Chapter 10 Input and Output

1. Similar to Example 10.6, consider a C program for an Atmel AVR that uses a UART to send 8 bytes to an RS-232 serial interface, as follows:

```
for(i = 0; i < 8; i++) {
  while(!(UCSR0A & 0x20));
  UDR0 = x[i];
}
```

Assume the processor runs at 50 MHz; also assume that initially the UART is idle, so when the code begins executing, `UCSR0A & 0x20 == 0x20` is true; further, assume that the serial port is operating at 19,200 baud. How many cycles are required to execute the above code? You may assume that the `for` statement executes in three cycles (one to increment `i`, one to compare it to 8, and one to perform the conditional branch); the `while` statement executes in 2 cycles (one to compute `!(UCSR0A & 0x20)` and one to perform the conditional branch); and the assigment to `UDR0` executes in one cycle.

Solution: The first pass through the `for` loop takes $3+2+1=6$ cycles. The second takes $2+2n+1$, where n is the number of times the `while` statement executes. After each write to `UDR0`, the UART needs to send 8 bits before `!(UCSR0A & 0x20)` will become true. At 19,200 baud, it takes $8/19,200$ seconds to do this, which is $50,000,000 \times 8/19,200 = 20,834$ cycles (rounding up). Thus, n must be large enough so that $3+2(n-1) >= 20,834$ (this is $n-1$ not n because the last execution of the `while` occurs after `!(UCSR0A & 0x20)` becomes false. The smallest integer value of n satisfying this is $10,417$. Hence, the second pass through the `for` loop takes $2+2n+1 = 20,837$. The remaining 6 passes through the `for` loop take the same amount of time, so the total time is $6+7\times 20,837 = 145,865$.

However, RS232 requires a start bit and at least one stop bit, so the UART needs to send 10 bits rather than 8. This makes the time $10/19,200$ sec, giving $26,042$ cycles.

In practice, these numbers are approximate because the architecture may introduce variability in the timing. For example, instructions may not be in cache, and the cache penalty could be substantial. Moreover, writing to `UDR0` and reading `UCSR0A` may involve transactions over the processor bus, and there may be other activities competing for the bus (e.g., other I/O activities or DMA). Hence, this number should be interpreted as a lower bound.

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;

// Interrupt service routine.
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_count > 0) {
    timer_count--;                    A
  }
}

// Main program.
int main(void) {
  // Set up interrupts to occur
  // once per second.
  ...
  // Start a 3 second timer.          B
  timer_count = 3;

  // Do something repeatedly
   // for 3 seconds.
  while(timer_count > 0) {
    foo();                            C
  }
}
```

Figure 10.1: Sketch of a C program that performs some function by calling procedure foo() repeatedly for 3 seconds, using a timer interrupt to determine when to stop.

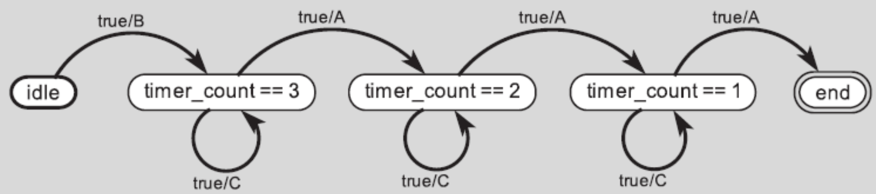
2. Figure 10.1 gives the sketch of a program for an Atmel AVR microcontroller that performs some function repeatedly for three seconds. The function is invoked by calling the procedure foo(). The program begins by setting up a timer interrupt to occur once per second (the code to do this setup is not shown). Each time the interrupt occurs, the specified interrupt service routine is called. That routine decrements a counter until the counter reaches zero. The main() procedure initializes the counter with value 3 and then invokes foo() until the counter reaches zero.

   (a) We wish to assume that the segments of code in the grey boxes, labeled **A**, **B**, and **C**, are atomic. State conditions that make this assumption valid.

   Solution: Assumptions needed for atomicity: (1) Interrupts are disabled while the interrupt service routine is executing. (2) The method foo() does not read or write the variable timer_count. Note that the invocation of the method foo() is most likely not atomic (unless it also disables interrupts, which would be an equally valid assumption). The interrupt could occur during the execution of foo(). It is not constrained to occur between executions of foo(). However, if foo() does not read or write timer_count, then it will behave as if it were atomic.

   (b) Construct a state machine model for this program, assuming as in part (a) that **A**, **B**, and **C**, are atomic. The transitions in your state machine should be labeled with "guard/action", where the action can be any of **A**, **B**, **C**, or nothing. The actions **A**, **B**, or **C** should correspond to the sections of code in the grey boxes with the corresponding labels. You may assume these actions are atomic.

Solution: One possible model for this program is shown below:



The state with the bold outline is the initial state and the one with a double outline is the final state. The intermediate states represent the three possible values of the timer_count variable. Each of these states has two transitions out of it, both guarded by the expression "true."

(c) Is your state machine deterministic? What does it tell you about how many times foo() may be invoked? Do all the possible behaviors of your model correspond to what the programmer likely intended?

Solution: The state machine is nondeterministic. The state machine shows that this program could result in any number of invocations of foo(), including zero. Zero executions of foo() is almost certainly a behavior that the programmer did not intend.

Note that there are many possible answers. Simple models are preferred over elaborate ones, and complete ones (where everything is defined) over incomplete ones. Feel free to give more than one model.

3. In a manner similar to example 10.8, create a C program for the ARM Cortex™ - M3 to use the SysTick timer to invoke a system-clock ISR with a jiffy interval of 10 ms that records the time since system start in a 32-bit int. How long can this program run before your clock overflows?

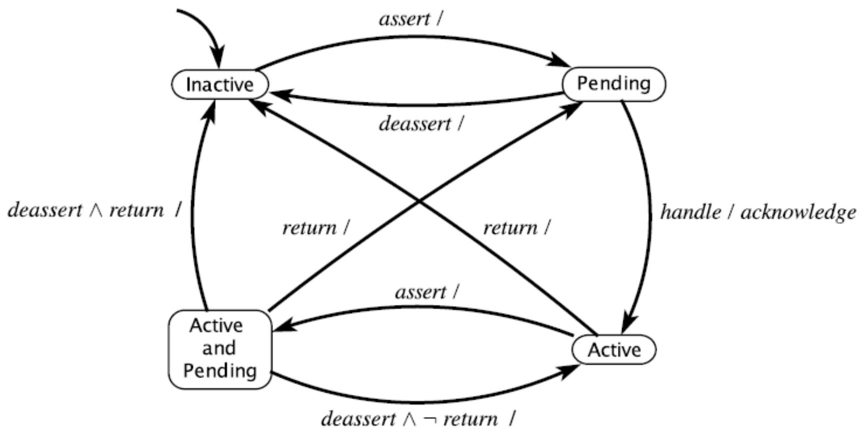
Solution:

```
volatile int32_t timerCount = 0;
void countSince() {
    timerCount++;
}
void main() {
    SysTickPeriodSet(SysCtlClockGet()/100);
    SysTickIntRegister(&countSince);
    SysTickEnable();
    SysTickIntEnable();
    ...
}
```

The largest positive number that fits in an `int32_t` data type is $2^{31} - 1 = 2{,}147{,}483{,}647$. Multiplying this by 10 ms indicates that the clock will overflow after about 249 days.

5. Suppose a processor handles interrupts as specified by the following FSM:

input: *assert*, *deassert*, *handle*, *return*: pure
output: *acknowledge*



Here, we assume a more complicated interrupt controller than that considered in Example 10.12, where there are several possible interrupts and an arbiter that decides which interrupt to service. The above state machine shows the state of one interrupt. When the interrupt is asserted, the FSM transitions to the Pending state, and remains there until the arbiter provides a *handle* input. At that time, the FSM transitions to the Active state and produces an *acknowledge* output. If another interrupt is asserted while in the Active state, then it transitions to Active and Pending. When the ISR returns, the input *return* causes a transition to either Inactive or Pending, depending on the starting point. The *deassert* input allows external hardware to cancel an interrupt request before it gets serviced.

Answer the following questions.

(a) If the state is Pending and the input is *return*, what is the reaction?

Solution: The FSM remains in state Pending.

(b) If the state is Active and the input is $assert \wedge deassert$, what is the reaction?

Solution: The machine moves to Active and Pending.

(c) Suppose the state is Inactive and the input sequence in three successive reactions is:

i. *assert* ,
ii. $deassert \wedge handle$,
iii. *return* .

What are all the possible states after reacting to these inputs? Was the interrupt handled or not?

Solution: The only possible state is Inactive. The ISR may or may not have executed. The fact that the input sequence includes *return* suggests that it was, but if the instruction set permits an erroneous program to issue a “return from interrupt” instruction even if not in an ISR, then the ISR may not have executed. We do not have enough information to be sure.

(d) Suppose that an input sequence never includes *deassert*. Is it true that every *assert* input causes an *acknowledge* output? In other words, is every interrupt request serviced? If yes, give a proof. If no, give a counterexample.

Solution: No. If the state is Active and Pending, then any *assert* input is ignored.

7. Consider the following program that monitors two sensors. Here `sensor1` and `sensor2` denote the variables storing the readouts from two sensors. The actual read is performed by the functions `readSensor1()` and `readSensor2()`, respectively, which are called in the interrupt service routine `ISR`.

```
char flag = 0;
volatile char* display;
volatile short sensor1, sensor2;

void ISR() {
  if (flag) {
     sensor1 = readSensor1();
  } else {
     sensor2 = readSensor2();
  }
}

int main() {
  // ... set up interrupts ...
  // ... enable interrupts ...
  while(1) {
     if (flag) {
        if isFaulty2(sensor2) {
           display = "Sensor2 Faulty";
        }
     } else {
        if isFaulty1(sensor1) {
           display = "Sensor1 Faulty";
        }
     }
     flag = !flag;
  }
}
```

Functions `isFaulty1()` and `isFaulty2()` check the sensor readings for any discrepancies, returning 1 if there is a fault and 0 otherwise. Assume that the variable `display` defines what is shown on the monitor to alert a human operator about faults. Also, you may assume that `flag` is modified only in the body of `main`.

Answer the following questions:

(a) Is it possible for the `ISR` to update the value of `sensor1` while the main function is checking whether `sensor1` is faulty? Why or why not?

Solution: No, because of `flag`. During the entire time that the main function is checking whether `sensor1` is faulty the value of `flag` must be non-zero. Hence, if an interrupt occurs during that time, `ISR` will update `sensor2`, not `sensor1`.

(b) Suppose a spurious error occurs that causes `sensor1` or `sensor2` to be a faulty value for one measurement. Is it possible for that this code would not report "Sensor1 faulty" or "Sensor2 faulty"?

Solution: Yes. It is possible for `ISR` to be invoked twice in a row before the main function gets a chance to check the value of a sampled sensor value, thus overwriting that value before `main` has checked it.

(c) Assuming the interrupt source for `ISR()` is timer-driven, what conditions would cause this code to never check whether the sensors are faulty?

Solution: If the interrupts occur so frequently that the main thread never gets to execute, and infinitely many interrupts occur, then the program will never check whether the sensors are faulty.

(d) Suppose that instead being interrupt driven, `ISR` and `main` are executed concurrently, each in its own thread. Assume a microkernel that can interrupt any thread at any time and switch contexts to execute another thread. In this scenario, is it possible for the `ISR` to update the value of `sensor1` while the main function is checking whether `sensor1` is faulty? Why or why not?

Solution: Yes. Suppose that `main` is checking `sensor2`, so `flag` is non-zero, and the thread scheduler interrupts it and begins executing `ISR`. Suppose that `ISR` checks the value of `flag`, reaching line 7, and then gets interrupted before executing line 7. Suppose that `main` continues executing, changing the value of `flag` to zero, and then begins checking `sensor1`. While checking `sensor1`, `main` could be interrupted again, at which point `ISR` could resume by executing line 7, updating the value of `sensor1`.