CENG 448/548 — Real-Time Operating Systems

South Dakota School of Mines & Technology

Laboratory Assignment Three

The objective of this assignment is to create a reliable and efficient driver for the 16550 UART. In order to achieve this, we need to create a new and improved UART Application Program Interface (API), and an interrupt-based driver which implements that API. You will be given the API, and a Version 0.01 (barely usable) implementation. You must bring it up to Version 0.9 (mostly working) or better. By the end of the course, you must have Version 1.0 (totally working, and could add new features), or your final project will not work.

First, you should download the `Lab_3_code.tgz` file from D2L and unpack it in your directory. Note that the files `src/uart.c` and `include/uart.h` have been replaced with a partially implemented driver for the 16550 UART. The header and the C file are reasonably well commented, but the meanings of all of the bits in the registers are not completely specified. That information is available in the AXI UART manual which is also available on D2L. A few minor modifications have been made to other files, as necessary to make it work with the new UART API.

The `hello world` task has been substantially modified. It is now running at a higher priority than the the `stats` task, and the output has changed. It now prints "Hello World" followed by three numbers. The first two numbers are the maximum and minimum times between running through the main loop. The third number is the difference between the maximum and minimum. The difference between maximum and minimum times is called "jitter." The ouput of the `stats` task may be corrupted with output of the `hello world` task.

Part 1: Make the hello world task run on schedule

1. Compile the code and let it run for five minutes. Make note of The CPU time used by the three tasks, the maximum and minimum times for the `hello world` task, the amount of jitter, and whether or not you detect any corruption of the stats task output. You will include those observations in your lab report.

2. Make the `hello world` task run every 100 ms and reduce its jitter to zero. This can be done by using `vTaskDelayUntil` in the `hello world` task instead of using `vTaskDelay`. Edit `hello.c`. Comment out the call to `vTaskDelay`. There are two lines that you must uncomment in order to enable `vTaskDelayUntil` to be used. Uncomment them and save the file.

3. Compile the code and let it run for five minutes. Make note of The CPU time used by the three tasks, the maximum and minimum times for the `hello world` task, the amount of

jitter, and whether or not you detect any corruption of the stats task output. You will include those observations in your lab report.

Part 2: Fix the output corruption

1. Edit `UART_16550.c`.

    - Add the code to create the mutexes and stream buffers in the `UART_16550_init` function. for each UART. Make sure that you create **recursive** mutexes. Remember that there is not much memory, so keep the stream buffers to the minimum size that you think will work.
    - Modify the `UART_16550_put_char`, `UART_16550_write_string`, `UART_16550_tx_lock`, and `UART_16550_tx_unlock` functions to acquire and release the transmitter mutex for the specified UART.
    - Modify the `UART_16550_get_char`, `UART_16550_read_string`, `UART_16550_rx_lock`, and `UART_16550_rx_unlock` functions to acquire and release the receiver mutex for the specified UART.

2. Re-run the experiment from Part 1 of the lab. Make note of The CPU time used by the three tasks, the maximum and minimum times for the `hello world` task, the amount of jitter, and whether or not you detect any corruption of the stats task output. You will include those observations in your lab report. Has the output corruption been fixed? At what cost? Has the CPU usage changed?

Part 3: The transmit driver

The basic idea is to create a stream buffer that tasks can write to, and an ISR that takes data from the stream buffer and sends it to the physical UART. Unfortunately, the hardware design of the 16550 UART is very poor, and in order to do this reliably, we *must* implement a software state machine. There is no other way. Most of the code is already written, and we will go over it in class. You must modify `UART_16550_init`, `UART_16550_configure`, `UART_16550_put_char`, and `UART_16550_handler`. You may also modify `UART_16550_write_string` to make it more efficient. The new code will use a stream buffer, and a shared "state" variable for each UART that is only modified within critical sections (when interrupts are disabled).

The shared variable keeps track of the current state of the transmit code. There are three possible states:

1. `TX_EMPTY`: The transmit holding register, the transmit FIFO, and the transmit stream buffer are all empty. This implies that there will not be a transmitter interrupt as long as the UART remains in this state. The only way to go from this state to either of the other states is by a task calling either `...put_char` or `...write_string`.

    - If the state is `TX_EMPTY`, then `...put_char` should write the character to the UART transmit FIFO, change the state to `TX_FIFO`, and enable the transmitter interrupt.

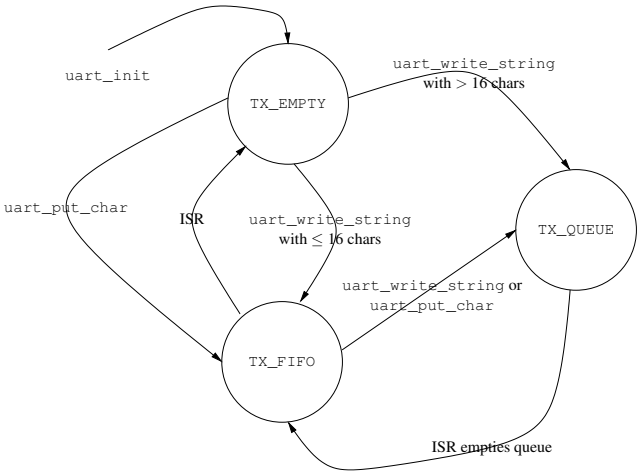

Figure 1: Software state machine for the optimal transmitter.

- If the state is `TX_EMPTY`, then `...write_string` may write up to 16 characters to the UART transmit FIFO, and change the state to `TX_FIFO`. If there are any remaining characters, then it should place them in the transmit stream buffer and change the state to `TX_STREAMBUFFER`. If the stream buffer becomes full, the function must exit the critical section and do a blocking write to the stream buffer. This will block the current task and allow other tasks to run, but since the current task still holds the mutex, no other task will get to write to the stream buffer until the current task releases the mutex. In any case, `...write_string` should enable the transmitter interrupt.

2. `TX_FIFO`: The transmit holding register and the UART transmit FIFO contain data, but the stream buffer is empty. In this state, `...put_char` and `...write_string` should write directly to the stream buffer and change the state to `TX_STREAMBUFFER`. If the ISR gets called in the `TX_FIFO` state, it should change the state to `TX_EMPTY` and disable the transmitter interrupt.

3. `TX_STREAMBUFFER`: If the UART is in this state, then the UART transmit FIFO contains data, and there is data in the stream buffer. In this state, `...put_char` and `...write_string` should write directly to the stream buffer. If the ISR gets called in the `TX_STREAMBUFFER` state, it should copy up to 16 characters from the stream buffer to the UART transmit FIFO. If the stream buffer becomes empty, it should change the state to `TX_FIFO`.

The `...put_char` and `...write_string` functions must acquire the UART mutex before taking any other actions. Additionally, the `...put_char` and `...write_string` functions must only change the state variable when the UART interrupt is disabled. The `taskENTER_CRITICAL` and `taskEXIT_CRITICAL` macros can be used to enable and disable *all* interrupts. You must keep the critical sections very, very short.

The optimal software state machine is shown in Figure 1. An alternative approach is to have `...write_string` simply call `...put_char` repeatedly. This simplifies the software state machine, as shown in Figure 2. You may wish to implement this simplified version first, and then update the `...write_string` function after the `...put_char` function is working.

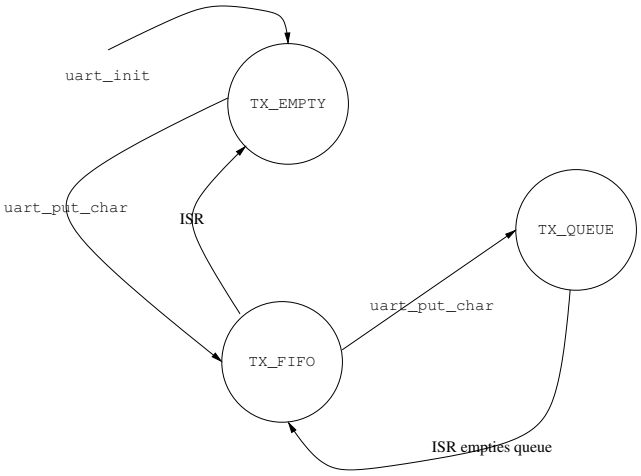

Figure 2: Software state machine for the "easy" transmitter.

As a first step, you can leave the ...`write_string` function as it is and just work on the ...`put_char` function. In this case, the ...`write_string` function only needs to acquire the mutex, then send each character to the ...`put_char` function, and release the mutex at the end. You should have set it up to do that in Part 3 of this assignment This approach makes the software state machine much simpler, as shown in Figure 2, but it is much less efficient. You can modify the ...`write_string` function to be more efficient after getting the ...`put_char` function to work.

Part 4: Comparison

Repeat the data gathering procedure from Part 1 and Part 2, and report any differences between the new code and the original code. Is the corruption fixed? Does the `hello world` task run every 100 ms? Is the jitter reduced to zero milliseconds?

Part 5: The receive driver

This code is much easier. Configure the UART so that a receiver interrupt is triggered when the UART receiver FIFO is almost full, or when a timeout has occurred. The ISR simply copies all of the data from the UART receiver FIFO to the receiver stream buffer. If the stream buffer is full, some data will be lost. The ...`get_char` function reads from the stream buffer. The original ...`read_string` function should work as it is. It should just acquire the receiver mutex, call the ...`get_char` function repeatedly, and release the mutex at the end. It could be improved to directly read multiple characters from the stream buffer.

Write a new task that reads characters from the UART and echoes them back to the user. Disable the `hello` task and the `stats` task.