AMBIENT SENSORS
make things that matter

Not Really, but Kind of Real Time Linux

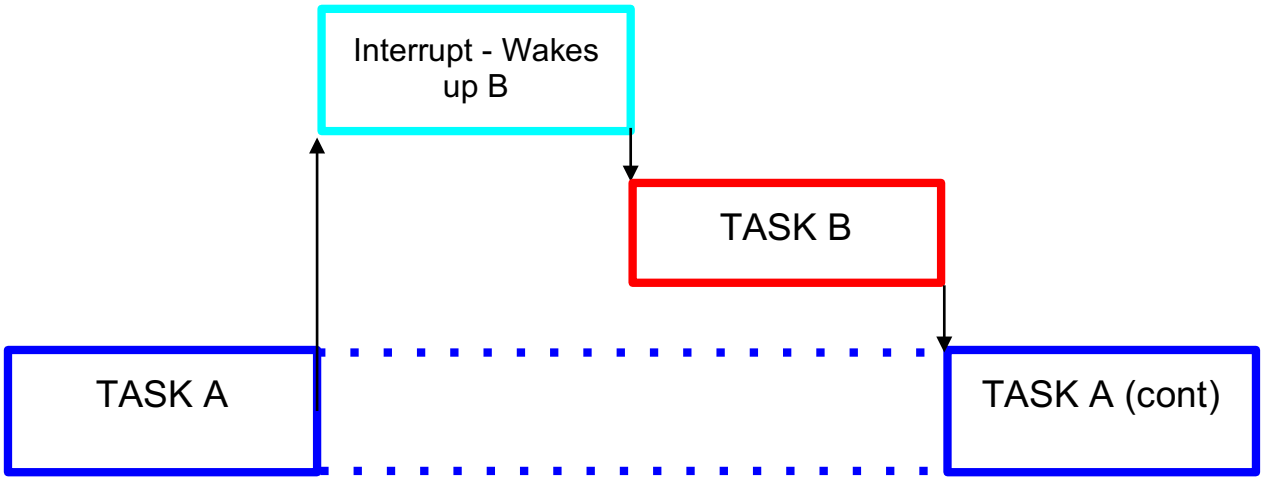
Statement of the problem:

Can a small embedded Linux system run a task deterministically enough to control external devices? More specifically, for our investigation, we wanted to know if a SBC (Single Board Computer like the Raspberry Pi or BeagleBone Black) be used to generate a pulse of 11ms or longer with a < 300usec resolution?

Problem - Unless you are using the Real Time Linux patches, Linux is not real time. If there are lots of tasks running on the system, the latency is not bounded, and can be very large.

Additional information on the problem:

The Linux scheduler decides when each task in the system gets a chance to run. With preemption - If a higher priority process wants to run, the scheduler stops the currently running process, and starts the higher priority one. Even kernel tasks are pre-emptible in modern Linux systems (if the kernel was built with CONFIG_PREEMPT - which is generally the default).



Here you see task A running, then an interrupt fires, and does something that allows task B to awaken. Task B happens to be higher priority than task A, so the scheduler runs task B. Finally, task B completes, and there are no other higher priority tasks, so the scheduler restarts task A where it left off. If Task A was your controller, that would be a problem. In general, in most

modern operating systems, you're going to be dealing with various OS overhead - interrupt latency, context switching time, memory allocation, paging, etc.

There is the concept of Real Time Linux - for example the PREEMPT-RT patches will remove unbounded latencies.

For this paper, we do not consider the patches, but for those interested, Andreas Ehmanns gave a great talk at the 2017 Embedded Linux Conference. In addition to discussing Real Time Linux, he had an excellent section on testing and tuning Linux which can be useful for anyone trying to find bottlenecks in any Linux system, whether you're doing Real Time or not:
https://openiotelcna2017.sched.com/event/9IuD/real-time-linux-on-embedded-multicore-processors-andreas-ehmanns-technical-advisor
https://www.youtube.com/watch?v=Q8vCi3ns0bs

Why not just use Real Time Linux and be done with it?

There is work involved with supporting a Linux system with the PREEMPT_RT patches. You don't just apply the patches, and you're done. Significant testing and tuning is required, and if the project has generated new drivers or kernel modules, modification of these may be necessary. Additionally, there may be lag in support for new kernel versions, and a patched kernel may be more sensitive to certain kinds of bugs than an unpatched kernel.

For those interested in the actual PREEMPT-RT patch for Linux, look up Andreas Ehmanns great talk from ELC 2017 at https://www.youtube.com/watch?v=Q8vCi3ns0bs. In addition to discussing Real Time Linux, he had an excellent section on testing and tuning Linux. This is useful for anyone trying to find bottlenecks in any Linux system, whether you're doing Real time or not.
Also, at ELC 2018, Julia Cartwright gave a talk about driver development on RT - "What Every Driver Developer Should Know about RT". This gives some really good information for driver writers. https://www.youtube.com/watch?v=-J0y_usjYxo

Generating Pulses

There are various ways to try to control a thread well enough to generate a pulse. Here are some of the methods we considered:

- Assert a GPIO, then do an 11ms delay in the kernel (sleep or busy-wait).
- The old fashioned Linux timers (timer wheel) works in jiffies, but is not very high resolution (multiple millisecond resolution), so doesn't meet the < 1ms resolution requirement, and is replaced by…
- Linux high resolution timers are specified in nanoseconds (but SBCs don't support ns granularity - 10-100usec is more normal). The task will run at the time you specified or later (no guarantees as to how much later it could run).
- On a multi-core system, dedicate a core to device control (asserting and de-asserting the GPIO), either keeping the core part of the Linux system, or a bare metal process.

(disadvantages to bare metal - lose the benefits of Linux - system features available, and communication with other processes on the other cores). One way of doing this and keeping the core as part of Linux is to reserve the core with Linux functionality.

- Keep the code in the L1 cache - avoid latency due to cache misses
- Setting the GPIO task to a high priority compared to the rest of the system can mitigate latency issues, but there are no guarantees. The Linux scheduler supports two "realtime" policies: SCHED_FIFO or SCHED_RR.
- Disable kernel preemption for that core: use function get_cpu() to disable kernel preemption and return the cpu id. When done, function put_cpu() must be called. Note that this should only be in critical sections. If a critical section is large, it can create problems with rcu_sched self-detected stalls. If you need to turn off stall detection, can do with:

```
$ sudo sh -c "echo 1 >
/sys/module/rcupdate/parameters/rcu_cpu_stall_suppress"
```

- Turn off real time throttling[i]

```
$ sudo sh -c 'echo -1 > /proc/sys/kernel/sched_rt_runtime_us'
```

Fairly early on in the investigation, we discovered the book "Raspberry Pi IoT in C"[ii] that claims by locking a process to a core, and using Linux “realtime” scheduling (high priority SCHED_FIFO), that a response of 15 microseconds is reasonable. I didn't see any discussion of reserving the core, which could make things even better.

This book seems to do its work primarily in user space, not kernel space, so following along that line, our first tests were in user space.

Experiments on the Raspberry Pi

The first tests consisted of:

1. Reserving one of the 4 cores on a Raspberry Pi 3.
2. Create a GPIO writing task that outputs 10 ms pulses (or less for checking accuracy).
3. Lock the GPIO writing task to the reserved core.
4. Make the task a high priority SCHED_FIFO.
5. Lock the task into memory to keep it from paging out (mlockall(MCL_CURRENT)) - (note that this applies to user space only - not in kernel space in the next section)
6. Run a bunch of busy-work user space programs on the Raspberry Pi (they should only run on the other 3 cores).
7. Measure the output from the GPIO to see how accurate the pulses are over time. (internal measure, external measure)

This generates a series of pulses. But please note that the problem we are trying to solve is NOT that of generating a pulse train (if this is needed, then use a Pi internal clock generator plumbed to a pin). Instead, we are trying to determine how accurately a GPIO pulse can be generated. We test this by having a thread that asserts the gpio, waits the requested pulse

width time, then de-asserts the gpio, repeating this process multiple times. If it could do this reliably, even with a very busy system, we could be confident that we could control a pulse whenever needed. This was the reason to generate a continuous pulse train of the target pulse width, and see how uniform the pulses were.

User space - functionality of the program:

1. Get and store the current time
2. Write the GPIO line high.
3. Do a usleep (this is not a busy-wait loop - the Linux system sleeps the task)
4. Write the GPIO line low.
5. Get and store the current time - compare to the start time in step 1 (verify the clock hasn't rolled over, etc).
6. Store the shortest time (to verify the pulse wasn't too short), and the the longest time (to see the worst case error in pulse length).

We decided to run the first set of experiments without any “Real-Time-like” modifications - so in essence there will be no latency minimization. We won’t know how much of an improvement we made until we see how well an unmodified Linux system works - so this will work as a control.

Therefore, the first test was run without reserving one of the cores: without SCHED_FIFO and high priority, and without locking the task into memory. We expected that there would be poor accuracy of the pulse width, and we were not disappointed:

Control Results (Unmodified Linux system)

Ideal Pulse (usec)	Shortest Pulse (rounded usec)	Longest Pulse (rounded usec)
10,000	10,060	14,000 (some runs showed > 20,000 usec)
1,000	1,060	7,100 (some runs showed > 9,000 usec)
100	140	2,900 (some runs showed > 6,200 usec)
10	30	1,500 (some runs showed > 6,900 usec)

The general idea here is that latency is very unpredictable, not in a small part due to an artificially swamped Linux system. Normally you wouldn’t expect your Linux system to be this loaded down. Hundreds of these swamping tasks could be thrown onto the same CPU core as the task controlling the GPIO, so the pulse width can vary a lot.

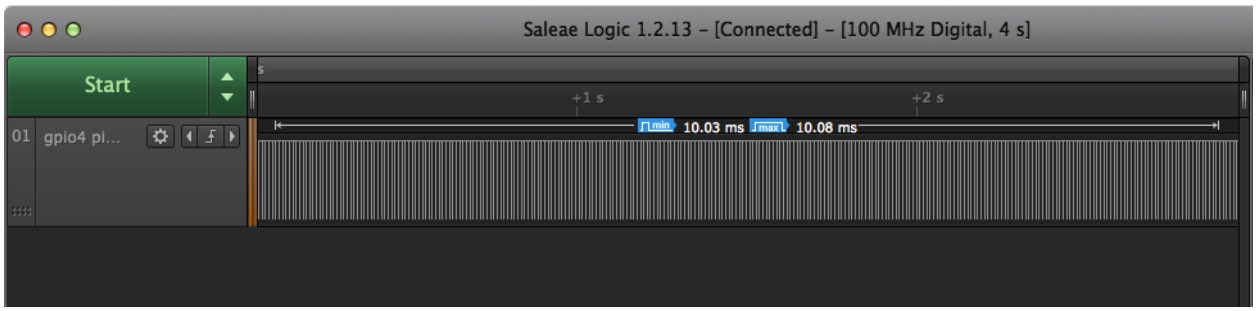
Now running the test while reserving one of the cores, locking our control task to the reserved core, setting up the scheduler to SCHED_FIFO and setting our task to high priority, and locking the task into memory, we see significantly better results:

User Space - Minimizing Latency

Ideal Pulse (usec)	Shortest Pulse (rounded usec)	Longest Pulse (rounded usec)
10,000	10,024	10,092
1,000	1,009	1,097
100	109	179
10	15	97

Obviously, these modifications are giving us more control - the pulse widths are much more bounded compared to the unmodified Linux system.

Note that the shortest pulse is never too short. This is expected - from the Linux main page usleep(3): "The usleep function suspends execution of the calling thread for (at least) usec microseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers."



We used a Salae Logic analyzer to take external measurements. You can see that during this 4 second time, the desired 10ms pulse was 30 - 80 microseconds longer than the ideal.

If the goal is to be able to control a pulse of 10ms, with an error of < 300usec or less, user space does look possible. It is disappointing to see the 100usec pulse having a > 50usec error, and the 10usec pulse having a > 80usec error. More work could be done to investigate other ways to get better resolution/less error in pulses generated form user space. But now having gathered some baseline data, we moved to our goal of running this experiment in kernel space.

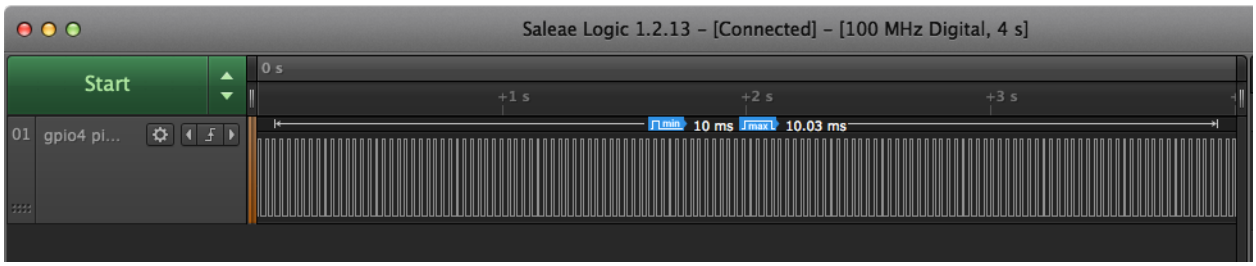
Kernel Module

- Write to gpios directly (avoiding any gpio driver code delays
- Reserve the core, and lock/pin the controlling task to that reserved core
- Set the scheduling to high priority SCHED_FIFO.
- Kernel space doesn't need to be locked in memory…
- And run 2 sets of tests - one with sleep (udelay) and one with busy-wait (usleep_range). This will help to determine which is a better fit for a given system.
- Try with and without disabling kernel preemption (turning off stall detection)
- Turn off real time throttling

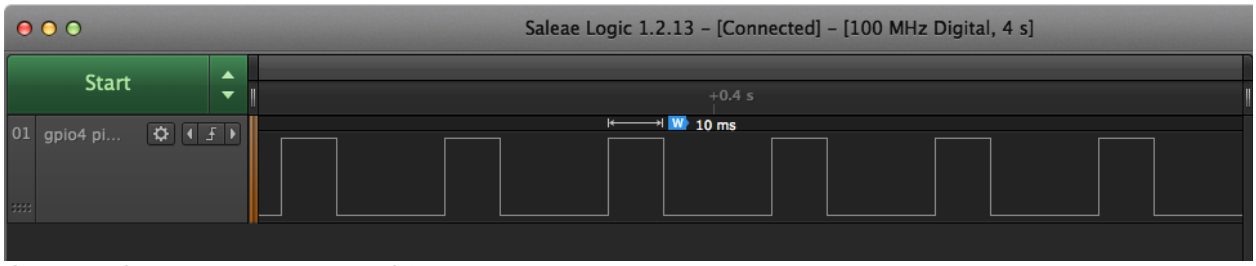
Results WITHOUT turning off kernel pre-emption
Kernel Module

Ideal Pulse (usec)	Shortest Pulse (rounded usec)	Longest Pulse	Average	Wait Mechanism
10,000	10,000	10,072	10,005	udelay (busy wait)
10,000	10,009	10,093	10,040	usleep_range (kernel sleep)
1,000	1,000	1,062	1,002	udelay (busy wait)
1,000	1,007	1,095	1,036	usleep_range (kernel sleep)
100	100	170	101	udelay (busy wait)
100	107	151	114	usleep_range (kernel_sleep)
10	10	87	11	udelay (busy wait)
10	14	74	22	usleep_range (kernel_sleep)

Normally, one is very cautious about running a busy/wait loop for a long period of time, but since we have isolated a core specifically for this thread, a busy wait loop isn't so onerous - there are no other tasks that are supposed to be running on this core. Could there be a problem with heat? Possibly - so it's important to evaluate just how long you are running the busy/wait over long periods of time, and what side effects this could have.



During this 4 second logic analyzer measurement (with the many busy work processes running), you see that the pulses range between spot on, and 30 microseconds too long.



Close up of the pulses, in a region of 10ms pulses.

Results WITH turning off kernel pre-emption using get_cpu Kernel module

Ideal Pulse (usec)	Shortest Pulse (rounded usec)	Longest Pulse (rounded usec)	Average	Wait Mechanism
10,000	10,000	10,063	10,003	udelay (busy wait)
1,000	1,000	1,060	1,002	udelay (busy wait)
100	100	152	100	udelay (busy wait)
10	10	88	11	udelay (busy wait)

And then, finally, these are the results of the separate experiment - turning off kernel preemption. It showed some affect, but it was not very significant. This makes sense, since very little is being scheduled on our reserved core anyway, so there wasn't much preempting of our GPIO controlling task.
(Note that usleep with kernel preemption turned off is not part of these results, since this combination can cause the kernel to generate messages such as:
BUG: scheduling while atomic: sh/16301/0x00000002…)

Kernel modules - do we really own our core?

How sure are we that the core belongs to our gpio control thread? Running the Linux top program can give us an idea.

```
top - 20:55:43 up 27 min,  6 users,  load average: 236.44, 536.81, 303.56
Tasks: 136 total,   3 running, 133 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us, 25.2 sy,  0.0 ni, 74.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:    947684 total,    75996 used,   871688 free,      196 buffers
KiB Swap:   102396 total,    41580 used,    60816 free.    11260 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND                P
   19 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migration/3            3
   20 root      20   0       0      0      0 S   0.0  0.0   0:00.00 ksoftirqd/3            3
   21 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kworker/3:0            3
   22 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/3:0H           3
 1066 root      20   0       0      0      0 R   0.0  0.0   0:00.00 kworker/3:1            3
16287 root      rt   0    1912      0      0 R 100.0  0.0   3:57.66 sh                     3
    1 root      20   0   23916   2448   1828 S   0.0  0.3   0:08.11 systemd                2
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.01 rcu_sched              2
   15 root      rt   0       0      0      0 S   0.0  0.0   0:00.57 migration/2            2
```

This is the experiment running on a mostly idle system: there are no busy work processes running on the other 3 cores. I've sorted the "top" output by processor core. Our controller thread is on core 3 (it's running as sh here). It is taking 100% of core 3 (this is a busy/wait, not a sleep). You'll notice that the kernel has created 5 other threads on our core, but they aren't doing much. The rest of the cores are mostly idle (74.8% - the other 3 cores).

```
top - 17:28:23 up  1:20,  6 users,  load average: 639.78, 295.22, 184.27
Tasks: 2396 total,   8 running, 2382 sleeping,   2 stopped,   4 zombie
%Cpu(s): 14.5 us, 84.4 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  1.2 si,  0.0 st
KiB Mem:    947684 total,   491496 used,   456188 free,     3160 buffers
KiB Swap:   102396 total,    64528 used,    37868 free.    17044 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND                P
   19 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migration/3           3
   20 root      20   0       0      0      0 S   0.0  0.0   0:00.00 ksoftirqd/3           3
   21 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kworker/3:0           3
   22 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/3:0H          3
10147 root      rt   0    1912    392    332 R 100.0  0.0   1:42.17 sh                    3
14279 root      20   0       0      0      0 R   0.0  0.0   0:00.00 kworker/3:1           3
    1 root      20   0   22860   2880   2312 S   0.0  0.3   0:06.21 systemd               2
    7 root      20   0       0      0      0 S   0.3  0.0   0:01.40 rcu_preempt           2
   15 root      rt   0       0      0      0 S   0.3  0.0   0:00.39 migration/2           2
```

Here is the system with the busy work processes running on cores 0-2, and our control thread running on core 3. There is no idle time in the system - it's really loaded down. You see 14% of the time spent in user space (the busy work processes are shell scripts), but a whopping 84.4% of the time spent in kernel process time, and 1.2% system interrupts. But even with the system totally busy, we still had those decent results.

Experiments on the BeagleBone Black

The BeagleBone Black has a single Cortex A8, but additionally has 2 PRU cores. These Programmable Realtime Units are 32 bit processors running at 200 MHz with single-cycle I/O access. They each have 8K program RAM, and 8K data RAM. Additionally, there is 12K shared data RAM between them.

There is no need on the BeagleBone Black to reserve a core and run our GPIO control thread on that core. Instead, let Linux run as it normally would, and use the Programmable Realtime Units (PRUs) to do the GPIO control. You don't run Linux on these PRUs, you write your own thread code. There is no OS latency on these, since there is no OS. The only delays you have are those you create in your thread (intentional or unintentional). To compute a delay for the pulse, count the lines of code, add their execution time to the delays you add to create your pulse, and that's how wide your pulse will be.

There are 25 low-latency I/Os for the PRUs. At 200MHz, a PRU gives deterministic sub-microsecond GPIO control (down to 5ns!). There is no pipeline, so you actually have real-time response. For more information on the PRUs, see Endnote[iii]

For this experiment, the entirety of the device control (the creation of a pulse on the GPIO) can be done in the PRU code. For more extensive device control, the code may be too large to fit into the PRUs. These PRUs are very accurate, but instruction space is limited. You won't be implementing massive amounts of code on them. The user will need to figure out how to divide up the logic between Linux on the A8, and the PRUs.

External communication (Lan, web, ssh, etc), pre and post processing of the data, etc., will normally happen on the A8, and then when the Real time work is needed, the A8 will signal the PRUs. For my experiments, I actually was able to implement everything on the PRUs.

But even in our case, where the device control is completely in the PRUs, Linux is needed to load the code into the PRUs.

There are 2 commonly used drivers to communicate from Linux to the PRUs:

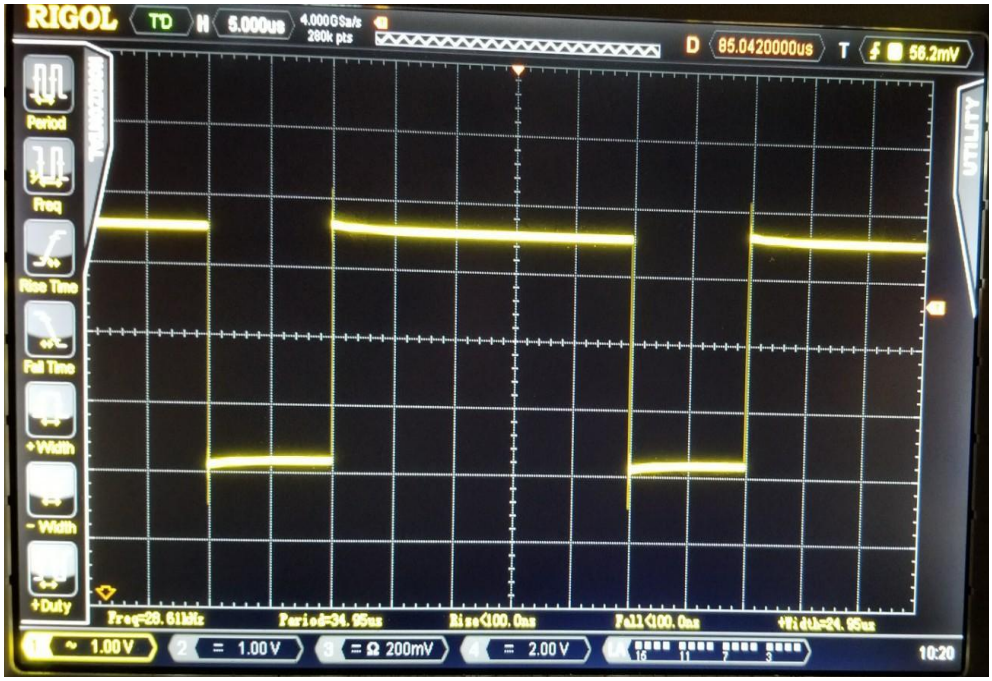
1. uio_pruss - memory maps all the PRU registers to user space - this can be a bit dangerous, but some find easier to use from user space.
2. pru_rproc - a proper Linux driver using Linux remote proc - this is normally preferred.

The general steps to get code working on the PRUs (These steps are courtesy of Jason Kridner's presentation[iv]):

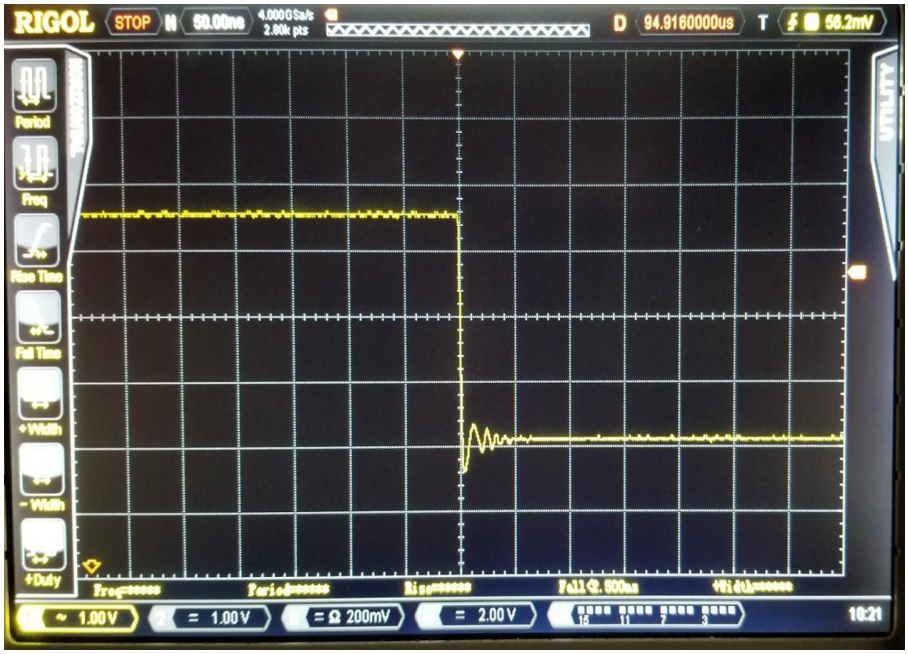
1. Write and compile the firmware you want running on the PRU.
2. Use the A8 (running Debian Linux) to load the firmware onto the PRU (I used the pru_rproc driver).
3. Create Device Tree entries to configure the kernel and the I/O pins.
4. Implement a communication mechanism between A8 and PRUs (as needed - that was not part of this experiment).

Results on the BeagleBone Black

The Salae logic analyzer couldn't measure any error (we hit the Nyquist sampling limit). Moving over to our high speed oscilloscope it showed that the error was on the order of 10s of nanoseconds (and we need to do more evaluation to see if some of that was due miscounting of instructions in the delay loop). In an effort to see more data, and to play with more accurate pulses, a second experiment was done with a 25 usec pulse.



The experiment output pulses that ranged from 25 usec to 24.95 usec. In this logic analyzer trace the pulse width is 24.95 usec.



This logic analyzer trace is zooming way in on the trailing edge of the signal - it looks pretty clean, especially considering it's just a standard probe connection to a breakout pin, no special connector was used.

Busy Work Processes on the BeagleBone Black

How do the Busy Work processes affect the BeagleBone Black? I ran the busy work processes a couple of times, taking measurements, expecting that they would have no impact on the PRU, and this assumption was correct. There was no observed difference. This makes sense since the busy work processes only run on Linux, which is on the A8, and the PRUs don't even notice the work being done there. So while there shouldn't be any impact on the PRUs from a very busy A8, communication between the PRUs and the A8 could be delayed in that case.

Appendices:

Please NOTE: for the latest version of all the scripts and source code (and this document), please go to the Ambient Sensors website:
https://www.ambientsensors.com

And look for the Documentation (currently under "Game Changers")

All Raspberry Pi examples were run on kernel version 4.9.73-v7+

All BeagleBone Black Examples were run on kernel version 4.4.9-ti-r25

Appendix 1: Reserving a Core

Reserving a core

For this investigation, isolcpus was used to reserve (isolate) a core. Details of the syntax of this will be given below. But first, there is one important piece of information. As of October 31, 2017, isolcpus was marked as deprecated. It still exists in the Linux kernel for now, but it is unknown how much longer it will be. Future investigation: use cpusets instead. This is to allow dynamic load balancing through the "cpuset.sched_load_balance" file, and as such is a bit of a different mechanism. Here is some information about cpusets for anyone that is interested:

- https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt
- Starting with "Isolating the Application" this website discusses how to use cpusets: http://linuxrealtime.org/index.php/Improving_the_Real-Time_Properties

How to use isolcpus to reserve a core of a multi-core CPU:

1. Isolate a core - modify file /boot/cmdline.txt to add the isolcpus directive to the line. For example, to isolate core 3, add this to the line:
   ```
   isolcpus=3
   ```
   This will leave cores 0, 1, and 2 in the normal Linux pool, but prevent the scheduler from putting any normal tasks on core 3.
2. Verify the isolcpus directive worked by doing:
   ```
   $ cat /sys/devices/system/cpu/isolated
   ```
   (after rebooting to make the isolcpus line take affect). The number "3" should be returned.
3. Specify a user space program to run on the isolated core - first calculate the mask that specifies that core. With the cores numbered 0-3, core 0's mask is 0x1, core 1's mask is 0x2, core 2's mask is 0x4, and core3's mask is 0x8. So to run the program called “programname” on core 3:
   ```
   $ taskset 0x8 programname
   ```
4. See which core a certain process is pinned to (if any):
   ```
   $ taskset -cp <pid>
   ```
   Where <pid> is the process id - for example, if user_pulse is process 943:
   ```
   $ taskset -cp 943
   pid 943's current affinity list: 3
   ```
5. To see which core a certain process is actually running on:
   ```
   $ ps -o pid,psr,comm -p <pid>
   ```
   Where <pid> is the process id - for example, if user_pulse is process 943:
   ```
   $ ps -o pid,psr,comm -p 943
   PID PSR COMMAND
   943   3 user_pulse
   ```
   The PSR column shows that user_pulse is running on CPU core 3. This should not change for this process, since, as we saw in the taskset command, this process is pinned to CPU core 3. But in general, unpinned processes can be scheduled onto any non-isolated core. This is a nice verification that the user space program really is running on the correct CPU core.
6. To verify that there are no other processes running on our isolated core, there is a slight variation on the previous command:
   ```
   $ ps -eo pid,psr,comm
   ```

Which shows a bunch of output, including a few processes on CPU core 3! There will be a few kworker threads, as well as ksoftirqd, and migration. You can also use the top command to see which CPU core is being used for processes, after running "top", hit 'f', then use the down arrow to highlight "P = Last Used Cpu (SMP)", then hit 'd' to display the field, and 's' to sort on it (if you wish), then 'q' to return to the display.

7. Note that for a kernel module, the way to lock/pin a task to an isolated core is to set the cpu affinity with function set_cpus_allowed_ptr().

Appendix 2: More Information on Scheduling

More Information on Scheduling

SCHED_FIFO information from man page for sched(7) (4.14):

> The scheduler is the kernel component that decides which runnable thread will be executed by the CPU next. Each thread has an associated scheduling policy and a static scheduling priority, sched_priority. The scheduler makes its decisions based on knowledge of the scheduling policy and static priority of all threads on the system.
>
> For threads scheduled under one of the normal scheduling policies (SCHED_OTHER, SCHED_IDLE, SCHED_BATCH), sched_priority is not used in scheduling decisions (it must be specified as 0).
>
> Processes scheduled under one of the real-time policies (SCHED_FIFO, SCHED_RR) have a sched_priority value in the range 1 (low) to 99 (high). (As the numbers imply, real-time threads always have higher priority than normal threads.) ...
>
> ... when a SCHED_FIFO threads becomes runnable, it will always immediately preempt any currently running SCHED_OTHER, SCHED_BATCH, or SCHED_IDLE thread. SCHED_FIFO is a simple scheduling algorithm without time slicing.
>
> ...
>
> A SCHED_FIFO thread runs until either it is blocked by an I/O request, it is preempted by a higher priority thread, or it calls sched_yield(2).
>
> SCHED_RR is a simple enhancement of SCHED_FIFO. Everything described above for SCHED_FIFO also applies to SCHED_RR, except that each thread is allowed to run only for a maximum time quantum. If a SCHED_RR thread has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority.

Various system calls for scheduling

setpriority(2)

Set the nice value of a thread, a process group, or the set of

threads owned by a specified user.

sched_setscheduler(2)
: Set the scheduling policy and parameters of a specified thread.

sched_setparam(2)
: Set the scheduling parameters of a specified thread.

sched_yield(2)
: Cause the caller to relinquish the CPU, so that some other thread be executed.

sched_setaffinity(2)
: (Linux-specific) Set the CPU affinity of a specified thread.

sched_setattr(2)
: Set the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of sched_setscheduler(2) and sched_setparam(2).

Appendix 3: Various Notes on the BeagleBone Black

Various Notes on the BeagleBone Black

Getting started:
http://beagleboard.org/support/bone101

There are startup instructions (opening start.htm on the beagle bone mass storage device).

Connect by usb to your computer (or if usb networking drivers aren't working, use an ethernet cable to connect to your network):

```
$ ssh debian@beaglebone.local
```

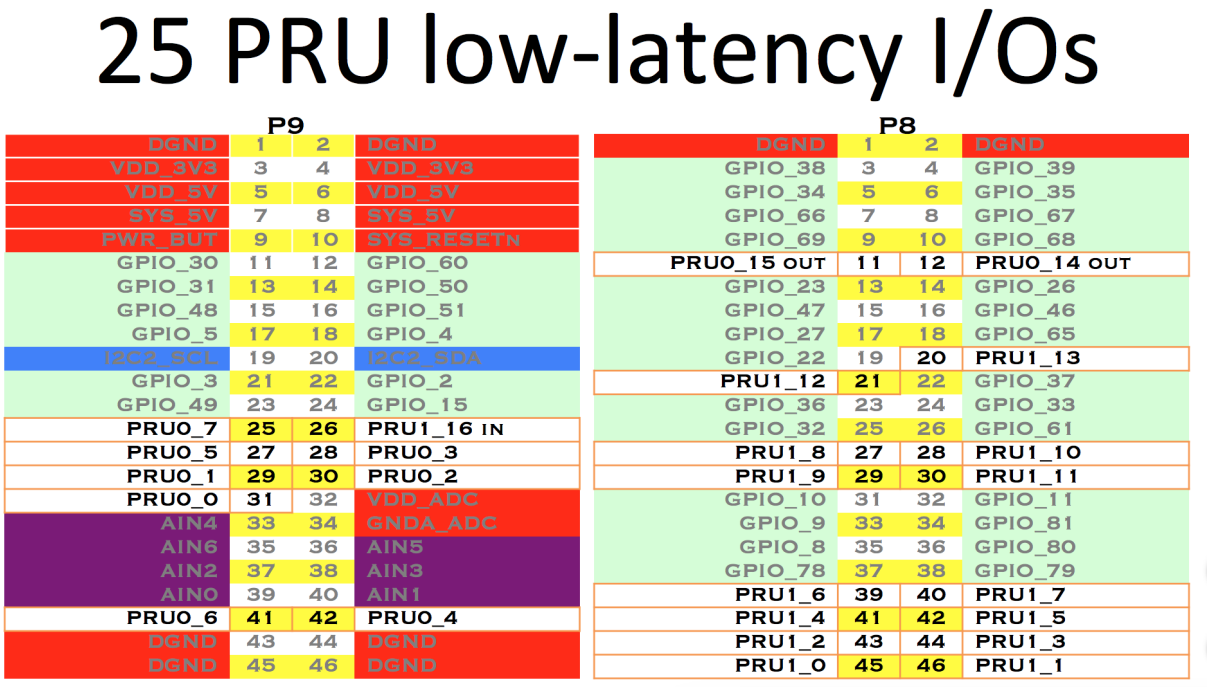
default password: temppwd

(if connected on USB, the IP address should be 192.168.7.2 or 162.168.6.2)

Some helpful information about the BeagleBone Black and kernel drivers can be found at:
http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

Intro to the BB PRU
http://beaglebone.org/pru

The PRUs have 25 low latency GPIOs, as seen in this slide:



(this slide is from https://beagleboard.org/static/PumpingStationOne20140628_Real-timeProgrammingWithBeagleBonePRUs.pptx.pdf).

When writing to the GPIOs in the PRU, write to register R30. To read from the pins, read from R31. See the code (PRU_gpiTtoggle.c) for an example; it writes to pins P8_12 and P8_11.

To figure out which pins map to which GPIOs in which modes, refer to the BeagleBone Black system reference manual at:
https://github.com/CircuitCo/BeagleBone-Black/blob/rev_b/BBB_SRM.pdf?raw=true

Page 83 (of the Rev B manual) shows the expansion header P8 pinout. There you see that gpio1[13] is on pin 11 of header P8. And that setting that GPIO to Mode6 will set up pru0 to write to this pin when bit 14 is written in R30 (pr1_pru0_pru_r30_14).

The PRU Getting started guide:
http://processors.wiki.ti.com/index.php/PRU-ICSS_Getting_Started_Guide

Developing on the BeagleBone Black

There is an IDE built into the BeagleBone Black. To use it, browse to:
http://192.168.7.2:3000/ide.html
(assuming ip address of 192.168.7.2)

Compilers

To get the PRU Cross compiler installed, follow the directions under "Setting up the PRU code generation tools" at:
https://www.zeekhuge.me/post/ptp_blinky/

After building the PRU code, copy it to /lib/firmware/am335x-pru0-fw, and reboot the PRU cores with (if you are using the remote proc driver that is done with: rmmod -f pru_rproc, and insmod your driver (e.g. /lib/modules/4.4.9-ti-r25/kernel/drivers/remoteproc/pru_rproc.ko).

There is a device tree blob overlay that is needed to use the GPIOs, the command:

```
$ echo bspm_P8_16_26 > /sys/devices/platform/bone_capemgr/slots
```

But in the version of distro I was using (kernel version 4.4.9-ti-r25) there is a default overlay already present, and it gets in the way. I removed it by editing file /boot/uBoot/uEnv.txt, and commenting out the one line with the overlay. I rebooted the BeagleBone Black, then was able to do the echo from the previous paragraph.

Appendix 4: Running the User Space Program and the Busy Scripts on the Raspberry Pi

How to run the user space program and the busy scripts on the Raspberry Pi:

- Create 3 terminal windows (either ssh into the Raspberry Pi, or in the Raspberry Pi GUI).
- To run the busy scripts, in terminal 2:
  ```
  $ (cd utils; ./runbusy.sh)
  ```
- To stop the busy scripts, do the following in terminal 1 (but the scripts will stop in 100 seconds automatically anyway):
  ```
  $ touch ~/.stoprunning
  ```
- To run the userspace program on with a 10ms pulse train, in terminal 3:
  ```
  $ sudo taskset 0x8 ./user_pulse 10000
  ```
  (the value you provide is in microseconds, so scale accordingly)

And normally, I would be running top in terminal 1, to see if anything was on CPU core 3. Note that this also adds to the execution load on the system.

Appendix 5: Running the Kernel Module and the Busy Scripts on the Raspberry Pi

How to run the kernel module and the busy scripts on the Raspberry Pi:

- If you wish to run with kernel preemption disabled, modify gpio_pulse.c - uncomment the #define DISABLE_KERNEL_PREEMPTION 1 line.
- Create 4 terminal windows (either ssh into the RPi, or in the RPi GUI)
- To run the busy scripts, in terminal 2:
  `$ cd utils; ./runbusy.sh`
- To stop the busy scripts, do the following in terminal 1 (but the scripts will stop in 100 seconds automatically anyway):
  `$ touch ~/.stoprunning`
- To run the kernel module, first you have to install the kernel module (KLM), in terminal
  `$ sudo insmod driver_gpio_pulse/gpio_pulse.ko` (loads the KLM into the kernel)
  `$ sudo chmod 666 /sys/gpio_control/*` (gives the pi user permission to control the KLM)
- And run the kernel setup script to suppress cpu stalls and turn off run time throttling
  `$ utils/kernel_setup.sh`
- And then run the kernel module in terminal 3:
  `$ utils/kernel_test.sh` (runs the KLM)
- To see the results from the run:
  `$ cat /sys/gpio_control/pulse_run`
- To see which gpio pin is being driven:
  `$ cat /sys/gpio_control/gpio_pin`
  Note, this is the gpio number, not the pin number on connector J8. To see how these relate to each other, look at the Raspberry Pi schematic "GPIO EXPANSION" and "J8". You will see that GPIO4 is on J8 pin 7 (the default for this example).
- To see the output from kernel module, type "dmesg" in an terminal 4. To continually watch the output, type "tail -f /var/log/kern.log"
- To modify the defined run parameters (which GPIO pin to drive, and the width of the pulse):
  `$ echo 5 > /sys/gpio_control/gpio_pin` (this changes the pin to drive to GPIO5 (J8 pin 29)
  `$ echo 1000 > /sys/gpio_control/pulse_width` (this changes the pulse width to 1000usec)
- If you have modified the kernel module and need re-install it, you have to uninstall the module, before re-installing it:
  `$ sudo rmmod gpio_pulse.ko`

And normally, I would be running top in terminal 1, to see if anything was on CPU core 3. Note that this also adds to the execution load on the system.

There are lots of pinouts online, the official schematics are here:
https://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/Raspberry-Pi-3B-V1.2-Schematics.pdf

The code for user space and kernel space uses pin 7 on connecter J8 (GPIO4).

Appendix 6: Setting up Your Raspberry Pi in Preparation for Running the Programs/Scripts Described in this Whitepaper

How to setup your Raspberry Pi in preparation for running the programs/scripts described in this whitepaper:

Following the instructions on
https://www.raspberrypi.org/documentation/linux/kernel/building.md

(The instructions listed here are for compiling on the Raspberry Pi directly. There are additional instructions on the referenced raspberrypi.org website that describe setting up the cross-compile environment. I did not use these instructions, since after the first big build and install is complete on the Pi, building of the user space program and kernel loadable module were very quick. That first build of the kernel on the Pi is a long process, though...)

- Set up the Raspberry Pi following their instructions to install the latest version of Raspbian, and log into the Pi.
- Install Git and the build dependencies:
  ```
  $ sudo apt-get install git bc
  ```
- Get the sources:
  ```
  $ git clone --depth=1 https://github.com/raspberrypi/linux
  ```
- Configure the kernel (for a Raspberry Pi 3):
  ```
  $ cd linux
  $ KERNEL=kernel7
  $ make bcm2709_defconfig
  ```
- Build and install the kernel, modules, and dtbs - get some coffee/tea....
  ```
  $ make -j4 zImage modules dtbs
  ```
  (very long step)
  ```
  $ sudo make modules_install
  $ sudo cp arch/arm/boot/dts/*.dtb /boot/
  $ sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
  $ sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
  $ sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
  ```
- Reboot to this newly compiled kernel
  (Be careful - watch for error messages, and don't run the next command if the previous one failed)
  At this point, you should be able to build the kernel loadable module (driver_gpio_pulse/gpio_pulse.c) as shown in Appendix C??

To run the user program (user_gpio_pulse/user_pulse.c):

- Install the bcm2835 library for accessing GPIOs on the Pi:
  ```
  $ wget http://www.airspayce.com/mikem/bcm2835/bcm2835-1.50.tar.gz
  $ tar xvfz bcm2835-1.50.tar.gz
  ```

```
$ cd bcm2835-1.50
$ ./configure
$ make
$ sudo make install
```

ENDNOTES

---

[i] Quick explanation of real-time throttling
"As long as there are real-time tasks, i.e. tasks scheduled as SCHED_FIFO or SCHED_RR, that are ready to run, they would consume all CPU power if the scheduling principles were followed. Sometimes that is the wanted behavior, but it would also allow that bugs in real-time threads completely block the system. To prevent this from happening, there is a real-time throttling mechanism which makes it possible to limit the amount of CPU power that the real-time threads can consume."
http://linuxrealtime.org/index.php/Basic_Linux_from_a_Real-Time_Perspective#Real-Time_Throttling

[ii] http://www.iot-programmer.com/index.php?option=com_content&view=article&id=33&catid=22&Itemid=110

[iii] For more information on the PRUs - There is a 2015 ELC presentation by Rob Birkett called "Enhancing Real-Time Capabilities with the PRU"
https://www.youtube.com/watch?v=plCYsbmMbmY
Additionally, please see the website - http://processors.wiki.ti.com/index.php/PRU-ICSS

[iv] Jason Kridner's video "Using the BeagleBone Real-time Microcontrollers" on this web site:
http://beaglebone.org/pru
At 17:30 is where he starts talking about the BBB, and 20:20 he starts talking about PRUs.