Final assignment ROS01

Youri Klaassens[a], *Nick van Endhoven*[b]

[a]*student Computer Engineering Rotterdam University of Applied Science, 0996211@hr.nl, Zwaag*
[b]*student Computer Engineering Rotterdam University of Applied Science, 0998831@hr.nl, Breda*

INTRODUCTION

This assignment considers a small part of a car. Those are the airbag and a blackbox system. The goal is to model tasks such that all tasks finish their execution before their given deadlines. The working platform is TI-RTOS. A second LaunchPad fires tasks to the main LaunchPad using GPIO pins. The tasks get executed and send a signal back to the second LaunchPad. This second LaunchPad will then validate if the tasks are finished executing before their given deadline. The tasks and deadlines are defined in table 1.

Task	*Arrival time*	*BC deadline*	*WC deadline*
Compass	200 ms ± 20 ms	5 ms	35 ms
Airbag	3000 ms ± 1500 ms	30 ms	32 ms
GPS	1300 ms ± 20 ms	5 ms	70 ms

Table 1. Characteristics of the tasks.

Table 2 shows the pin mapping between two LaunchPads.

Description	*Pin*	*GPIO*	*IN/OUT*
Compass event	61	6	IN
Airbag event	62	7	IN
GPS event	63	8	IN
GPS response	64	9	OUT
Airbag response	1	10	OUT
Compass response	2	11	OUT

Table 2. Pin mapping.

Both the *compass* and *GPS* tasks communicates via a shared UART resource using 9600 baud 8N1. This is the UART0 peripheral in the LaunchPad. The *compass* task has to print 32 bytes of ASCII data containing a counter that increments every time the task is executed. The *GPS* task has to print the same as the *compass* task except it has to print 64 bytes instead of the 32 bytes.

IMPLEMENTATION

After the device is booted an initialization process is started. The UART and GPIO parameters are initialized and configured. A main thread is created that summons different threads for each task with a given priority. Each of these tasks are configured shown in listing 1.

```
void* main_thread(void* args)
{
    struct sched_param  spc1, spc2, spc3;
    pthread_attr_t      pta_prio_1, pta_prio_2, pta_prio_3;
    pthread_t           ptc_c, ptc_a, ptc_g;

    check_errno( sem_init(&int_sem_compass, 0, 0) );
    check(pthread_attr_init(&pta_prio_1));
    check(pthread_attr_setstacksize(&pta_prio_1, 1024));
    check(pthread_attr_getschedparam(&pta_prio_1, &spc1));
    spc1.sched_priority = 1;
    check(pthread_attr_setschedparam(&pta_prio_1, &spc1));
    check(pthread_create(&ptc_c, &pta_prio_2, &compass_task, NULL));
    check_errno(pthread_join(ptc_c, NULL));
    check_errno( sem_destroy(&int_sem_compass) );

  ...

    return NULL;
}
```

Listing 1. Initialization

Each input GPIO pin gets its own callback function that will be executed on an interrupt. These interrupt are enabled in the main before the main thread is created. Listing 2 shows these callback functions. A semaphore is posted, respective to their task, when one of these events triggers. This allows the tasks running in a different thread to continue execution. The way this implemented is shown in listing 3.

```
/* ISR for GPIO 06 */
void on_compass_event(uint_least8_t index)
{
    sem_post(&int_sem_compass);
}

/* ISR for GPIO 07 */
void on_airbag_event(uint_least8_t index)
{
    sem_post(&int_sem_airbag);
}

/* ISR for GPIO 08 */
void on_gps_event(uint_least8_t index)
{
    sem_post(&int_sem_gps);
}
```

Listing 2. Interrupt callbacks

In listing 3 also shows the implementation of the *compass* task. When executed it writes a 32 byte ASCII message using UART on baud 9600. This messages contains a counter, increased each execution, and fills the rest of the message with the character 'C'. The *GPS* tasks does the same thing but it writes a 64 byte message instead with the character 'G'. The function `write_to_uart` is shown in listing 4. The *airbag* task only has to wait for 30 ms before sending a signal

bag.

```c
/* Task for compass */
void* compass_task(void* args)
{
    static int count = 0;

    while(1)
    {
        sem_wait(&int_sem_compass);

        write_to_uart(count, 'C', 32);

        count += 1;

        GPIO_write(Board_GPIO_11, 1);
        Task_sleep(1);
        GPIO_write(Board_GPIO_11, 0);
    }

    return NULL;
}
```

Listing 3. Tasks implementation

Since the UART resource has to be shared among two different threads a mutex lock is implemented. Each time a mutex will lock the `UART_write` function, preventing other threads from using it. This function is configured to be blocking, thus only continuing after it finishes the writing process. After the function is fully executed it will unlock again making it available for other threads to use. This is also shown in listing 4

```c
void write_to_uart(int cnt, char c, size_t size)
{
    char str[size];
    sprintf(str, "%d", cnt);

    int cnt_size = 0;
    int i = cnt;

    if(cnt == 0)
        cnt_size = 1;

    while(i != 0)
    {
        i /= 10;
        cnt_size += 1;
    }

    for(;cnt_size < size - 2; cnt_size++)
        str[cnt_size] = c;

    str[size - 3] = '\n';
    str[size - 2] = '\r';
    str[size - 1] = '\0';

    check_errno(pthread_mutex_lock(&uart_lock));
    UART_write(handle, str, size);
    check_errno(pthread_mutex_unlock(&uart_lock));
}
```

Listing 4. Write to UART implementation

RESULTS

Unfortunately we did not manage to get it fully working. Our program seems to unexpectedly crash whenever the two threads try to enter the mutex locked area at around the same time. Figure 1 shows the results of the program without writing to UART.

Figure 1. Result without shared UART resource

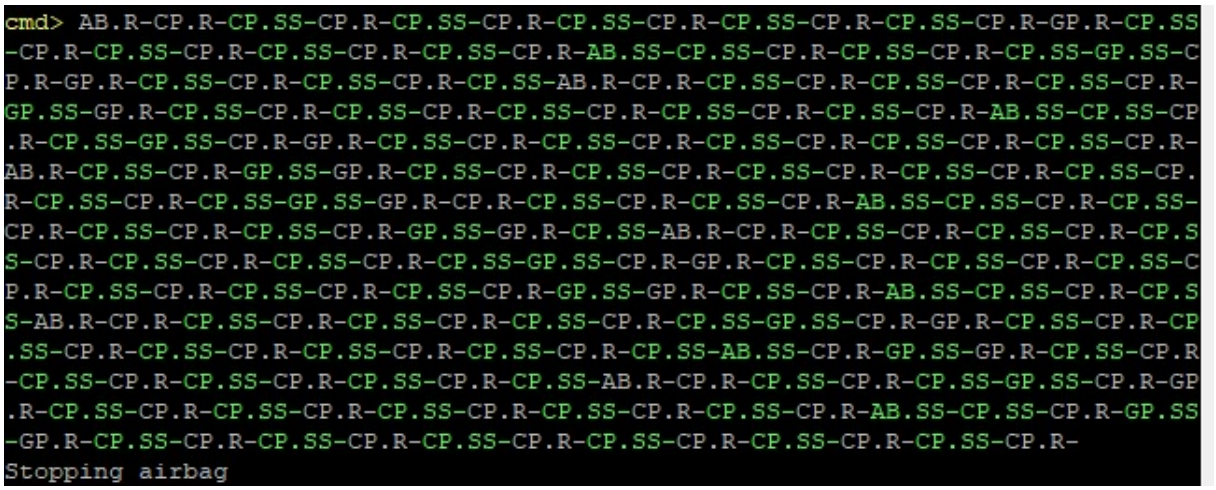

Figure 2 shows the full results of the project with the `UART_write` implementation.

Figure 2. Result with shared UART resource

CONCLUSION