Question 1

(a) What relationship is maintained among the values of `bufin`, `bufout` and `bufcnt`? Try to find an alternative implementation with only two integer variables.

`bufin` is the index where the next byte will be placed in the buffer. `bufout` is the index where the next byte will be read from the buffer. `bufcnt` is the number of bytes currently in the buffer. The variables maintain the start and the end of a circular buffer and the count of elements in it. Specifically, `bufcnt` tracks how full the buffer is, allowing the code to distinguish between a full and empty buffer. Given that its a circular buffer, `bufin` and `bufout` increment $\mod N$ where N is the size of the buffer.

b) Why is it necessary in `serial_putc` to disable interrupts before checking `txidle`?

Interrupts must be disabled before checking `txidle` to prevent race conditions. Specifically, if an interrupt occurred right after checking `txidle` but before setting it inside `serial_putc`, the UART handler could see `txidle` as true and decide not to send a character, even though `serial_putc` was about to do so. To make a comparison, this is somewhat similar to making an atomic transaction to a relational database in order to maintain consistency.

(c) Why must interrupts be disabled during the command `bufcnt++`?

Disabling interrupts during `buffcnt++` is necessary because the increment operation is not atomic in the ARM Cortex-M, meaning that cutting it short can cause consistency issues (such as the counter being incremented but not stored). Disabling interrupts ensures that this operation completes without interference.

(d) Study the difference between the `wfe` and `wfi` instructions, and explain why `wfe` is needed in this program.

- `wfi` - Wait for interrupt. This instruction halts the CPU until any interrupt occurs.
- `wfe` - Wait for event. This instruction halts the CPU until any event occurs.

Here the difference lies in what event is defined as. Since events can be more than just interrupts, the scope of the `wfe` instruction - specifically defined for cases with a multi-core or multi-threaded system, covering cases where some cores can remain functional while some other specified cores or threads are halted.

(e) If it was important to have interrupts disabled for the shortest possible time, how could the code of `serial_putc` be modified so as to remain safe?

To minimise the time interrupts are disabled and keep the code safe, we could restructure `serial_putc` to disable interrupts only around the critical sections where shared variables are modified:

```
void serial_putc(char ch) {
    while (bufcnt == NBUF) pause(); // Wait if buffer is full

    intr_disable();
    int localTxIdle = txidle; // Copy txidle status with interrupts disabled
    if (localTxIdle) {
        UART_TXD = ch;
        txidle = 0;
        intr_enable(); // Re-enable interrupts as soon as possible
    } else {
        intr_enable(); // Re-enable interrupts before working with buffer
        txbuf[bufin] = ch;
        intr_disable(); // Disable again before modifying shared variables
        bufcnt++;
        bufin = (bufin+1) % NBUF;
        intr_enable(); // Re-enable interrupts after modification
    }
}
```

Question 2

To enhance the `heart-intr` program to show an animated heartbeat using the timer interrupt handler, you can modify the interrupt handler to cycle through a series of frames that represent the heartbeat animation. We can do so by defining some animation frames, and then modify the interrupt handler to change the frames to give the illusion that the static frames of the heart are actually moving.

Not entirely sure how to do this, but here is a C function that could be used for this objective:

```
const int NUM_FRAMES = ...;
const int FRAMES[NUM_FRAMES] = { ... }; // Animation frames
int currentFrame = 0;

void timerInterruptHandler() {
    displayFrame(FRAMES[currentFrame]);
    currentFrame = (currentFrame + 1) % NUM_FRAMES;
    ...
}
```

Unfortunately, we're limited by the interrupt frequency. The smoothness and speed of the animation are directly limited by how frequently the timer interrupt can fire. Too fast may not be visually distinguishable, and too slow would not look smooth.

Feeding in the animation as static frames may also be inefficient, since we only need one frame and then only feed in the difference of each frames (a complement) of the previous frame, which could be done with an exclusive-or operation.

Question 3

For this question, we will need to implement a buffering scheme to store random bytes as they are generated, and provide the `randint()` and `roll()` functions to utilise these bytes. First, we need to establish the interrupt handler:

```
volatile unsigned char rngBuffer[RNG_BUFFER_SIZE];
volatile int rngBufferWriteIndex = 0;
volatile int rngBufferReadIndex = 0;
volatile int rngBufferCount = 0;

void rng_handler(void) {
    if (RNG_VALRDY) {
        rngBuffer[rngBufferWriteIndex] = RNG_VALUE;
        rngBufferWriteIndex = (rngBufferWriteIndex + 1) % RNG_BUFFER_SIZE;
        rngBufferCount++;
        RNG_VALRDY = 0; // Reset the event flag
    }
}
```

With that, the `randint(void)` and the `roll(void)` would be implemented as such:

```
unsigned randint(void) {
    unsigned value = 0;
    for (int i = 0; i < 4; i++) {
        while (rngBufferCount == 0);
        unsigned char byte = rngBuffer[rngBufferReadIndex];
        rngBufferReadIndex = (rngBufferReadIndex + 1) % RNG_BUFFER_SIZE;
        __disable_irq();
        rngBufferCount--;
        __enable_irq();
        value |= ((unsigned)byte << (i * 8));
    }
    return value;
}
```

```
unsigned roll(void) {
    unsigned char byte;
    unsigned value;
    do {
        while (rngBufferCount == 0);
        byte = rngBuffer[rngBufferReadIndex];
        rngBufferReadIndex = (rngBufferReadIndex + 1) % RNG_BUFFER_SIZE;
        __disable_irq();
        rngBufferCount--;
        __enable_irq();
        value = (unsigned)byte % 6 + 1;
    } while (value > 6);
    return value;
}
```

Question 4

Essentially, the adapter would need to do the following:
Upon entering the interrupt, save the registers that the C function might alter onto to stack. Then, we call the C interrupt handler function, then after it returns a value, restore the registers from the stack.

We may assume that `r0-r3` are the currently used registers, which means that our assembly program would look like the following:

```
.global uart_interrupt_adapter
.type uart_interrupt_adapter, %function

uart_interrupt_adapter:
    push {r0-r3, r12, lr}    @ Save registers that uart_handler might alter
    bl uart_handler          @ Call the C function interrupt handler
    pop {r0-r3, r12, lr}     @ Restore registers
    rti                      @ Return from interrupt using special instruction
```

Question 5

To answer this question, we need to consider the difference between an upward and a downward growing stack.

- Downward-Growing Stack: Local variables such as arrays are usually stored at lower memory addresses than the function's return address. Anyone can more easily access the function's address and overwrite that.

- Upward-Growing Stack: If the stack were to grow upwards, control data would be stored below the actual data. This might make it harder to overflow to the important control data.

At first glance, this does seem like the better alternative, but one counter example of the upward-growing stack is sufficient to disprove this observation. I'm not sure what that counter example is though. (Stumped)