107062381 楊孝偉

Checkpoint 2 Report

Testpreempt.c

In this file, I will use only two variables : buffer and token. Buffer is used for the shared buffer between the producer and consumer. Token is used to generate characters from A to Z for the producer.

In the main function, buffer is initialized to '1", which simply means that buffer is empty and producer can generate an alphabet to it. Token is initialized to A. Main will spawn consumer thread, while main itself calls producer. This is done to not waste main's thread, more economical.

In the producer function, before it enters the infinite loop, I initialized the token to 'A' first, to make sure that token is really initialized to 'A'. After it enters the loop, it will check whether buffer is not equal to 1, if yes, then buffer is still full (consumer hasn't consumed yet), producer will keep polling and rely on the preemptive mechanism. Else if buffer is already empty, buffer will take the value in token, then update the value in token. When buffer is taking the value of token, it should be in critical section since SDCC suggests that you can surround the code fragment where they access the shared variable, in this case it's buffer, to ensure that the shared variables are accessed atomically.

In the consumer function, before it enters the infinite loop, we initialize Tx for polling. After it enters the loop, it will check whether buffer is equal to 1, if yes, then buffer is still empty (producer hasn't produced it yet), then it will keep polling. Else, SBUF(reading or writing register) will take the value in buffer, then we reset buffer to 1 to indicate if it's empty, then we check if Tx is busy (serial port hasn't finished writing it yet), then we keep polling again. Finally, we reset It to 0 again.

Preemptive.c

In this file, I use quite a lot of variables, I will explain some later as I explain the functions. Sp[4] is used for the saved stack pointers. Cur_thread is for the current thread ID. Bitmap is used to determine which thread ID is a valid thread.

First, we define SAVESTATE. It is a C macro for saving the context of the current thread and it is written in inlined assembly. First we push the ACC, B register, DPTR, and PSW onto stack. Then we save the stack pointer for the current thread into the saved stack pointers array as indexed by the current thread ID.

Next, we define a C macro for restoring the context of the current thread by basically doing reverse the operation of SAVESTATE.

Bootstrap is the start-up code to set up and run the first thread. First we initialize the bitmap to 0000 (which is 0 in decimal value), which means that all of the threads are available for use. Then we create a thread for main and set current thread to this thread ID and restore its context with RESTORESTATE so that it starts running the main function. Stack was set up by the threadcreate(main). Stack 0 now has the return address of main.

Next is the thread create function to create a thread data structure so it is ready to be restored. First we check to see whether we have not reached the maximum number of threads. Maximum number of threads is reached if the bitmap value is equal to 15 (which is 1111 in binary value), because we will set the bit value to 1 if a thread is used. If so, we will return -1 which is not a valid thread ID. Otherwise, we will find a thread ID that is not in use and grab it. I did this by first initializing a variable called mark to 1 (this means that mark will have the value of 0001 in binary initially). Then it will enter the while loop. First it will look for which thread is available by keep shifting the mark to the left by 1 bit position. If there is an available thread, then bitmap & mark value will be 0, then we simply set the new thread accordingly. Then we update the bitmap value to indicate that the selected thread is now used by XOR-ing the bitmap with mark. Then we calculate the starting stack location for new thread. We will check if new thread is 0, 1, 2 or 3. If new thread is 0, we set the address to 0x3F as the hardware stack in 8051 is pre increment. If new thread is 1, 2 and 3, we set the address to 0x4F, 0x5F, 0x6F respectively. Then we save the current SP in a temporary and set SP to the starting location for the new thread, which is address basically. Then we push the return address of fp onto stack by pushing DPL and DPH as in SDCC convention, 2-byte ptr is passed in DPTR, but push instruction can only push it as two separate registers, DPL and DPH. Then we initialize the registers to 0. So we assign ACC to 0 by ANL, an instruction to perform a bitwise logical AND operation between the specified operands and stores the result in the destination operand (ACC), and push it four times for ACC, B, DPL and DPH. Then we need to push PSW registers. RS1 and RS0 is used for register bank selection. Therefore, we can fill RS1 and RS0 by shifting the new thread ID by 3 as RS1 and RS0 is 4 and 3 bits away from the least significant bit. Then we push the PSW. Then we can write the current stack pointer to the saved stack pointer array, SP will take the previously saved SP, and return this newly created thread ID.

Thread Yield function is called by a running thread to yield control to the other thread. First it will find the next thread that can run and set the current thread ID to it. Then if the bitwise AND value of bitmap and the current thread ID is not equal to 0, we break from the while loop. This means that thread can run, so we break from the loop. If the bitwise AND value of that thread ID and bitmask is equal to 0, this means that thread ID is still not active, then the while loop will continue to find the next thread that can run. It is guaranteed that at least one thread is active, so this loop will always terminate.

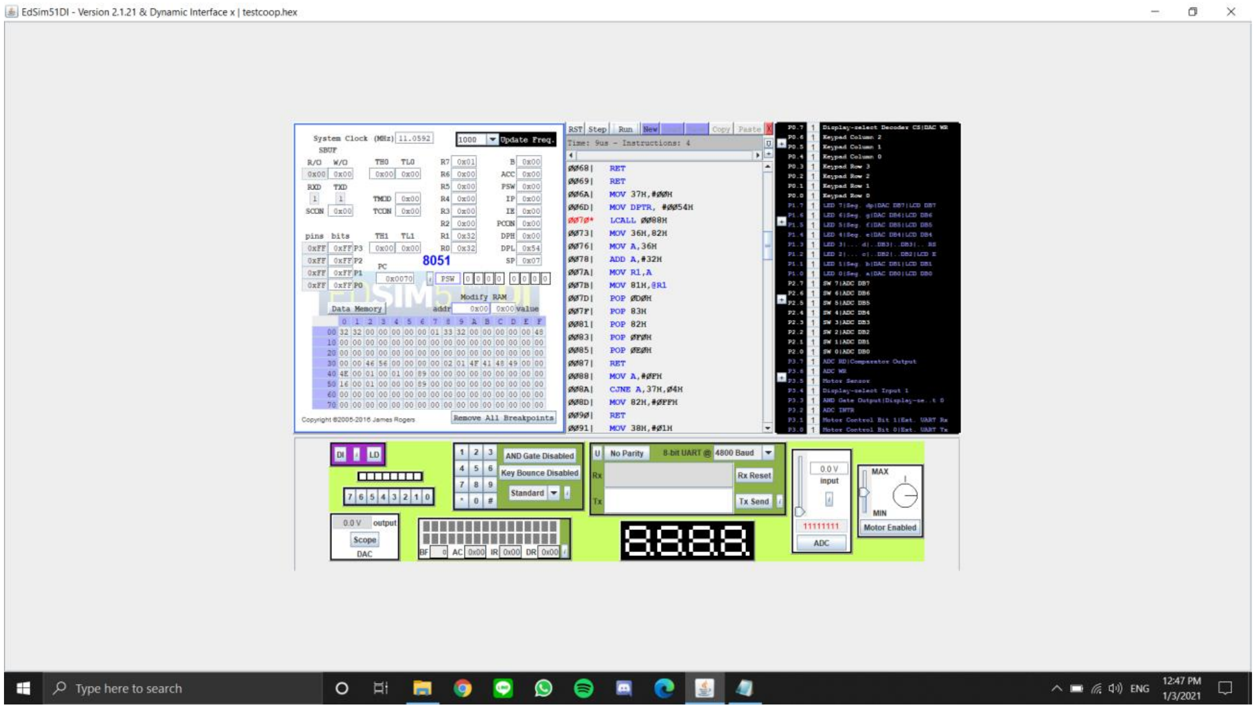
I did not fill the Thread Exit function as this function is not used anywhere.

myTimer0Handler function to be the ISR for Timer0 to serve the purpose of preemption. I used the code from ThreadYield, added the RETI assembly instruction, saved the content of R0 to R7 to registers that have been saved by RESTORESTATE. This is done to preserve the value of R0 to R7.

Breakpoints

1. Threadcreate(main)
   Before calling threadcreate(main)

107062381 楊孝偉

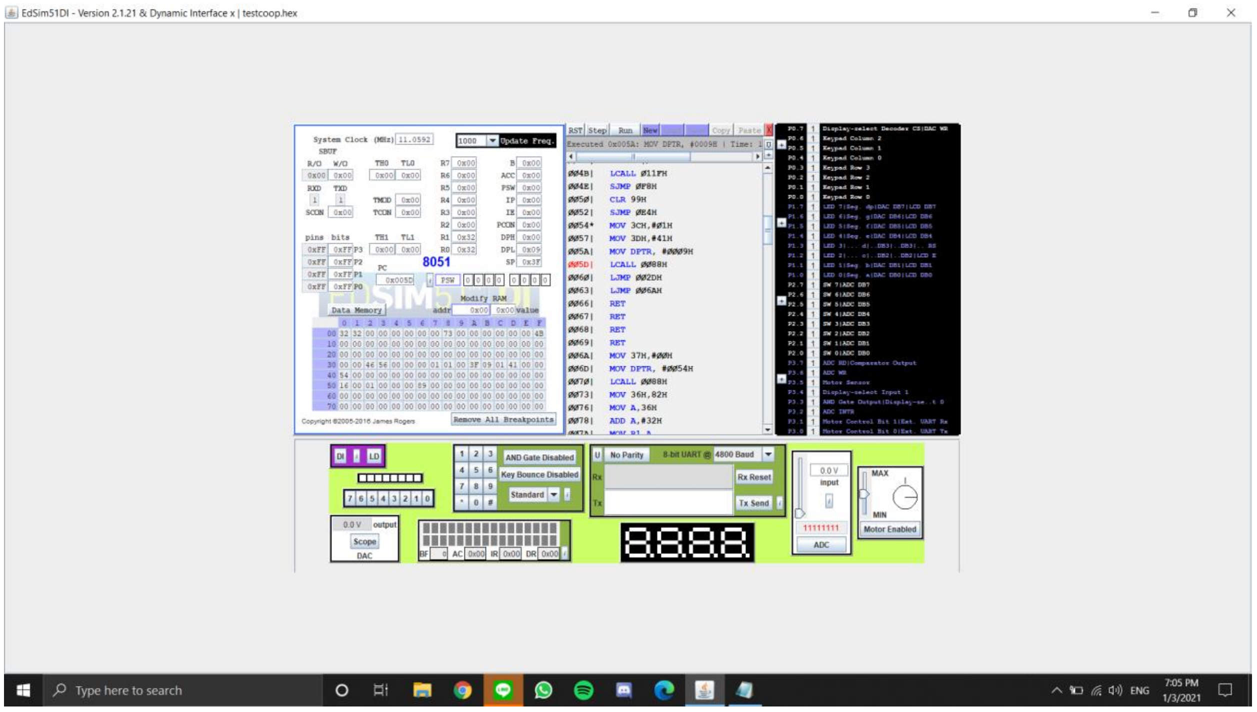


EdSim51DI - Version 2.1.21 & Dynamic Interface x | testcoop.hex

```
System Clock (MHz) 11.0592      1000 Update Freq.
R7 0x01  R6 0x00  R5 0x00  R4 0x00  R3 0x00  R2 0x00  R1 0x32  R0 0x32
B 0x00  ACC 0x00  PSW 0x00  IP 0x00  IE 0x00  PCON 0x00  DPH 0x00  DPL 0x54  SP 0x07
PC 0x0070

Data Memory
    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00 32 32 00 00 00 00 00 01 33 32 00 00 00 00 00 48
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 00 00 46 56 00 00 00 00 02 01 4F 41 48 49 00 00
40 4E 00 01 00 01 00 89 00 00 00 00 00 00 00 00 00
50 16 00 01 00 00 00 89 00 00 00 00 00 00 00 00 00
60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
Time: 9us - Instructions: 4
0068|  RET
0069|  RET
006A|  MOV 37H,#00H
006D|  MOV DPTR, #0054H
0070*  LCALL 0088H
0073|  MOV 36H,82H
0076|  MOV A,36H
0078|  ADD A,#32H
007A|  MOV R1,A
007B|  MOV 81H,@R1
007D|  POP 0D0H
007F|  POP 83H
0081|  POP 82H
0083|  POP 0F0H
0085|  POP 0E0H
0087|  RET
0088|  MOV A,#0FH
008A|  CJNE A,37H,04H
008D|  MOV 82H,#0FFH
0090|  RET
0091|  MOV 38H,#01H
```

Before calling threadcreate(producer)



EdSim51DI - Version 2.1.21 & Dynamic Interface x | testcoop.hex

```
System Clock (MHz) 11.0592      1000 Update Freq.
R7 0x00  R6 0x00  R5 0x00  R4 0x00  R3 0x00  R2 0x00  R1 0x32  R0 0x32
B 0x00  ACC 0x00  PSW 0x00  IP 0x00  IE 0x00  PCON 0x00  DPH 0x00  DPL 0x09  SP 0x3F
PC 0x005D

Data Memory
    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00 32 32 00 00 00 00 00 00 73 00 00 00 00 00 00 4B
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 00 00 46 56 00 00 00 01 01 00 3F 09 01 41 00 00
40 54 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50 16 00 01 00 00 00 89 00 00 00 00 00 00 00 00 00
60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

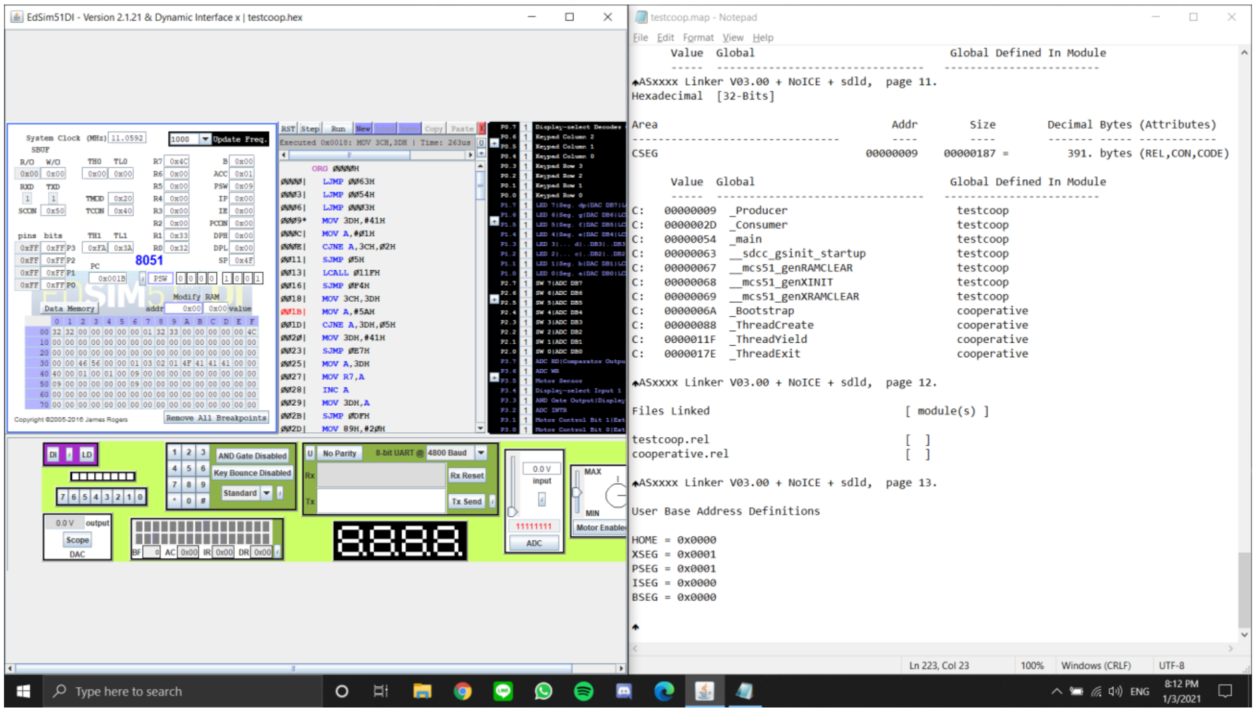
```
Executed 0x005A: MOV DPTR, #0009H | Time: 1
004B|  LCALL 011FH
004E|  SJMP 0F8H
0050|  CLR 99H
0052|  SJMP 0E4H
0054*  MOV 3CH,#01H
0057|  MOV 3DH,#41H
005A|  MOV DPTR, #0009H
005D|  LCALL 0088H
0060|  LJMP 002DH
0063|  LJMP 006AH
0066|  RET
0067|  RET
0068|  RET
0069|  RET
006A|  MOV 37H,#00H
006D|  MOV DPTR, #0054H
0070|  LCALL 0088H
0073|  MOV 36H,82H
0076|  MOV A,36H
0078|  ADD A,#32H
```

For the stack changes,

when it is running the threadcreate for main, there is no threads created yet. Therefore, thread ID will be set to 0. The address for thread ID 0 is 0x40. Therefore, when pushing DPL, DPH, ACC 4 times and PSW, they all will be pushed to the memory starting at 0x40. This can be seen from the edsim51. After the command reached the line "PUSH 82H" which means PUSH DPL, the data in memory 0x40 that was previously filled with the data 4E turns into 54. When push ACC 4 times is carried out, the data in 0x42 to 0x46 all changed to 00 as ACC is filled with 00. When pushing PSW, the data in 0x46 that was previously filled with 89 now changed to 00.

When it is running the threadcreate for producer, now thread create will take the ID of 1. Thread ID 1 will take the memory starting at 0x50. This means that pushing DPL, DPH, ACC 4 times and PSW will now fill the memory starting 0x50. The rest of the procedures are the same as the previous one.

2. Producer is running



```
Executed 0x0018: MOV 3CH,3DH | Time: 263us

        ORG 0000H
0000|   LJMP 0063H
0003|   LJMP 0054H
0006|   LJMP 0003H
0009*   MOV 3DH,#41H
000C|   MOV A,#01H
000E|   CJNE A,3CH,02H
0011|   SJMP 05H
0013|   LCALL 011FH
0016|   SJMP 0F4H
0018|   MOV 3CH,3DH
001B|   MOV A,#5AH
001D|   CJNE A,3DH,05H
0020|   MOV 3DH,#41H
0023|   SJMP 0E7H
0025|   MOV A,3DH
0027|   MOV R7,A
0028|   INC A
0029|   MOV 3DH,A
002B|   SJMP 0DFH
002D|   MOV 89H,#20H
```

```
PC 0x001B   SP 0x4F   DPH 0x00   DPL 0x00   R0 0x32   R1 0x33   PSW 0000 1001
```

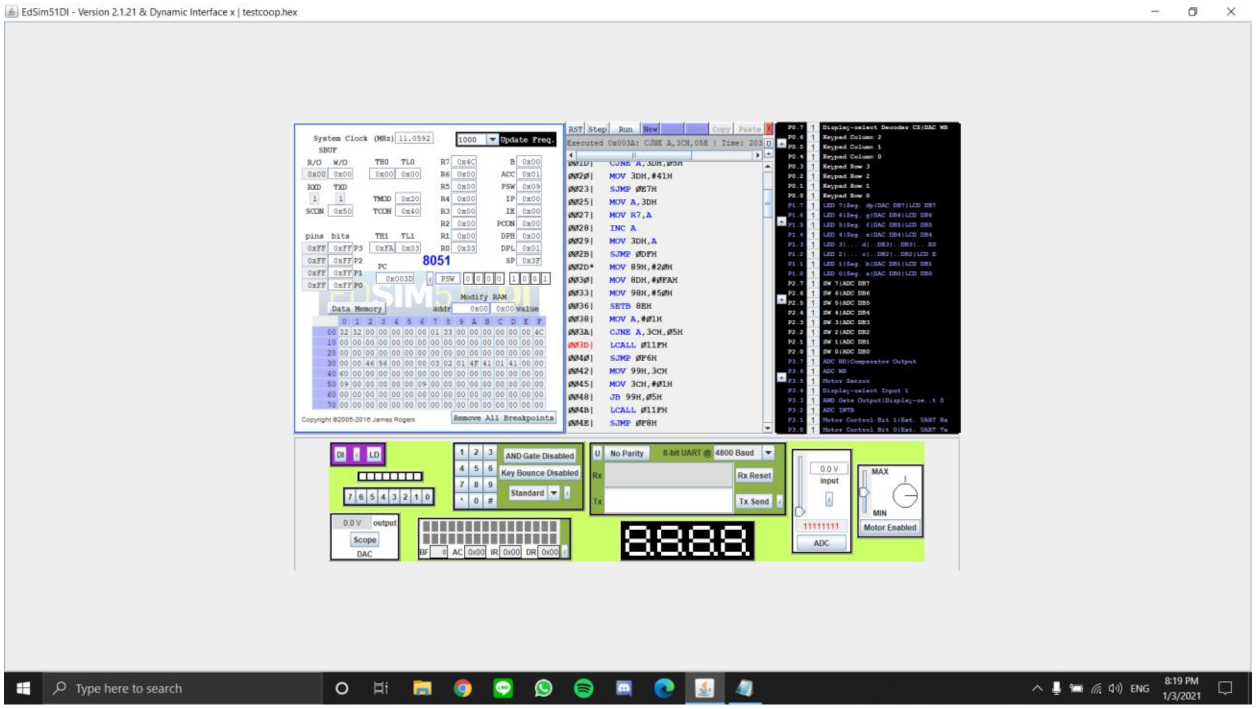
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	32	32	00	00	00	00	00	01	32	33	00	00	00	00	00	4C
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	46	56	00	00	01	03	02	01	4F	41	41	41	00	00
40	40	00	01	00	01	00	09	00	00	00	00	00	00	00	00	00
50	09	00	00	00	00	00	09	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

```
     Value  Global                              Global Defined In Module
     -----  --------------------------------   ------------------------
ASxxxx Linker V03.00 + NoICE + sdld,  page 11.
Hexadecimal  [32-Bits]

Area                                    Addr        Size        Decimal Bytes (Attributes)
--------------------------------        ----        ----        ------- ----- ------------
CSEG                                00000009    00000187 =        391. bytes (REL,CON,CODE)

     Value  Global                              Global Defined In Module
     -----  --------------------------------   ------------------------
C:   00000009  _Producer                          testcoop
C:   0000002D  _Consumer                          testcoop
C:   00000054  _main                              testcoop
C:   00000063  __sdcc_gsinit_startup              testcoop
C:   00000067  __mcs51_genRAMCLEAR                testcoop
C:   00000068  __mcs51_genXINIT                   testcoop
C:   00000069  __mcs51_genXRAMCLEAR               testcoop
C:   0000006A  _Bootstrap                         cooperative
C:   00000088  _ThreadCreate                      cooperative
C:   0000011F  _ThreadYield                       cooperative
C:   0000017E  _ThreadExit                        cooperative

ASxxxx Linker V03.00 + NoICE + sdld,  page 12.

Files Linked                         [ module(s) ]

testcoop.rel                         [   ]
cooperative.rel                      [   ]

ASxxxx Linker V03.00 + NoICE + sdld,  page 13.

User Base Address Definitions

HOME = 0x0000
XSEG = 0x0001
PSEG = 0x0001
ISEG = 0x0000
BSEG = 0x0000
```

I know that producer is running since, after the command MOV 3CH 3DH, which means that buffer takes the value in token, the value in 3CH that was previously different from 3DH, now have the same value, which indicates that buffer takes the value in token, which means that producer is running.

3. Consumer is running

107062381 楊孝偉


EdSim51DI - Version 2.1.21 & Dynamic Interface x | testcoop.hex

```
Executed 0x003A: CJNE A,3CH,05H | Time: 203
001D|   CJNE A,3DH,05H
0020|   MOV 3DH,#41H
0023|   SJMP 0E7H
0025|   MOV A,3DH
0027|   MOV R7,A
0028|   INC A
0029|   MOV 3DH,A
002B|   SJMP 0DFH
002D*   MOV 89H,#20H
0030|   MOV 8DH,#0FAH
0033|   MOV 98H,#50H
0036|   SETB 8EH
0038|   MOV A,#01H
003A|   CJNE A,3CH,05H
003D|   LCALL 011FH
0040|   SJMP 0F6H
0042|   MOV 99H,3CH
0045|   MOV 3CH,#01H
0048|   JB 99H,05H
004B|   LCALL 011FH
004E|   SJMP 0F8H
```

I know that consumer is running by the command CJNE A 3CH 05H, which means that it checks whether A, that has the value of 1 is the same as 3CH, which is buffer, if yes, then thread yield, which is represented by LCALL 011FH. These commands can only be found in consumer, and since they are carried out by the edsim51, it means that consumer is working.

4. Since the producer and consumer both do not call ThreadYield again, they rely on the preemptive mechanism to context switch. Preemptive mechanism use interrupt timer to context switch instead of yield. Therefore, the interrupt is triggering on a regular basis.