Code Explanation

testpreempt.c:

I implemented this code similar to the testcoop.c so I would like to explain only the difference parts.

- For producer and consumer, I used the __critical {} to ensure my code is accessed atomically. By doing this I can remove the calling of Threadyield.

Preemptive.c:

I implemented this code similar to the cooperative.c so I would like to explain only the difference parts.

- For initialize and enable timer 0, I follow the provided google docs
- Then I used 'EA' (Enable Interrupts) to control whether interrupts are globally enabled or disabled during critical sections of the program. So the format of the code using this EA will be something like this:

  ```
  EA = 0;  // Disable interrupts temporarily
  // Critical section (e.g., thread creation, saving state)
  EA = 1;  // Enable interrupts after the critical section
  ```

  By disabling interrupts, I can ensure that these operations aren't interrupted and after that it can resume and the system can handle tasks like context switching or responding to events like timers

- Then for my timer0handler, I follow the provided information from google docs including using RETI instead of RET (to return from the interrupt). My implementation is based on my ThreadYield function.

Screenshots for compilation

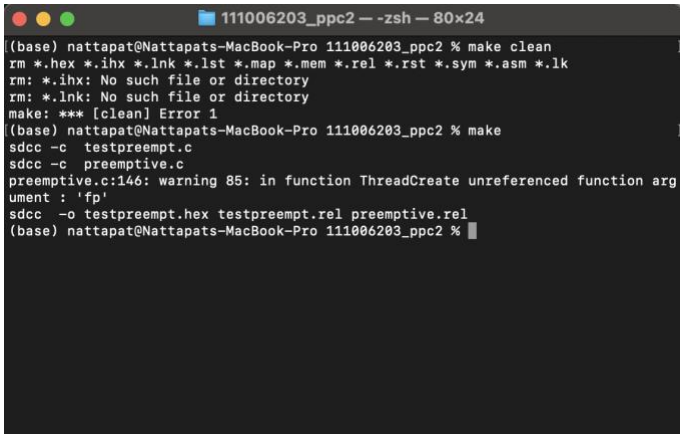

Figure 1 Screenshot for compilation

Screenshots and explanation

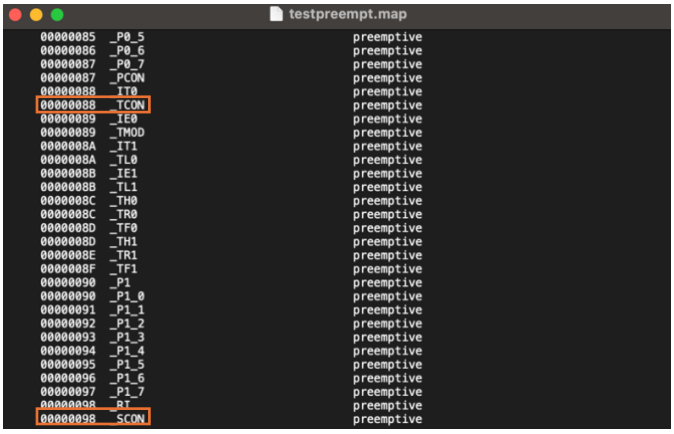

Figure 2 testcoop.map (will be used for explanation in the following section)

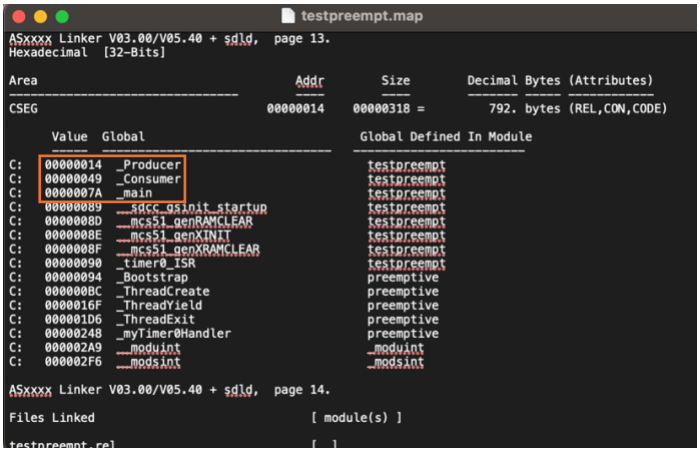

Figure 3 testcoop.map (will be used for explanation in the following section)

Before each ThreadCreate call

- ThreadCreate(main), Address of main is 0x7A, as shown in figure 3

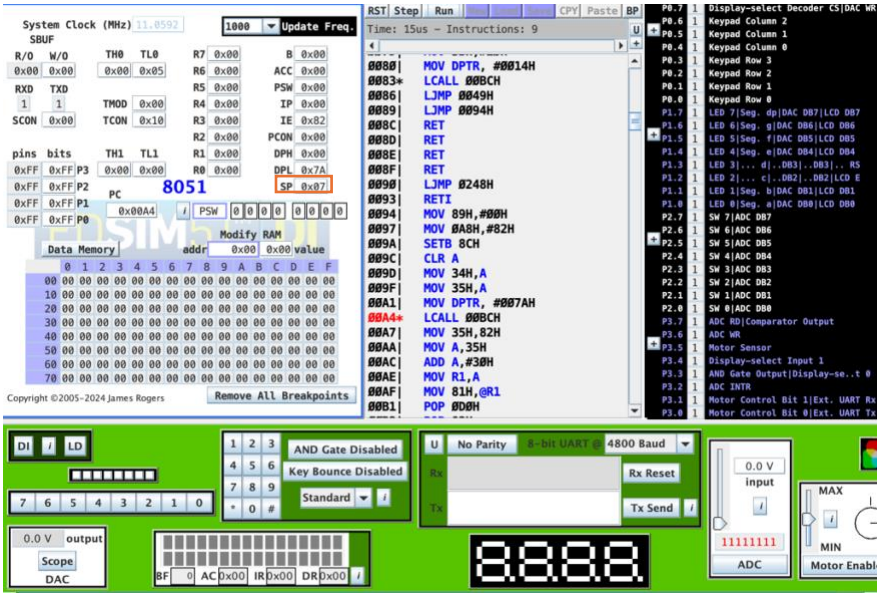

Figure 4 ThreadCreate(main)

When perform LCALL, the DPTR with the value 0x7A (which is the address of the main) the return address is pushed into the stack 2 bytes which leads into the changes of SP from 0x07 -> 0x09.

- ThreadCreate(Producer), Address of producer is 0x14, as shown in figure 3

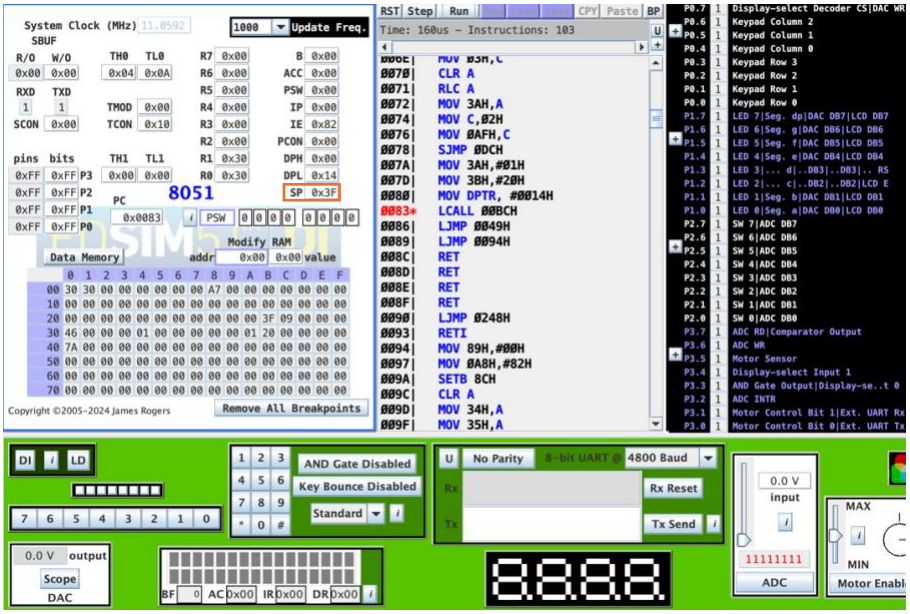

Figure 5 ThreadCreate(Producer)

When perform LCALL, the DPTR with the value 0x14 (which is the address of the producer), the return address is pushed into the stack 2 bytes which leads into the changes of SP from 0x3F -> 0x41.

When the producer is running

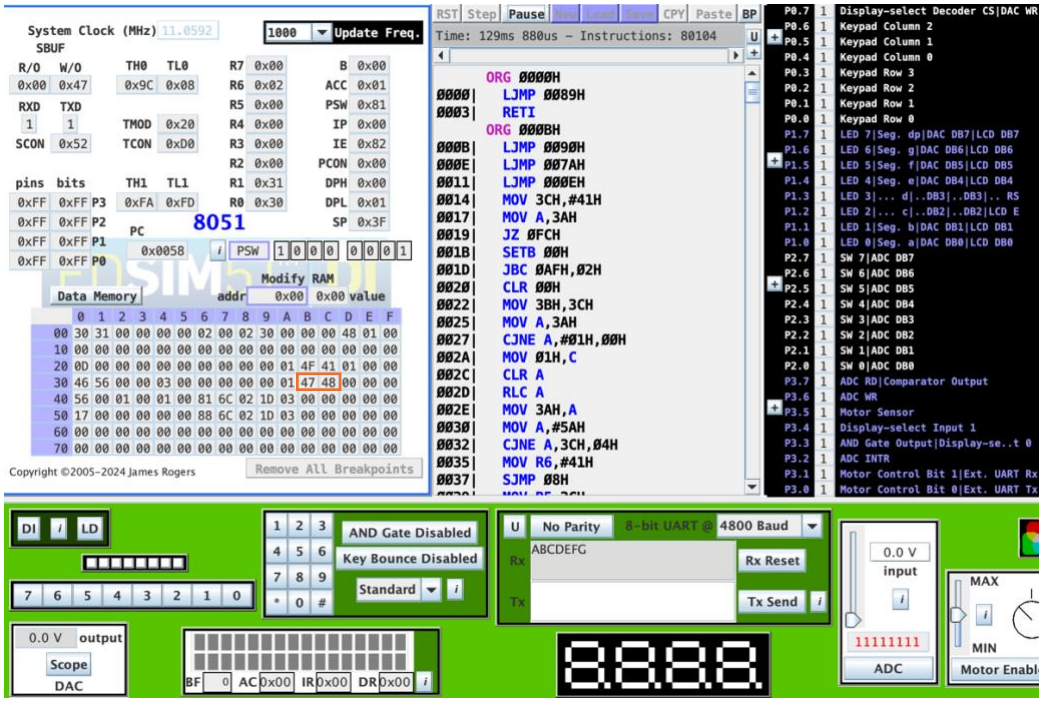

Figure 6 Producer (running)

As I have set *nextChar* (indicate which character will be transmit into SBUF) and *buffer* (represent shared buffer) in the address of 0x3B and 0x3C, respectively. Since producer is the one who generates a character and keep it in the shared buffer, therefore if the values in those addresses change that means the producer is running. (And it's changing over time)

When the consumer is running

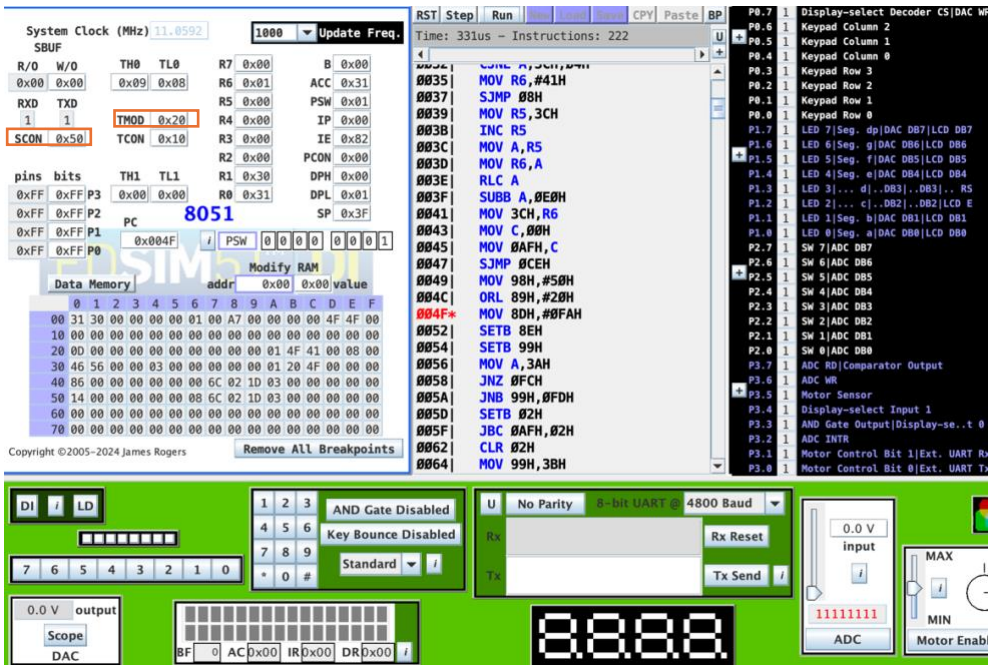

Figure 7 Consumer (running)

When the consumer is running, the value of TMOD and SCON will be changed to what we set in the code, which is (|= 0x20, preserves the Bootstrap code's setting) and 0x50, respectively.

How can you tell that the interrupt is triggering on a regular basis?

Yes, the interrupt triggers regularly because When TH0 and TL0,Timer 0 (TMOD = 0, in bootstrap) reaches its **maximum count and overflows** (goes back to 0), it automatically triggers the interrupt, calling the function specified for handling Timer 0 interrupts. This invokes myTimer0Handler, which performs thread switching. The regular execution of threads confirms the periodic interrupts