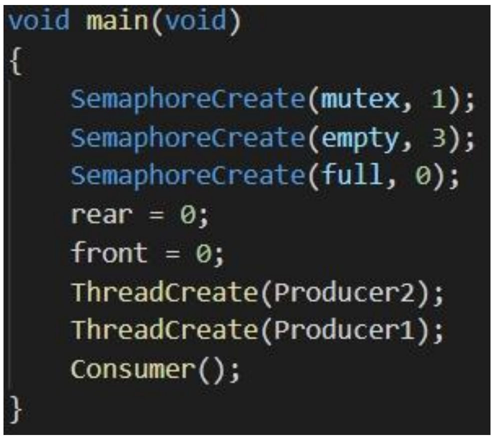
Program Project Checkpoint 4

106070034 孫振原

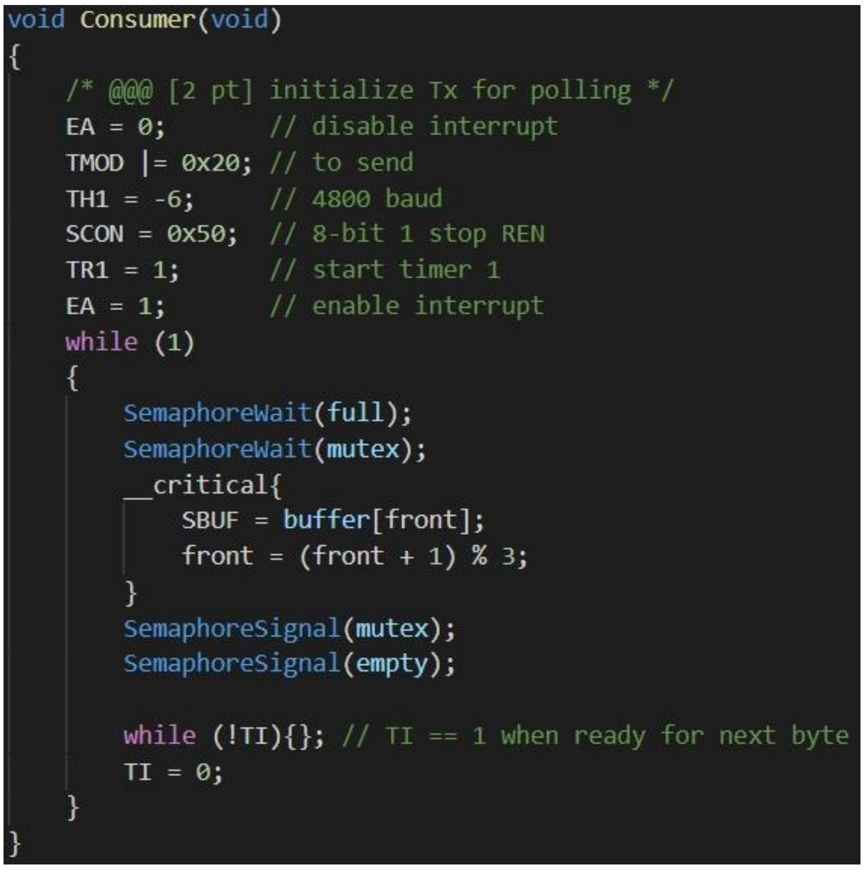
1. **main() function**



```c
void main(void)
{
    SemaphoreCreate(mutex, 1);
    SemaphoreCreate(empty, 3);
    SemaphoreCreate(full, 0);
    rear = 0;
    front = 0;
    ThreadCreate(Producer2);
    ThreadCreate(Producer1);
    Consumer();
}
```

After the bootstrap, we create thread0 for main function first. In the main function, we create 3 semaphore for Producer-Consumer Problem, mutex, empty and full. We also set rear and front for the buffer, so the buffer can look like a circular queue. Second, we create thread1 for Producer2, thread2 for Producer1 through the ThreadCreate() function. Finally, we set the thread0 to the Consumer, so producers and consumer are using different thread and access control by semaphore.

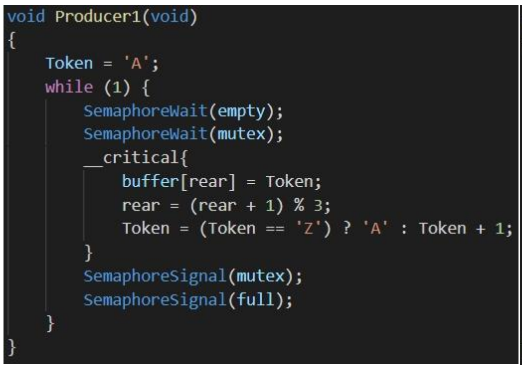
2. **Consumer()**



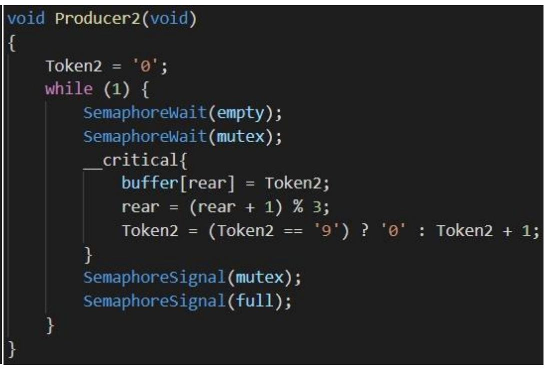
```c
void Consumer(void)
{
    /* @@@ [2 pt] initialize Tx for polling */
    EA = 0;        // disable interrupt
    TMOD |= 0x20;  // to send
    TH1 = -6;      // 4800 baud
    SCON = 0x50;   // 8-bit 1 stop REN
    TR1 = 1;       // start timer 1
    EA = 1;        // enable interrupt
    while (1)
    {
        SemaphoreWait(full);
        SemaphoreWait(mutex);
        __critical{
            SBUF = buffer[front];
            front = (front + 1) % 3;
        }
        SemaphoreSignal(mutex);
        SemaphoreSignal(empty);

        while (!TI){}; // TI == 1 when ready for next byte
        TI = 0;
    }
}
```

In the Consumer() function, we do the initialization of the timer for UART first. Second, we use a while loop to consume the characters forever. The consumer is blocked by full and mutex, that means when full >= 1 (i.e. the buffer has more than one character) and mutex == 1, the consumer can get into the critical section. Then, the consumer consume the character in the buffer by putting the character into SBUF. Finally, leave critical section and do the while loop when TI == 1.

3. **Producer()**



```
void Producer1(void)
{
    Token = 'A';
    while (1) {
        SemaphoreWait(empty);
        SemaphoreWait(mutex);
        __critical{
            buffer[rear] = Token;
            rear = (rear + 1) % 3;
            Token = (Token == 'Z') ? 'A' : Token + 1;
        }
        SemaphoreSignal(mutex);
        SemaphoreSignal(full);
    }
}
```



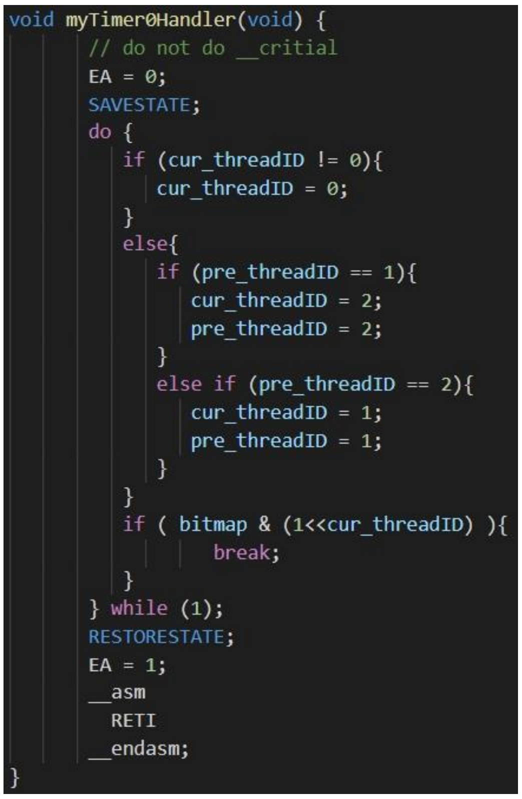
```
void Producer2(void)
{
    Token2 = '0';
    while (1) {
        SemaphoreWait(empty);
        SemaphoreWait(mutex);
        __critical{
            buffer[rear] = Token2;
            rear = (rear + 1) % 3;
            Token2 = (Token2 == '9') ? '0' : Token2 + 1;
        }
        SemaphoreSignal(mutex);
        SemaphoreSignal(full);
    }
}
```

In the Producer() function, Producer1 produces character ‘A’ to ‘Z’ by Token, and Producer2 produces number ‘0’ to ‘9’ by Token2. Same as Consumer() function, we use a while loop to produce characters forever. In the semaphore, we use empty and mutex to block producers. When empty >= 1 (i.e. there is more than one empty in the buffer) and mutex == 1, the producer can get into the critical section. In the critical section, we put the character into the circular buffer, which can afford 3 characters. Finally, leave critical section.

4. **The relation between Producer and Consumer**

The Producers and Consumer are sharing a buffer, so we need to use a semaphore to avoid mutual exclusive. When putting characters into buffer, we use rear and front to implement a circular queue as a 3-deep buffer. By doing so, either produce or consume can be represented in order and three characters in a time.

5. **myTimer0Handler**



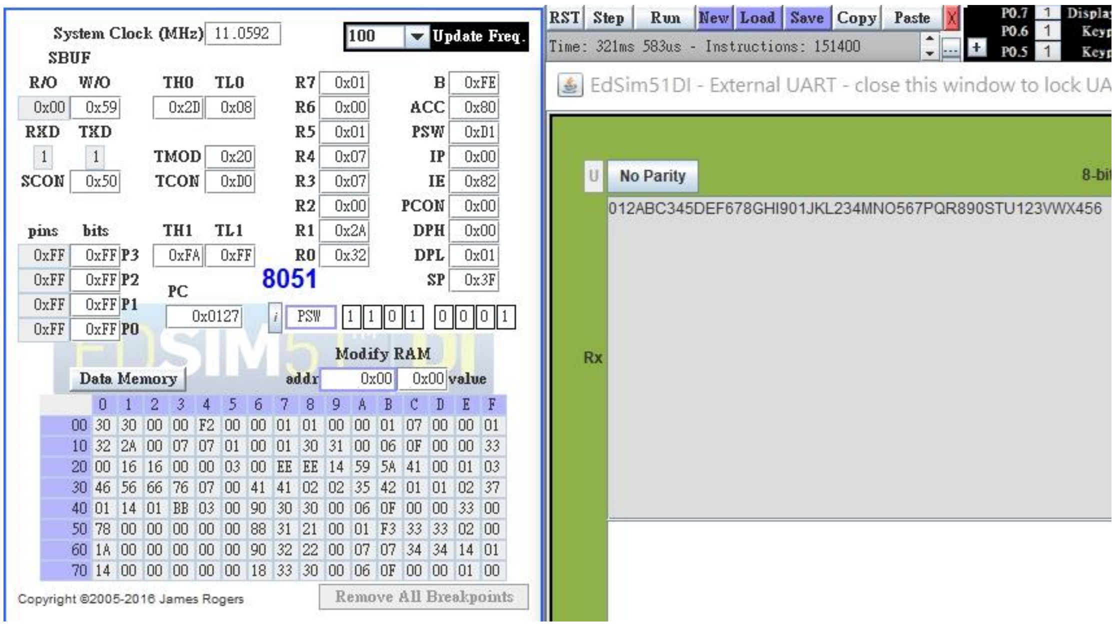
```
void myTimer0Handler(void) {
        // do not do __critial
        EA = 0;
        SAVESTATE;
        do {
            if (cur_threadID != 0){
                cur_threadID = 0;
            }
            else{
                if (pre_threadID == 1){
                    cur_threadID = 2;
                    pre_threadID = 2;
                }
                else if (pre_threadID == 2){
                    cur_threadID = 1;
                    pre_threadID = 1;
                }
            }
            if ( bitmap & (1<<cur_threadID) ){
                    break;
            }
        } while (1);
        RESTORESTATE;
        EA = 1;
        __asm
          RETI
        __endasm;
}
```

In myTimer0Handler, the policy is like the order in 0, 1, 0, 2, 0, 1, 0, 2... 0 means thread0 (Consumer), 1 means thread1 (Producer2) and 2 means thread2 (Producer1). First, we set EA = 0 in the function to assure it will not be interrupted. Then, use two variables to record current and previous thread ID to achieve the policy mentioned above. Change to the thread and check it is using or not. After restoring state, we set EA = 1, which means we enable interrupt. Finally, we write an assembly code to return.

6. **Fairness**

In this policy we mentioned above, we can assure the fairness. If we use the normal RR scheduling, like 0, 1, 2, 0, 1, 2…, the characters produced by thread1 will never be consume by Consumer. Because the buffer will be covered by thread2, the Consumer will consume the characters produced by thread2 forever. By using our policy, we can avoid the problem of starvation

7. **Screenshot**



System Clock (MHz) 11.0592 | 100 Update Freq.

SBUF R/O 0x00 W/O 0x59 | RXD 1 TXD 1 | SCON 0x50

TH0 0x2D TL0 0x08 | TMOD 0x20 | TCON 0xD0 | TH1 0xFA TL1 0xFF

R7 0x01 | R6 0x00 | R5 0x01 | R4 0x07 | R3 0x07 | R2 0x00 | R1 0x2A | R0 0x32

B 0xFE | ACC 0x80 | PSW 0xD1 | IP 0x00 | IE 0x82 | PCON 0x00 | DPH 0x00 | DPL 0x01 | SP 0x3F

pins	bits	
0xFF	0xFF	P3
0xFF	0xFF	P2
0xFF	0xFF	P1
0xFF	0xFF	P0

8051 PC 0x0127 PSW 1 1 0 1 0 0 0 1

addr 0x00 value 0x00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	30	30	00	00	F2	00	00	01	01	00	00	01	07	00	00	01
10	32	2A	00	07	07	01	00	01	30	31	00	06	0F	00	00	33
20	00	16	16	00	00	03	00	EE	EE	14	59	5A	41	00	01	03
30	46	56	66	76	07	00	41	41	02	02	35	42	01	01	02	37
40	01	14	01	BB	03	00	90	30	30	00	06	0F	00	00	33	00
50	78	00	00	00	00	00	88	31	21	00	01	F3	33	33	02	00
60	1A	00	00	00	00	00	90	32	22	00	07	07	34	34	14	01
70	14	00	00	00	00	00	18	33	30	00	06	0F	00	00	01	00

Copyright ©2005-2016 James Rogers

Time: 321ms 583us - Instructions: 151400

P0.7 1 | P0.6 1 | P0.5 1

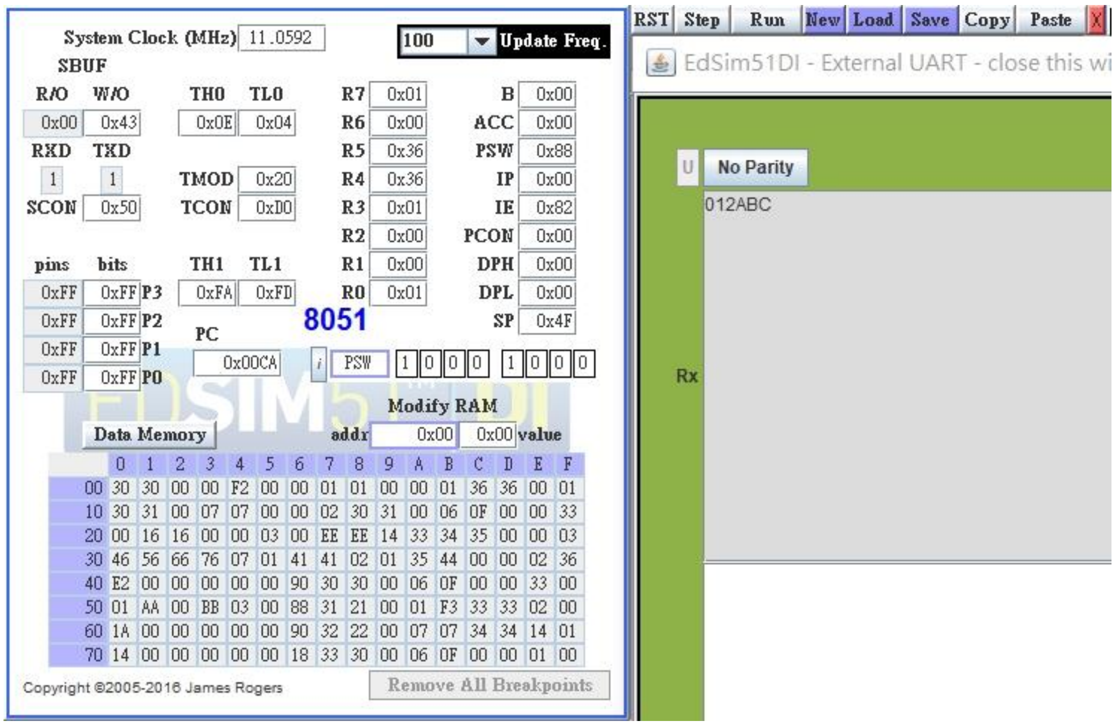
EdSim51DI - External UART - close this window to lock UA

```
012ABC345DEF678GHI901JKL234MNO567PQR890STU123VWX456
```

Explanation:

Data Memory 0x2A to 0x2C is the buffer, we can see currently is 59, 5A, 41 (i.e. Y, Z, A in ASCII code)

Data Memory 0x35 is current thread ID = 0, which means it is doing the Consumer thread and prepare to consume Y, Z, A.



System Clock (MHz) 11.0592 | 100 Update Freq.

SBUF R/O 0x00 W/O 0x43 | RXD 1 TXD 1 | SCON 0x50

TH0 0x0E TL0 0x04 | TMOD 0x20 | TCON 0xD0 | TH1 0xFA TL1 0xFD

R7 0x01 | R6 0x00 | R5 0x36 | R4 0x36 | R3 0x01 | R2 0x00 | R1 0x00 | R0 0x01

B 0x00 | ACC 0x00 | PSW 0x88 | IP 0x00 | IE 0x82 | PCON 0x00 | DPH 0x00 | DPL 0x00 | SP 0x4F

pins	bits	
0xFF	0xFF	P3
0xFF	0xFF	P2
0xFF	0xFF	P1
0xFF	0xFF	P0

8051 PC 0x00CA PSW 1 0 0 0 1 0 0 0

addr 0x00 value 0x00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	30	30	00	00	F2	00	00	01	01	00	00	01	36	36	00	01
10	30	31	00	07	07	00	00	02	30	31	00	06	0F	00	00	33
20	00	16	16	00	00	03	00	EE	EE	14	33	34	35	00	00	03
30	46	56	66	76	07	01	41	41	02	01	35	44	00	00	02	36
40	E2	00	00	00	00	00	90	30	30	00	06	0F	00	00	33	00
50	01	AA	00	BB	03	00	88	31	21	00	01	F3	33	33	02	00
60	1A	00	00	00	00	00	90	32	22	00	07	07	34	34	14	01
70	14	00	00	00	00	00	18	33	30	00	06	0F	00	00	01	00

Copyright ©2005-2016 James Rogers

EdSim51DI - External UART - close this wi

```
012ABC
```

Explanation:

Data Memory 0x2A to 0x2C is the buffer, we can see currently is 33, 34, 35 (i.e. 3, 4, 5 in ASCII code)

Data Memory 0x35 is current thread ID = 1, which means it is doing the Producer2 thread and just produce the characters ‘3’, ‘4’ and ‘5’.