Digital Systems: Problem sheet 3

Mike Spivey, Hilary Term, 2019

1 Figure 1 shows the interrupt-based driver for serial output that we studied in the lecture. The functions `intr_disable()` and `intr_enable()` use the instructions `cpsid i` and `cpsie i` to change the `PRIMASK` bit in the CPU that allows interrupts. The function `pause()` uses the `wfe` instruction to halt the CPU until an interrupt occurs.

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

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

(c) Why must interrupts be disabled during the command `bufcnt++`? *Hint: consider the assembly-language equivalent of the command.*

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

(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?

2 Program `heart-intr` embeds all the code needed to multiplex the LED display in the handler for the timer interrupt, allowing the 'main program' to be devoted to other tasks – but it can display only a static image. Show how to enhance it to show an animated heartbeat, like the one in Lab 2. What are the limitations of this approach?

3 The NRF51822 has a hardware random number generator. When appropriately configured, it periodically generates an interrupt with handler `rng_handler`, and an eight-bit random number can then be retrieved from the device register `RNG_VALUE` before resetting the event flag `RNG_VALRDY` to zero. Design an interrupt-based driver for the random number generator; provide two functions

```
unsigned randint(void);
unsigned roll(void);
```

```
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 {
        txbuf[bufin] = ch; bufcnt++;
        bufin = (bufin+1) % NBUF;
    }
    intr_enable();
}
```

Figure 1: *Code for interrupt-driven serial output*

such that `randint()` returns a random four-byte value each time it is called, and `roll()` returns a random integer between 1 and 6, with each outcome having precisely equal probability. Arrange a suitable buffering scheme so that the caller of these functions does not have to wait if it calls them infrequently enough, and introduce a subroutine whenever doing so avoids repeating the same code in more than one place. What should happen if random bytes are generated by the hardware faster than the program is consuming them?

4 A cut-down version of the Cortex-M0 saves only the program counter and the processor status register to the stack before invoking the interrupt handler. It does not set `lr` to a magic value, but leaves it unchanged, and returning from the interrupt requires a special instruction `rti` (with encoding 0xbfd0). Design an assembly-language adapter that allows a C function such as `uart_handler` in Exercise 1 to be installed as an interrupt handler.

5 In the buffer overrun attack of Lecture 7, a long input string was able to overflow the buffer in the frame of `getnum()` and replace its return address. This worked because the buffer was stored at a lower address than the register save area in `getnum`'s stack frame. Would buffer overrun attacks be prevented if the stack were to grow upwards in memory instead of downwards?