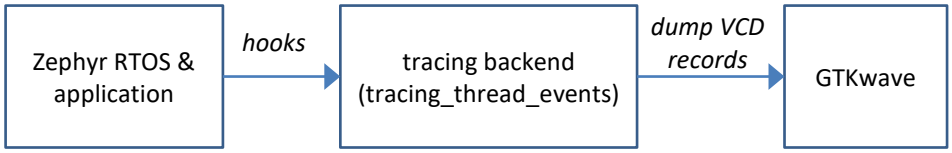
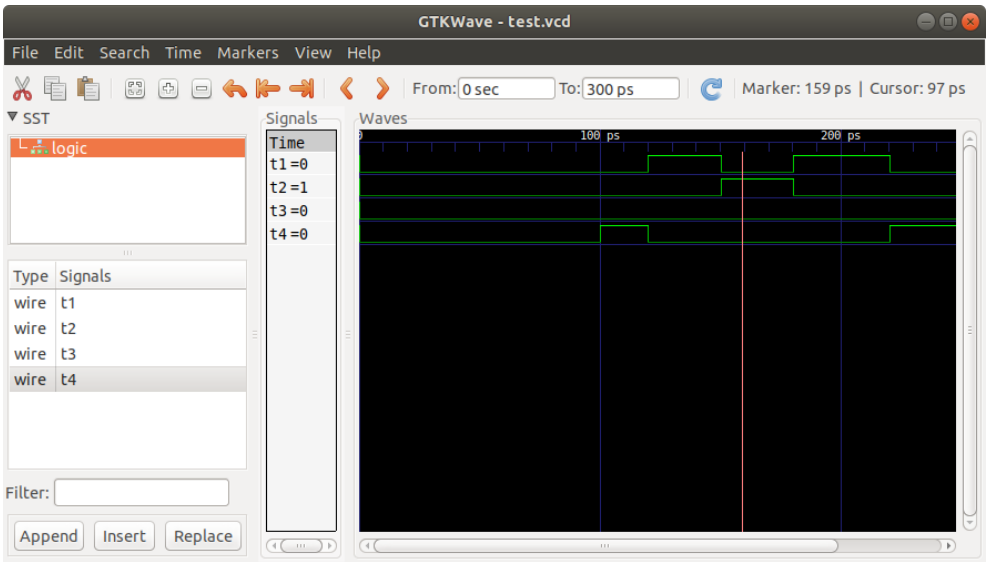
Assignment 3 Thread Event Tracing in Zephyr RTOS (100 points)

Zephyr RTOS is equipped with trace hooks for thread execution events, such as thread switch in and out. A backend component can be employed to collect these trace events and pass them to a visualization program running in a host machine. In Zephyr v14.2, there are 3 example backend components that have their interfaces defined in *tracing_cpu_stats.h*, *tracing_ctf.h*, and *tracing_sysview.h*.

In this assignment, you are asked to develop a new tracing backend with its interface defined in *tracing_thread_events.h*. Tracing data should be recorded in VCD (Variable Change Dump https://en.wikipedia.org/wiki/Value_change_dump) format and can be dumped to a host machine. We can then use the visualization tool GTKwave (http://gtkwave.sourceforge.net/) to analyze the execution of multi-threaded programs.





The events to be collected in *tracing_thread_events* should include:

1. thread_swapp_in and _out (to be represented as rising and falling edges in VCD)
2. thread_create (to be represented as a pulse in VCD)
3. thread_ready (to be represented as a pulse in VCD)
4. thread_pend (to be represented as a pulse in VCD)
5. mutex_lock and _unlock (to be represented as a pulse in VCD)

Configuration parameter CONFIG_TRACING_THREAD_EVENTS should be used to enable *tracing_thread_events* backend. The other two configuration parameters should also enabled:

1. CONFIG_THREAD_MONITOR – to enable a list of all threads
2. CONFIG_THREAD_NAME – to have thread name in a character array.

For dumping VCD records, you may consider a simple approach that

1. Save event records in an internal buffer. The buffer should have a fixed size and no additional events will be saved when the buffer is full.
2. Have a tracing_dump call to print the event records on console. The tracing_dump(), along with tracing_start() and tracing_end() can be implemented in the backend and invoked via tracing hook *sys_trace_void()* and *sys_trace_end_call().*
3. Use PuTTY session logging to write the records into a file at the host machine.
4. Have a bash or python script to clean up the log file into a vcd file for GTKwave.

To test your *tracing_thread_events* backend, you need to develop an application program, named *trace_app,* that creates multiple threads for periodic tasks. The program should terminate the execution of all threads after a fixed amount of time and then invoke *tracing_dump* to print the event records. As shown in the following diagram, periodic tasks in real-time embedded systems can be simply expressed as endless loops with time-based triggers. In the task body, specific computation should be done and locks must be acquired when entering any critical sections.

```
TASK periodic_task()
{
        < local variables >
        initialization() and wait_for_activation();
        while (condition) {
                <task body>
                wait_for_period();
        }
}
```

(from "*Ptask: an Educational C Library for Programming Real-Time Systems on Linux*" By Giorgio Buttazzo and Giuseppe Lipari, *EFTA 2013*)

To simulate real work, the task body is defined as the following sequence:

<compute_1> <lock_*m*> <compute_2> <unlock_*m* > <compute_3>

where "lock_*m*" and "unlock_*m*" are locking and unlocking operations on mutex *m*, and "compute_n" indicates a local computation. To simulate a local computation, we will use a busy loop of *n* iterations in the assignment, i.e.

```
volatile uint64_t n;

while(n > 0) n--;
```

The input to your program is a specification of a task set. The task parameters of each task is defined in *struct Tasks*:

```
struct Tasks
{
        char t_name[32];        // task name
        int priority;           // priority of the task
        int period;             // period for periodic task in miliseconds
        int loop_iter[3];       // loop iterations for compute_1, compute_2 and compute_3
        int mutex_m;            // the mutex id to be locked and unlocked by the task
};
```

For instance, in the given *task_model.h*, a task set of 6 tasks are defined:

```
#define THREAD0 {"task 0", 2, 10, {100000, 100000, 100000}, 2}
#define THREAD1 {"task 1", 3, 28, {100000, 100000, 100000}, 0}
#define THREAD2 {"task 2", 4, 64, {100000, 100000, 100000}, 2}
#define THREAD3 {"task 3", 5, 84, {100000, 100000, 100000}, 1}
#define THREAD4 {"task 4", 6, 125, {100000, 100000, 100000}, 0}
#define THREAD5 {"task 5", 7, 165, {100000, 100000, 100000}, 2}

struct Tasks threads[NUM_THREADS]={THREAD0, THREAD1, THREAD2, THREAD3, THREAD4,
        THREAD5};
```

Due Date

The due date is 11:59pm, March 31.

What to Turn in for Grading

- Your submission is a zip archive, named RTES-LastName-FirstInitial_03.zip, that includes
  - A pdf report to include a brief description of your implementation and the GTKwave screenshots of two runs of *trace*_app with CONFIG_PRIORITY_CEILING=0 and 10, respectively. The input task set of the two runs must be stated in the report. The screenshots should be chosen to illustrate the effect of priority inheritance. Don't forget to add your name and ASU id in the report.
  - A patch file to include all your changes in the Zephyr 1.14.2 commit for the implementation of *tracing_thread_event* and a sample application *trace_app*. The script to clean up session log file should also be included in *trace_app.* The patch file should be named as *assignment3.patch* and should be created from the root directory of Zephyr source tree.
  - A readme text file with all the commands you use. Alternatively, you could have a script file with comments inside on how to use it.
- Note that any object code or temporary files should not be included in the submission. Submit the zip archive to the course Canvas by the due date and time.
- Please make sure that you comment the source files properly and the readme file includes a description about how to make and use your software. Don't forget to add your name and ASU id in the readme file.
- There will be 20 points penalty per day if the submission is late. Note that submissions are time stamped by Canvas. If you have multiple submissions, only the newest one will be graded. If needed, you can send an email to the instructor to drop a submission.
- The assignment must be done individually. No collaboration is allowed, except the open discussion in the forum on Canvas. The instructor reserves the right to ask any student to explain the work and adjust the grade accordingly.
- Failure to follow these instructions may cause deduction of points.
- Here are few general rule for deductions:
  - Cannot apply patch or compilation error -- 0 point for the assignment.
  - Must have "–Wall" flag for compilation -- 5-point deduction for each warning.
  - 10-point deduction if no compilation or execution instruction in README file.
  - Source programs are not commented properly -- 10-20-point deduction.
- ASU Academic Integrity Policy (http://provost.asu.edu/academicintegrity), and FSE Honor Code (http://engineering.asu.edu/integrity) are strictly enforced and followed.

How your program will be graded:

1. Unzip your submission and apply your patch to Zephyr source code

2. cd */Zephyr/samples/trace_app*, mkdir build, cd build, cmake, and make
3. Enable Putty session logging
4. Run your application image on Galileo Gen 2 board
5. Examine your source code and report, and view event trace via GTKwave on host.