DIGITAL SYSTEMS HT2018

SHEET 3

GABRIEL MOISE

Question 1

(a) The relationship that is maintained between the value of bufin, bufout and bufcnt is (bufout + bufcnt) % NBUF = bufin, so we can replace the value of bufin in the program, by just using bufout and bufcnt:

```
void serial_putc(char ch) {
    intr_disable();
    if (txidle) {
        UART_TXD = ch;
        txidle = 0;
    } else {
        while (bufcnt == NBUF) {
            intr_enable();
            pause();
            intr_disable();
        }
        txbuf[(bufout+bufcnt)%NBUF] = ch; // txbuf[bufin] = ch;
        bufcnt++;
        //bufin = (bufin+1) % NBUF;
    }
    intr_enable();
}
```

(b) Let's suppose that at some point in the program, the txidle value is 0, meaning that the program has work to do. Now, let's suppose that we want to print a character 'a' so we call serial_putc('a'). The first thing to do is to check if txidle is 0 or not, in our case it is 0, so we will follow the else branch. However, after we decided to branch to "else", an interrupt occurs and txidle becomes 1, so in this case the character could have been printed. But, as we already chose the "else" branch, 'a' will be put in the buffer(assuming that the buffer is not full and we do not have to wait) and then another call to serial_putc will happen, for example serial_putc('b'), which we initially wanted to be printed after 'a'. This time, as txidle is 1, 'b' will be put on UART_TXD, so it will be printed before 'a', which is a bug. So, it is mandatory to disable interrupts in serial_putc before checking txidle.

(c) The Assembly-language equivalent for bufcnt ++ is :

```
ldr r0, =bufcnt
ldr r1, [r0]
INTERRUPT!!
add r1, r1, #1
str r1, [r0]
```

The interrupts must be disabled during this command as an interrupt might occur between the load and add instructions. In the uart_handler, if UART_TXDRDY is 1, then we will print the next character from the buffer and then decrease bufcnt by 1, so after the interrupt, when we will add 1 to bufcnt, we will basically leave it unchanged, therefore we will have a bug here.

(d) The difference between the WFE and WFI instructions is that the interrupt events will "wake up" from WFE only if they are not masked by the I, F, A bits (in our case the WFE suspends the execution until an interrupt occurs (one that is not masked as we have when we call intr_disable()), whereas the WFI suspends the execution until any kind of interrupt occurs, so disabling the interrupt does not have the same effect. However, from what I see in the program, we use the WFE instruction for the instruction pause() in serial_putc, between to instructions: intr_enable() and intr_disable(), therefore here no interrupts are masked, so I think that in this program the use of WFI wouldn't differ from the use of the WFE instruction.

(e) We can leave more space for interrupts in serial_putc this way:

```
void serial_putc(char ch) {
    while (bufcnt == NBUF) pause();
    intr_disable();
    if (txidle) {
        intr_enable();
        txidle = 0;
        UART_TXD = ch;
    } else {
        txbuf[(bufout+bufcnt)%NBUF] = ch;
        bufcnt++;
    }
    intr_enable();
}
```
This was safe because when we take a look at the uart_handler for interrupts:
```
void uart_handler(void) {
    if (UART_TXDRDY) {
        UART_TXDRDY = 0;
        if (bufcnt == 0)
            txidle = 1; // the important part !
        else {
            UART_TXD = txbuf[bufout];
            bufcnt --;
            bufout = (bufout+1) % NBUF;
        }
    }
}
```
we see that in order to get to that point of the program we needed txidle = 1 in the serial_putc, so we needed to have UART_TXDRDY = 1 to enter the if condition, then UART_TXDRDY became 0, then in order for us to have txidle = 1, bufcnt needed to be 0, so we had nothing in the buffer, and after that txidle became 1. Coming back to the serial_putc part, it is now safe to set txidle to 0 because if an interrupt would happen during this time, nothing will happen since we established that UART_TXDRDY is 0, then we can add the character ch to UART_TXD, and it's important here that only after this instruction, the UART_TXDRDY becomes 1 (as we are ready to have another character printed), so we might have problems with the interrupts if we had other functions to follow(that's the main reason why I swapped the txidle = 0 and UART_TXD = ch instructions, the effect remains the same, but now we are safe to disable the interrupts.
Furthermore, we cannot enable the interrupts for the else branch since the first instruction requires a calculation of bufout and bufcnt and they can be both modified here since the buffer is non-empty (idle is 0 since UART_TXDRDY is also 0) and the other instruction was discussed at (c).

Question 2

To make the beating pattern of the heart, we change the advance function, which will use a row variable that will take values in [0..300), with the meaning that the first 70% of the time we have a heart, the next 10% we have a small heart, the next 10% again a heart and the last 10% a small heart, the same idea that we used for the beating heart for the second lab. Therefore, we will have:

```
void advance(void) {

    row++;

    if (row < 210) GPIO_OUT = heart[row%3];

    else if (row < 240) GPIO_OUT = small[row%3];

    else if (row < 270) GPIO_OUT = heart[row%3];

    else if (row < 300) GPIO_OUT = small[row%3];

    else if (row == 300) row = 0;

}
```

I scaled the problem by a factor of 3 because each 1% of the total time is used to print 3 rows, not just one.

The limitations of the program are the beating pattern which is fixed no matter what the images we want to display are. However, we can change the values we want to use for a different pattern in the same manner.

Question 3

The pieces of the program are:

- the initialization of the random generator

```
void rng_init(void){

  RNG_STOP = 1;

  RNG_INTENSET = 1;

  RNG_START = 1;

  enable_irq(RNG_IRQ);

}
```

Gabriel : Can you please explain me what each of them does? Because I played with them for a long time to get them right and I can say that it works just because I got lucky to find the right configuration for rng_init.

- the array that stores the 8-bit random values that we get in RNG_VALUE and the current number of elements that are in (I bounded nr by 1000 because in the eventuality that we produce random numbers faster than we use them, I no longer store them, because I already have enough of them)

  ```
  static volatile unsigned randArray [1000];
  static unsigned nr;
  ```

- the rng_handler function that acts when an interrupt occurs

```
void rng_handler(void)

{

  if (RNG_VALRDY == 1)

  {

    randArray[nr%1000] = RNG_VALUE;

    nr = (nr + 1)%1000;

    RNG_VALRDY = 0;

  }

}
```

- the function randint that returns a random 32-bit value (so we need to combine 4 values from randArray)

```
unsigned randint(void){

  if(nr>=4)

           return ((randArray[nr-1] << 24) + (randArray[nr-2] << 16) + (randArray[nr-3] << 8) +
randArray[nr-4]); // here we combine the values we get if we have at least 4 values

  else while (nr<4) pause(); // we wait until we get more values from interrupts otherwise

  return 0; //this never happens, but it gets rid of a warning from the compiler that the function
might not return anything

}
```

Some trials for randint: // The values have to be from 0 to 2^{32}-1 = 4,294,967,295

Random(0) = 3962082884

Random(1) = 3716654257

Random(2) = 320334799

Random(3) = 1251360825

Random(4) = 3802712943

Random(5) = 3662876932

Random(6) = 4003713207

Random(7) = 1087499553

Random(8) = 3069500113

Random(9) = 1827783082

Random(10) = 125721313

Random(11) = 1072709487

Random(12) = 3556964786

Random(13) = 1018317541

Random(14) = 4156537490

Random(15) = 530776140

Random(16) = 2267596100

Random(17) = 3862887198

Random(18) = 3279716902

Random(19) = 252426720

- the function roll that returns a random value from 1 to 6 (we basically use a random value and take it mod 6). However, since the random numbers are uniformly distributed on [0..2^{32}-1], where 2^{32}-1 is a multiple of 6 plus 3, then 6(0), 1, 2 and 3 will be more likely to appear than 4 and 5 and we don't want this to happen. Therefore, we will say that when we get a 0, 1, 2 or 3, the process should repeat in order to get another value different from those. This way, we get a uniform distribution for 1 .. 6.

```
unsigned roll(void)
{
  unsigned check;
  check = randint() ;
  if (check>=4){
  check = check % 6;
  if (check == 0) check = 6;
  return check;
}
  else return roll(); // So, we get check less than 4, we do this again.
}
```

Some trials for roll:

Roll: 6

Roll: 5

Roll: 2

Roll: 6

Roll: 5

Roll: 5

Roll: 5

Roll: 4

Roll: 4

Roll: 5

Roll: 1

Roll: 2

Roll: 2

Roll: 3

```
Roll: 2

Roll: 4

Roll: 2

Roll: 3
```

Now, let’s write a program where we roll the die count times and we print how many times we got each value from 1 to 6. (I substituted the function roll here to have everything in the while).

```
static volatile unsigned die [6];

void init(void) {

    int count = 0;

    nr = 0;

    serial_init();

    rng_init();

    start_timer();

    while (count<900)

    {

      if (nr>4){

        unsigned value;

        value = randint(); // equivalent of check from roll();

        if (value >=4) // Same reasoning as for roll();

             {

                    value = value % 6;

                    die[value] ++; // the vector that counts the number of appearances for 1..6

                    nr = nr - 4;

                    count++;

             }

      }

      else pause();

    }

    int i;

    for (i=1;i<=5;i++) serial_printf("%d was rolled %u times\r\n",i ,die[i]);

    serial_printf("6 was rolled %u times\r\n",die[0]);

    stop_timer();

}
```

RESULT 1: // count = 900

```
1 was rolled 142 times

2 was rolled 132 times

3 was rolled 157 times

4 was rolled 161 times
```

5 was rolled 172 times

6 was rolled 136 times

RESULT 2: //count = 6000

1 was rolled 1040 times

2 was rolled 999 times

3 was rolled 937 times

4 was rolled 954 times

5 was rolled 1067 times

6 was rolled 1003 times

RESULT 3 :// count = 60000

1 was rolled 10426 times

2 was rolled 9674 times

3 was rolled 10467 times

4 was rolled 9497 times

5 was rolled 10411 times

6 was rolled 9525 times

So, we can see that the distribution is close to even, and this is expected as the numbers we choose are random and the interval contains an equal number of multiples of 6, of multiples of 6 plus 1 etc., so the probability of obtaining any value is the same, 1/6 in our case.

Question 4

Since we are told that the cut-down version of the Cortex-M0 only saves the program counter (PC) and the processor status register (PSR) to the stack before invoking the interrupt and since we need to save not only them, but also the values of the registers r0,r1,r2,r3 (and r12?) and the lr, we will do it manually. (Also, the values of registers r4-r7 (r8-r11) are automatically saved by the subroutine that is called (in our case the interrupt handler)). So, we will have:

```
...

push (r0-r3, lr) @ In the lecture we have r12, is it necessary too?

bl function @ execute the body of the function

pop (r0-r3, lr)

rti @ returning from the interrupt

...
```