

Real Time Operating Systems Design and Programming

Lab 5

OS Concurrency Lab

Producer and Consumer

Contents

1	Overview.....	2
2	The environment.....	2
3	Lab Procedure.....	2

1 Overview

In this exercise, you will focus on the producer-consumer problem. You may want to use some primitives of the RTX in order to solve the problem. There may be many possible solutions and you are expected to evaluate their advantages and disadvantages.

2 The environment

You will begin from the simple one producer and one consumer scenario where the producer will repeatedly produce an integer ranging from 0 to 19 at a random rate. The consumer will consume the integer created by the producer. Both tasks share a single FILO buffer. Notice, the buffer provided could easily result in buffer overflow or underflow. Overflow is less of concern and you can assume you have infinite space for buffer. The buffer size is predefined as ten and should be enough for the lab. Determining the optimised size of the buffer requires the knowledge of queuing network and performance evaluation, which is not expected for this lab. So the point is to avoid the consumer consuming an empty buffer, which will cause underflow. Moreover, the RTX scheduling applies round-robin policy, which means it will pre-empt producer or consumer periodically, this is a huge source of data race conditions and you are **not** allow to change the round-robin settings in order to simulate the multicore environment.

Some of the random timing is generated by `rand()` function, you may want to change the defined `RANDOM_SEED` value to see different timing results.

In this lab, you may use the watch window and the logic analyser functions of μ Vision - see the lab procedure for more details (This requires hardware debugger support. STLINK2, for example, supports these functions).

3 Lab Procedure

1. Open the template project. Scan through the code.
2. Notice there are few global variables: `producer_datum`, `consumer_datum` and `i`. The variable `i` keep tracks of the buffer order and points to the next available buffer space. The reason they are declared as global variables instead of local variables is that μ Vision cannot determine the value of local variables as their scope is confined by the local function, meaning they are probably stored in a CPU register, which μ Vision is unable to access during run time.
3. First of all, make sure only the producer is running by commenting the line:

```
taskID2=os_tsk_create(Consumer,0);
```

Build and load the program, then enter debug mode. Add variables `i`, `producer_datum`, `consumer_datum` to the watch window, if the project template has not done this for you already. This means you can observe the values of these variables in run time.

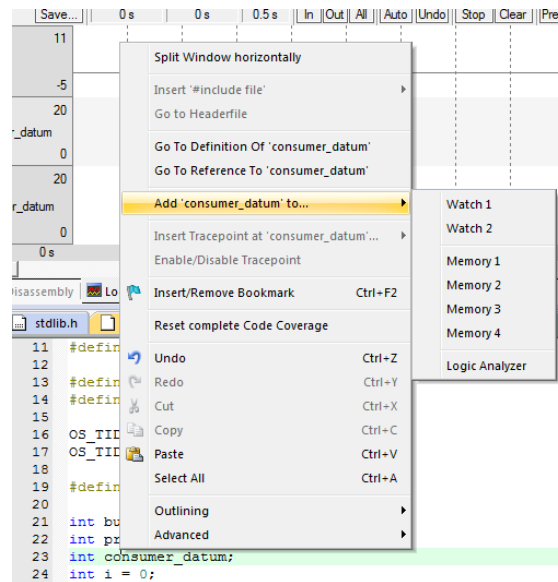


Figure 1 Right click on the variable and select Add to - Watch 1

Now run the program for a while, describe what you see and try to give an explanation regarding the value of `i`.

The blue LED is flashing as the producer is generating integers. Watch memory window 1 which should indicate that the `producer_datum` is always changing while `consumer_datum` remains zero. The `i` will be incremented every time the producer produces a new integer. Buffer **overflow** will occur eventually if the program runs for long enough so that `i` runs larger than 9; this may cause the program to be trapped in some error handler as the producer overwrites some other memory blocks!

- Now do the opposite, just run the consumer and comment the line:

```
taskID2=os_tsk_create(Consumer,0);
```


Rebuild, load your program. Enter debug mode and run, describe what you see and try to give an explanation regarding the value of `i`.

The green LED is flashing as the consumer is consuming integers. Watch memory window 1 which should indicate that the `consumer_datum` is always changing while `producer_datum` remains zero. The `i` will be decremented every time the consumer consumes a new integer. Buffer **underflow** will occur right at the beginning. This will do no harm to the program as the consumer does not change the memory content.

- Now run both the producer and consumer; repeat what you have done for Q4 and Q3. Describe what you see and try to give an explanation regarding the value of `i`. You are free to change the delay time in `append()`, `extract()`, `produce()` and `consume()` to see how it affects the result.

Blue LED and green LED are flashing. The *i* will sometimes be incremented by the producer or be decremented by the consumer depending on the random time delay. If *i* is above zero, then *consumer_datum* will copy the previous value of *producer_datum*, otherwise its value is unpredictable as it is reading from another part in the memory (buffer **underflow**). It is also possible that buffer **overflow** occur (in the very long run, or the delay time for extract and consume is longer) if *i* grows larger than 9. As the producer is writing to the memory, either buffer underflow or buffer overflow will contaminate memory space, which could eventually trigger the error handler.

6. An alternative way to examine the problem is to use the logic analyser given that your board supports tracing- if not feel free to skip this part. The logic analyser from μ Vision can sample the value of specific memory location and plot the value so that everything can be represented in a visual and potentially more intuitive way. Do the following to configure the logic analyser if the template project has not already done this for you.

Exit debug mode first if you are debugging. Select Target Options  or ALT-F7. In the Debug tab, click on the Settings next to the debugger selector. Choose the Trace tab in the Cortex-M Target Driver Setup. Configure the Trace tab as follows, selecting a core clock suitable for your board (not that by default the core clock runs at 16MHz on the Discovery board):

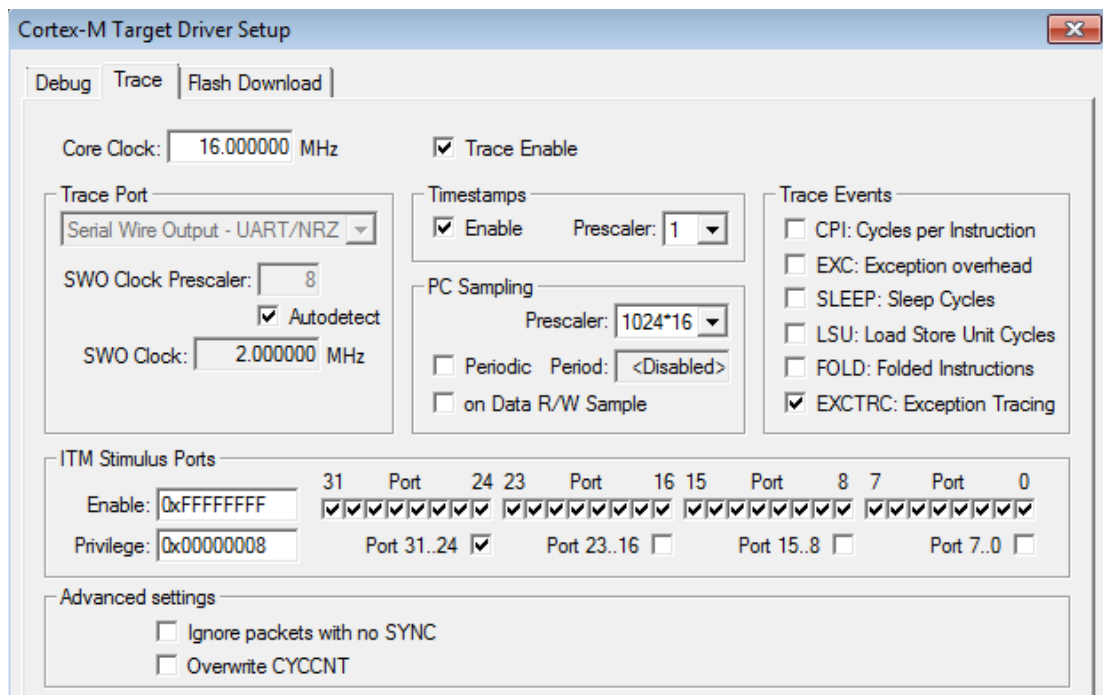



Figure 2 Set up SWO trace

This enables the debugger to trace the CPU activity through serial wire output in real time.

After this, enter debug mode, and click the logic analyser icon . You can add variables to the logic analyser by right clicking on the variable and selecting add to logic analyser. You can also directly set up everything in the Setup window of the logic analyser. Additionally you can adjust the display range of

monitored variables- for example, set both producer_datum and consumer_datum's Max and Min to 19 and 0 respectively.

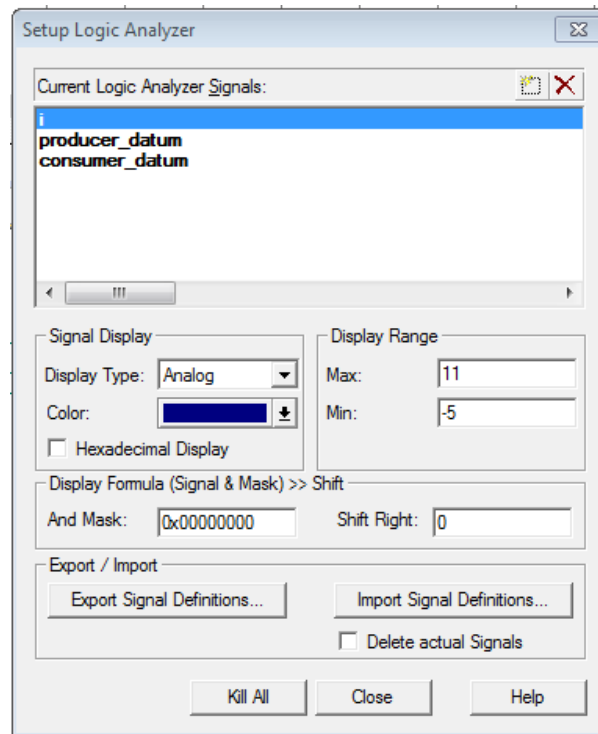


Figure 3 Setup Logic Analyzer

Once you have set up the logic analyser, you can run your code and monitor the corresponding values of monitored variables.

Run your code in debug mode, watch the logic analyser and check if it matches with your result in Q5. A simple way to check if things are working well is to compare the trace of producer_datum and consumer_datum. Also checking the value of i could lead to some conclusions.

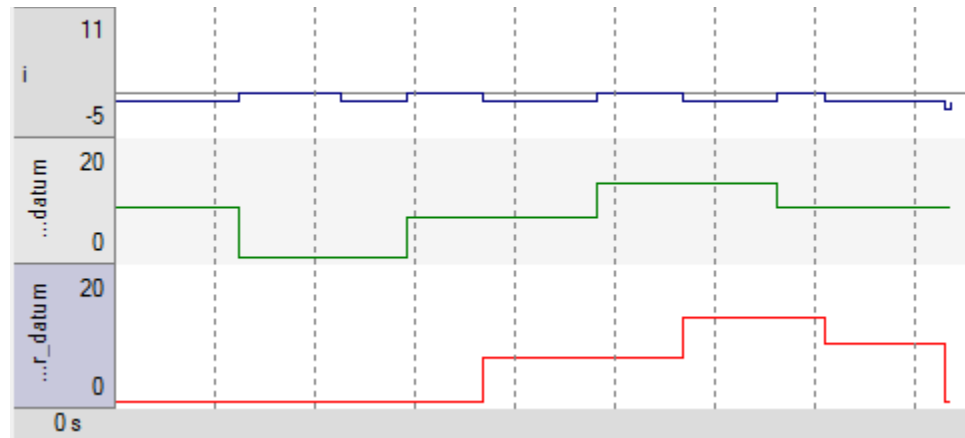


Figure 4 A Single Play

You are free to think of and implement your own solutions at this point to tackle the problem, ideally without delaying either producer or consumer unnecessarily.

7. It is suggested that the access to buffer should be made mutually exclusive. Does that work and if so, how? Try to implement this idea and see if it solves the problem, explain why or why not. You may refer to the slides on how to use the mutex primitive in RTX.

See the solution project for the code.

Implementing only the mutex has no positive effect at all. The consumer can still extract a datum from an empty buffer. Things could get worse if buffer underflow contaminates mutex related information. For example, if your producer can no longer release the mutex, it is probably due to the underflow contamination. This can be easily seen if the delay time for consumer is significantly reduced.

8. Using the mutex may not be really useful for this simple scenario. However, can you see how it will be useful in more complicated scenarios? Try to explain under what circumstances it will be a good idea.

It is generally needed as `appendix()` and `extract()` are not atomic. It is possible that one task is preempted while it actually appended or extracted and waiting to modify the `i`. Mutex can be useful when there are multiple writers. Also it is necessary to make sure that underflow doesn't happen so that mutex related information will not be contaminated.

9. It is suggested that by strictly controlling the sequence of the program flow, this problem can be solved. Try to modify the Producer and the Consumer Tasks to make sure there will be at least one produced datum in the buffer before the consumer consumes. Run your code and see if it works or not, and try to explain the result. You may refer to the slides on how to use the mutex primitive in RTX.

See the solution project for the code.

Semaphore can be utilised to implement this signalling mechanism. As it does solve the buffer underflow problem if `append()` and `extract()` are atomic. However, when they are not atomic, these could still lead to unpredictable memory corruption problem or wrong reading sequence of data. This is however not easy to notice in the provided template, but if you prolong the delay time of `extract()` significantly, it will become obvious.

10. The following producer task and consumer task code can successfully tackle the problem:

```
init(){
    sem_init(&sem1,1);
    sem_init(&sem2,0);
    ...
}

Producer(){
    Datum=produce();
    sem_wait(&sem1);
    append(datum);
    sem_post(&sem1);
    sem_post(&sem2);
}

Consumer(){
    sem_wait(&sem2);
    sem_wait(&sem1);
    datum=extract();
    sem_post(&sem1);
    consume(datum);
}
```

Implement this and explain why it tackles the problem by explicitly analysing roles of `sem1` and `sem2`. You should pay particular attention to their initialisation.

[See the solution project for the code.](#)

This producer consumer problem actually includes the reader and writer problem as the buffer is shared. So mutex is definitely needed, this is why `sem1` is presented as a binary semaphore, which is effectively a

mutex that makes sure the access to buffer is not interrupted. Sem2 keeps track of the number of datum in the buffer and stops the consumer from extracting from an empty buffer.

11. What if the two `sem_send()`s in the producer task are swapped? Does that matter? Explain why.

See the solution project for the code.

It is not necessary to implement the code to see the result. Pre-emption in any other code does not affect the result naturally, it only affects the result if pre-emption occurs between these two `sem_send()` lines. In that case, the counter of datum will be incremented first, then pre-empted by the consumer; the consumer will not be allowed to enter the buffer even if it has the ticket for `sem1` due to the fact that `sem2` is blocked by producer, so it does not matter. However it is better programming practise to release the mutex as long as the code exits the critical section.

12. What if the two `sem_wait()`s in the consumer task are swapped? Does that matter? Explain why.

See the solution project for the code.

It is not necessary to implement the code to see the result. Assume pre-emption occurs between the two swapped lines; deadlock could occur because the consumer is holding the access to the buffer while waiting for the new datum, if there is no datum, then producer will not be able to create either as it cannot append new datum to the buffer unless consumer releases `sem1`.

13. It is suggested that by combining the method mentioned in Q7 and Q9, you can figure out another solution, which is effectively the same as the method in Q10. Try to implement this and run your code. Are they the same? If yes, explain why. If not, what are the differences and which is better?

See the solution project for the code.

This implies a signalling scheme with mutex is required. Actually `sem1` in Q10 plays the role of the mutex, so in terms of solving the problem, they are effectively the same. The difference between using a mutex and semaphore to access the buffer is that a task can increment the semaphore even if it does not hold a ticket for it, whereas a mutex can only be released by the task currently accessing it. So it may be a better idea to use mutex so that other tasks cannot corrupt your access to the buffer. But given that your code is correct and neat, they are the same.

14. This is a bonus question. It is possible to implement a general semaphore using an integer variable (assume it is private and other functions cannot modify it) and two binary semaphores. How? You may draw inspiration from Q10. Assume you have the following global variables:

```
OS_SEM semIDA;  
OS_SEM semIDB;  
int tickets;
```

Produce the implementation of the following based on RTX semaphore primitive (assume it is binary and set all timeouts to infinite):

```
void general_init_sem(int initial_value);  
void general_wait_sem(void);  
void general_send_sem(void);
```

Notice that the RTX semaphore is actually general. If you like, test your answer, running the template project may not reflect the real situation!

[See the solution project for the code.](#)