS + 1;

  PIT_Stop();

  // remove delay_remaining
  if (pending & T1_MASK) vcs[T1_CHNL].delay_remaining = 0;
  if (pending & T2_MASK) vcs[T2_CHNL].delay_remaining = 0;
  if (pending & T3_MASK) vcs[T3_CHNL].delay_remaining = 0;
  if (pending & T4_MASK) vcs[T4_CHNL].delay_remaining = 0;

  // reset pending
  pending = 0;

  // find the minimum delay
  for (int i = 0; i < NUM_CHANNELS; i++) {
    if (lowest_delay == 0 && vcs[i].delay_remaining) lowest_delay = vcs[i].delay_remaining;
    if (vcs[i].delay_remaining && lowest_delay > vcs[i].delay_remaining) {
      lowest_delay = vcs[i].delay_remaining;
      //lowest_idx = i;
      HARD_SET_THREAD(i);
    } else if (lowest_delay && lowest_delay == vcs[i].delay_remaining) {
      SOFT_SET_THREAD(i);
      //lowest_idx = i;
    }
  }

  // update all delays accordingly
  if (vcs[T1_CHNL].delay_remaining && !(pending & T1_MASK)) vcs[T1_CHNL].delay_remaining -= lowest_delay;
  if (vcs[T2_CHNL].delay_remaining && !(pending & T2_MASK)) vcs[T2_CHNL].delay_remaining -= lowest_delay;
  if (vcs[T3_CHNL].delay_remaining && !(pending & T3_MASK)) vcs[T3_CHNL].delay_remaining -= lowest_delay;
  if (vcs[T4_CHNL].delay_remaining && !(pending & T4_MASK)) vcs[T4_CHNL].delay_remaining -= lowest_delay;

  if (pending && lowest_delay) { PIT_Set(lowest_delay - HANDOVER_OFFSET); PIT_Start(); }
}
```

Here, you can see all the aspects described above in the flowcharts. A couple things to note

- Multiple macros to consider, like the TX_MASK to check the **pending** channel bits
- SOFT_SET_THREAD(x) and HARD_SET_THREAD(x) are macros to set **pending**, hard-set meaning using an AND operation so only 1 bit is pending, and soft-set meaning using an OR operation so it adds a pending thread (meaning it ends at the same instance, which is very unlikely)
- HANDOVER_OFFSET is a macro that equals 8.5us, as PIT_Handover takes 8.5us to complete. So I subtract it from the PIT_Set call

```c
void PIT_IRQHandler(void) {

  //clear pending IRQ
  NVIC_ClearPendingIRQ(PIT_IRQn);

  // check to see which channel triggered interrupt
  if (PIT->CHANNEL[0].TFLG & PIT_TFLG_TIF_MASK) {
    DEBUG_STOP(DBG_5);
    // clear status flag for timer channel 0
    PIT->CHANNEL[0].TFLG &= PIT_TFLG_TIF_MASK;
    // Do ISR work


    // ======================================================
    //PIT->CHANNEL[0].CVAL;
    if (T1_MASK & pending) osThreadFlagsSet(vcs[T1_CHNL].thread_id, vcs[T1_CHNL].thread_flag);
    if (T2_MASK & pending) osThreadFlagsSet(vcs[T2_CHNL].thread_id, vcs[T2_CHNL].thread_flag);
    if (T3_MASK & pending) osThreadFlagsSet(vcs[T3_CHNL].thread_id, vcs[T3_CHNL].thread_flag);
    if (T4_MASK & pending) osThreadFlagsSet(vcs[T4_CHNL].thread_id, vcs[T4_CHNL].thread_flag);

    PIT_Handover();
    //osThreadFlagsSet(vcs[T3_CHNL].thread_id, vcs[T3_CHNL].thread_flag);
    // ======================================================

    }
  if (PIT->CHANNEL[1].TFLG & PIT_TFLG_TIF_MASK) {
    // clear status flag for timer channel 1
    PIT->CHANNEL[1].TFLG &= PIT_TFLG_TIF_MASK;
    // Do ISR work
  }
}
```

The IRQ Handler is very straight forward, it sets all the osFlags if they are pending, and then hands over the PIT to the next pending channel. PIT_Start and PIT_Stop have not been altered. PIT_Set is simply the following:

```
217 void PIT_Set(uint32_t delay) {
218   // set the delay value
219   PIT->CHANNEL[0].LDVAL = PIT_LDVAL_TSV(delay);
220 }
```

Macros in timers.h

```
// ================================================================================
#define HARD_SET_THREAD(idx) (pending = (1 << idx))
#define SOFT_SET_THREAD(idx) (pending |= (1 << idx))
#define NUM_CHANNELS 4

#define MICROS_TO_PIT(us) ((24 * us) - 1)

#define T1_MASK 1
#define T2_MASK 2
#define T3_MASK 4
#define T4_MASK 8

#define T1_CHNL 0
#define T2_CHNL 1
#define T3_CHNL 2
#define T4_CHNL 3

#define T1_FLAG 16
#define T2_FLAG 32
#define T3_FLAG 64
#define T4_FLAG 128

/*#define T1_uS 28
#define T2_uS 10
#define T3_uS 86
#define T4_uS 179*/

#define T1_uS 400
#define T2_uS 670
#define T3_uS 150
#define T4_uS 200

#define OFFSET 0
#define HANDOVER_OFFSET 206

typedef struct virtualChannel_t {
  uint32_t thread_flag;
  osThreadId_t thread_id;
  uint32_t delay_remaining;
} virtualChannel;
// ================================================================================
```

precDelay and PIT_Init:

```
95  void precDelay(uint32_t virtual_channel, uint32_t us, osThreadId_t tid, uint32_t flag){
96    if (!(initialized & (1 << virtual_channel))) {
97      PIT_Init(virtual_channel, MICROS_TO_PIT(us), tid, flag);
98      initialized |= (1 << virtual_channel);
99    }
100   // add to our PIT w 4 virtual channels
101   PIT_Add(virtual_channel, MICROS_TO_PIT(us));
102   //PIT_Start();
103 }
104
105 void PIT_Init(uint32_t virtual_channel, uint32_t delay, osThreadId_t tid, uint32_t flag) {
106   // Enable clock to PIT module
107   SIM->SCGC6 |= SIM_SCGC6_PIT_MASK;
108
109   // Enable module, freeze timers in debug mode
110   PIT->MCR &= ~PIT_MCR_MDIS_MASK;
111   PIT->MCR |= PIT_MCR_FRZ_MASK;
112
113   // set return thread & flag to set for virtual channel
114   vcs[virtual_channel].thread_id = tid;
115   vcs[virtual_channel].thread_flag = flag;
116   vcs[virtual_channel].delay_remaining = delay;
117
118   // No chaining
119   PIT->CHANNEL[0].TCTRL &= PIT_TCTRL_CHN_MASK;
120
121   // Generate interrupts
122   PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TIE_MASK;
123
124   /* Enable Interrupts */
125   NVIC_SetPriority(PIT_IRQn, 128); // 0, 64, 128 or 192
126   NVIC_ClearPendingIRQ(PIT_IRQn);
127   NVIC_EnableIRQ(PIT_IRQn);
128 }
```

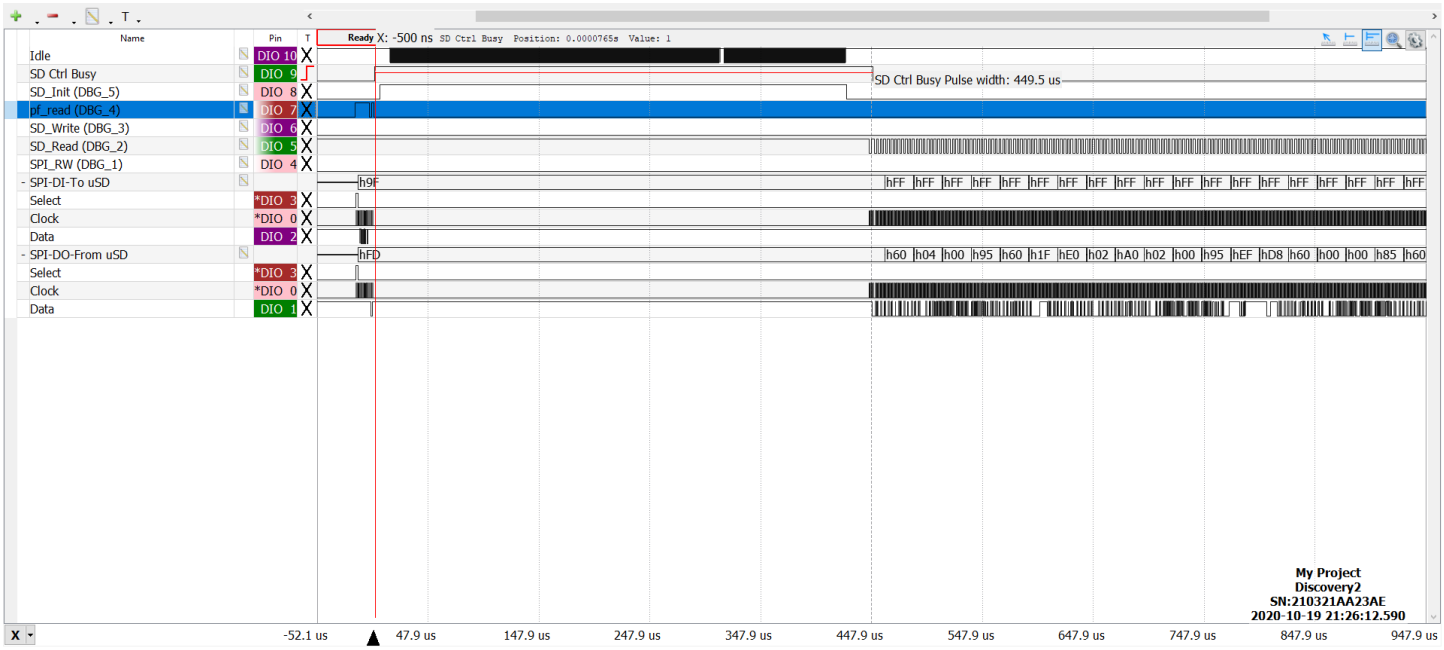
Fairly straight forward and similar to what's given, only difference is that precDelay check's if it has been initialized (so doesn't have to do it every time). PIT_Init initializes the PIT timer on the first call (of each virtual channel) as well as initializes our vcs array of virtual channels. The struct can be seen in the previous figure.

USE PRECISION DELAY FOR SD READ COMMAND

Using a delay of 420us, we can see in the figure below that a delay of 450us is made. 30us off, but recovered idle time of roughly 412us. During this run, the number switched between 450us and the max for SD reading of 460us.

I could set the delay higher, but it's a trade off of wanting more idle time versus reading faster from the SD card

- **What are the values of the statistics reported on the LCD? What is the fraction of time the idle thread executes? What do the statistics indicate has changed from the previous case?**
    - Blocks: 1159, Total Time: 11650 -> slightly lower time than last time
    - Loops ~2845000 Idle Time: 483 -> lower loops and idle time than osDelay(1) since we are using a more precising timing method than 0.3 - 1ms



I would say that despite the multiple channels implementation not being perfect below delays of 50us (as no idle time is recovered), calling a single channel of lengthier delays 50us – 1000us) is impressive and precise. Especially when considering how drastically osDelay was jumping around every time it was called.

RETROSPECTIVE

- I honestly enjoyed this project, but (and I know we are in a time crunch this semester) I wish I had more time on it. I am writing this tonight, the day it's due, and I had 2 tests and 2 projects due today. This past week has been extremely difficult, and maybe spending a more time on the algorithm and trying new approaches, I can see places to optimize. During my own development process, especially with embedded systems, its imperative that I sit down and draw/sketch/pseudo-code out my ideas before jumping in. Normally with programming it's straight forward, but with this system there is a lot of small technical things to consider, and when left unchecked, can lead you down a bad path & spaghetti code.
- The IDE has so far been very useful with debugging, the only particular 'bug' I found entering breakpoints on lines which were not going to occur. For example, if I had an if statement, and set a breakpoint in the else, then it would sometimes stop in the else (VISUALLY on that line), but when observing the disassembly, it's not actually in that block. This is kind of expected but would save time if it didn't do that.