Tuan Nguen

**Question 1:**

a) The invariant that is maintained between the variables bufin, bufout and bufcnt is that if bufin < bufout, means that the array is wrapping around, so bufcnt = bufin + NBUF – bufout. Otherwise, bufcnt = bufin – bufout. We can use this to eliminate one of the variables. Calculate bufcnt + bufout and if it is larger than NBUF, we decrement the sum by NBUF. Otherwise, that is bufin.

void uart\_handler(void) {

if(UART\_TXDRDY) {

UART\_TXDRDY = 0;

if(bufcnt == 0)

txidle = 1;

else {

UART\_TXD = txbuf[bufout];

bufcnt--;

bufout = (bufout+1) % NBUF;

}

}

}

/\* serial\_putc – send output character \*/

void serial\_putc(char ch) {

while(bufcnt == NBUF) pause();

intr\_disable();

if(txidle) {

UART\_TXD = ch;

txidle = 0;

}

else {

if(bufout+bufcnt > NBUF){

txbuf[bufout+bufcnt-NBUF] = ch;

bufcnt++;

}

}

intr\_enable();

}

b) We want to disable the interrupts before checking txidle because there could be a situation when the variable is set to 0, and we want it to remain that way, but if the handler is called, the uart\_handler can change it back to 1.

c) The bufcnt++ instruction can be implemented in assembly like this:

ldr r0, =bufcnt

ldr r1, [r0]

add r1, r1, #1

str r1, [r0]

If an interrupt happens between the second and the third line, then the effect of bufcnt--will be lost, because when the interrupt handler returns, it is an old value that is incremented.

d) The wfi command cannot see interrupts that have happened before it, whereas the wfe command can. So if an interrupt happens during the comparison between the bufcnt and NBUF, then the program would crash.

e)

/\* serial\_putc – send output character \*/

void serial\_putc(char ch) {

while(bufcnt == NBUF) pause();

intr\_disable();

if(txidle) {

intr\_enable();

UART\_TXD = ch;

txidle = 0;

}

else {

if(bufout+bufcnt > NBUF){

txbuf[bufin] = ch;

intr\_disable();

bufcnt++;

intr\_enable();

bufin = (bufin + 1) % NBUF;

}

}

}

**Question 2:**

static const unsigned small[] = {

0x2df0, 0x5fb0, 0x8af0

};

static unsigned row = 0;

void heartBigDelay(void) {

unsigned n = 1050000

while(n-- > 0){

for(int p = 0; p < 3; p++){

GPIO\_OUT = heart[p]

}

}

}

void heartSmallDelay(void) {

unsigned n = 150000

while(n-- > 0){

for(int p = 0; p < 3; p++){

GPIO\_OUT = heart[p]

}

}

}

void smallDelay(void) {

unsigned n = 150000

while(n-- > 0){

for(int p = 0; p < 3; p++){

GPIO\_OUT = small[p]

}

}

}

void timer1\_handler(void) {

if (TIMER1\_COMPARE[0]) {

heartBigDelay();

smallDelay();

heartSmallDelay();

smallDelay();

TIMER1\_COMPARE[0] = 0;

}

}

void init\_timer(void) {

TIMER1\_STOP = 1;

TIMER1\_MODE = TIMER\_Timer\_Mode;

TIMER1\_BITMODE = TIMER\_16Bit;

TIMER1\_PRESCALER = 4; // 1MHz = 16MHz / 2^4

TIMER1\_CLEAR = 1;

TIMER1\_CC[0] = 1000 \* TICK;

TIMER1\_SHORTS = BIT(TIMER\_COMPARE0\_CLEAR);

TIMER1\_INTENSET = BIT(TIMER\_INTEN\_COMPARE0);

TIMER1\_START = 1;

set\_priority(TIMER1\_IRQ, 3);

enable\_irq(TIMER1\_IRQ);

}

void init(void) {

GPIO\_DIR = 0xfff0;

GPIO\_PINCNF[BUTTON\_A] = 0;

GPIO\_PINCNF[BUTTON\_B] = 0;

GPIO\_OUT = heart[0];

init\_timer();

while (1) {

pause();

}

}

The drawbacks of the way that my program is designed(I guess there is less naïve way to do it) is that for every „picture” has to be written in its own separate function, and if we want several “pictures” to flash this way would be infeasible. The major drawback is that there is only one main program, so only one task can be executed at a time, and the other tasks have to be called at appropriate times, with their entire state stored in variables.

**Question 3:**

\* I will explain my idea of what the program should be \*

The program will be similar to the one we have seen for the interrupt based prime number program. In the rng\_handler will be checked why an interrupt has occurred, so we want to check if RNG\_VALRDY is 1. If it is, then we test if the buffer is empty. If it is, then we get the number that has been generated and pass it to randint(), where we will take 4 random numbers, pad left each of them and then put the next one to the right of the previous one. If the buffer is not empty, get a number from the buffer and pass it to randint(). Decrease bufcnt and increase bufout. In the randint() function we check if the buffer is full, and if it is, then we pause, until at least one place in the buffer is freed. Disable interrupts, check if the program is idling, and depending on that 1) we put a randomly generated number into our number that we have to return; 2) fetch a number from the buffer and put that into our number that we have to return. Enable interrupts. For roll() the procedure would be the same, but instead of appending 4 8-bit numbers, we can calculate mod6 of the decimal number, corresponding to the 8-bit binary.

**Question 5:**

If the stack grew upwards, instead of downwards, the program would still be vulnerable. If there is a register rd that points to the start of the buffer, and there is an operation that involves rd, which there probably will be, we can write our code on the buffer. Then use that same register to return to our code and then execute it.