CSE 522 - Real-time Embedded Systems - Spring 2021

Report - Thread event tracing in Zephyr RTOS

Shyam Joshi - 1218594676
sjoshi46@asu.edu

Implementation

In thread_events.c, the following logic is used to record thread events:

```
k = irq_lock();                     /* Disable interrupts for the
processor that is running the thread */
check = trace;                      /* Copy the tracing flag to a local
variable */
irq_unlock(k);                      /* Enable back the interrupts for the
processor */

if(check)
{
if(tEvents_index > fix_size) {return;}
      k_tid_t id = k_current_get();
      const char *name = k_thread_name_get(id);
      threadEvents[tEvents_index].event='i';
      strcpy(threadEvents[tEvents_index].thread_name,name);
      threadEvents[tEvents_index].TSC = (temp/400);
      tEvents_index++;
}
```

1. Interrupts are disabled for the local core which is executing the thread. They are disabled because they provide atomic access. Mutex cannot be used because, whenever mutex_lock/mutex_unlock is invoked, sys_trace_void() is called and if we use mutex inside them, then it will create an infinite recursion.
2. trace flag is copied into a local variable.
3. Interrupts are enabled back.
4. tEvent_index is the variable for accessing the custom structure array which holds the trace data. It is checked if it has reached till the maximum size allowed, if yes then return.
5. Thread name is acquired using thread id of the executing thread.
6. Using tEvent_index, time stamp, thread name and event type are filled in the custom structure and tEvent_index is incremented to access the next empty custom structure array.

Custom structure is defined as follows:

```
struct threadInfo         // custom structure for recording time stamps
{
    char event;                          // name of the event {i:
task_switched_in, o: task_switched_out, e: other tasks}
    char thread_name[26];                // name of the thread
    unsigned int TSC;                      // time stamp variable
};
```

There are three different symbols used to indicate the type of the event and they are:

1. ‘i’ : For the event when sys_trace_switched_in() is invoked.
2. ‘o’ : For the event when sys_trace_switched_out() is invoked.
3. ‘e’ : For the events such as sys_trace_thread_create(), sys_trace_thread_ready(), sys_trace_thread_pending(), mutex_lock(), mutex_unlock().

mutex_lock() and mutex_unlock() is implemented in sys_trace_void() with the same above logic. The only difference is that a condition is placed to validate the argument with the id of event mutex_lock and mutex_unlock.

trace flag is set and unset in sys_trace_void() and sys_trace_end_call() by validating the argument of the function. They are called inside tracing_start() and tracing_end() respectively, defined in main.c by passing appropriate argument to start and end tracing.

In main.c, the following tasks are performed:

1. NUM_MUTEXES number of mutexes are defined.
2. tracing_start is invoked to start recording trace data.
3. NUM_THREADS number of threads are created and thread’s structure is passed as an argument for it’s callback function.
4. Each of the threads are assigned the name defined in task_model.h

Same callback function is assigned to each of the threads and the following task is performed:

1. Number of iterations to be performed in each computation are stored locally.
2. A time stamp is collected before starting a busy loop.
3. A condition is checked to see if TOTAL_TIME amount of time is expired or not. If yes, then the thread returns.
4. Else, it performs compute_1, acquires lock, performs compute_2, releases lock and performs compute_3.
5. A new time stamp is collected and in the condition before entering the busy loop, it is subtracted from the time stamp collected before entering the busy loop and the result is compared with TOTAL_TIME.
6. When the last thread exits, tracing_end() and tracing_dump() are invoked.

The following are the two screenshots from GTKWave with CONFIG_PRIORITY_CEILING=0 and CONFIG_PRIORITY_CEILING=10 with the same input task set in the two runs as defined in task_model.h

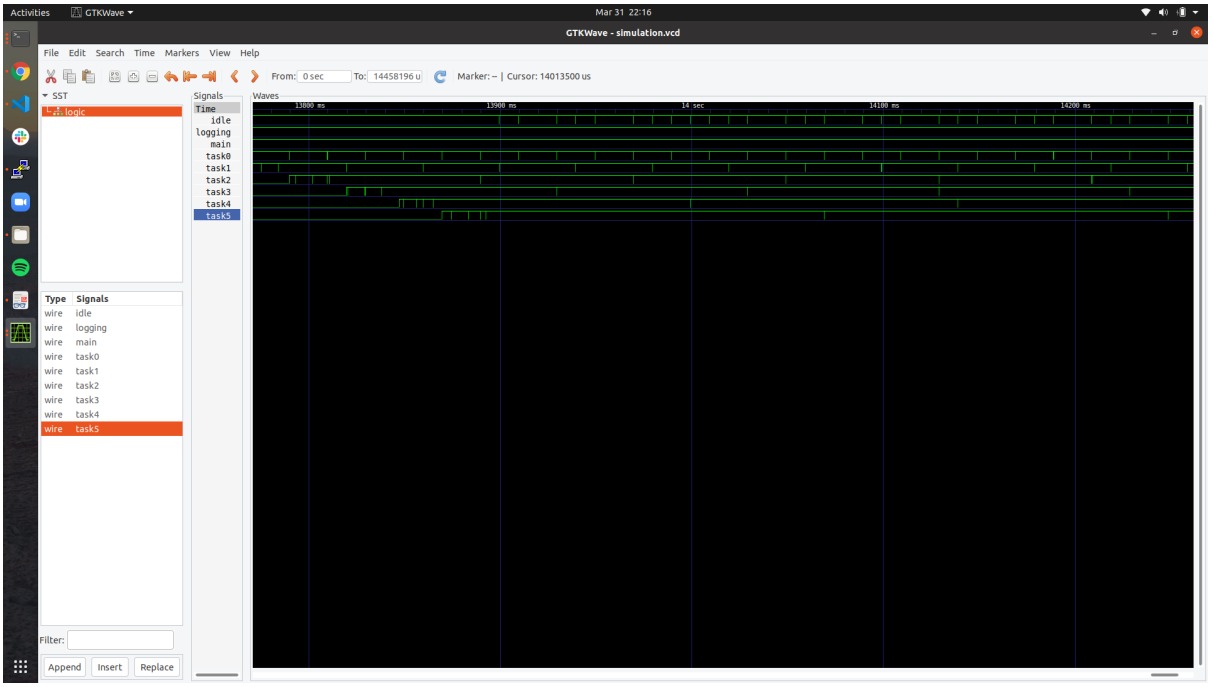

Figure 1. GTKWave screenshot with CONFIG_PRIORITY_CEILING=0

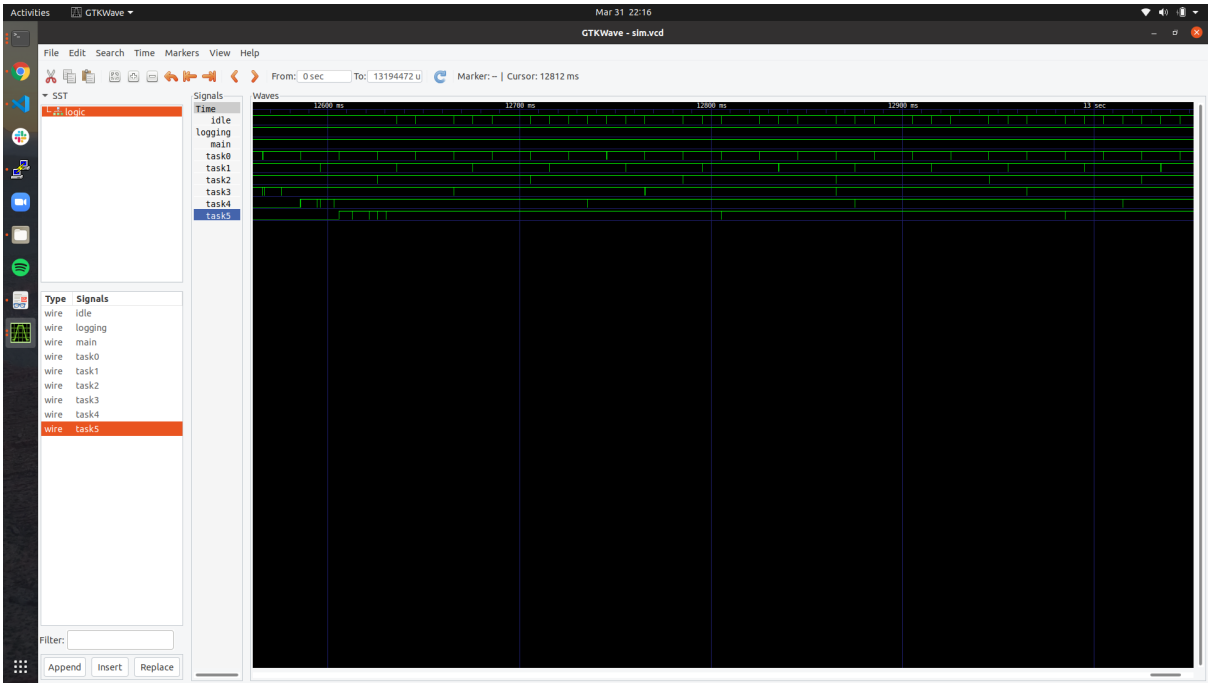

Figure 2. GTKWave screenshot with CONFIG_PRIORITY_CEILING=10