Question 1.
The objective of this problem is to design a communication network.

Part a) Yes, the grounds do need to be connected together, because the encoding scheme uses voltage (potential difference), which must have a common reference.

Part b) The resistor can be any value from 5kΩ to 25kΩ. Hardware debouncing is not necessary, because the switches will only be changed while the power is off.

Part c) Give the RingInit, RingAccept, RingSend functions along with the interrupt handlers required.

Buffered I/O means the actual SCI I/O must occur in the background. The foreground passes data using Put and Get

```
#include "RxFifo.h"
#include "TxFifo.h"

#define TDRE 0x80
#define RDRF 0x20

void RingInit(void){
    RxInitFifo();
    TxInitFifo();
    SC0BDH=0;
    SC0BDL=1; // baud rate is 500,000 bits/sec
    DDRC=0x00; // PC2-PC0 switch inputs
    SC0CR1=0;
    SC0CR2=0x2C;
    asm(" cli");
}

char RingAccept(void){ char letter;
    while (RxGetFifo(&letter) == 0){};
    return(letter);
}
```
void RingSend(char destinationID, char data){
    char message = (destinationID<<5)+(data&0x1F); // combine ID and data
    while (TxPutFifo(message) == 0);;
    SCOCR2=0xAC;
}

#pragma interrupt_handler SciHandler
void SciHandler(void){
    char data;
    char destinationID;
    if(SCORS1 & RDRF){ // incoming message
        data = SCODRL; // read message, acknowledge interrupt
        destinationID = (data&0x0E0)>>5; // look at ID
        if(destinationID == (PORTC&0x07)){ // match?
            RxPutFifo(data&0x1F); // yes, capture for this node
        } else{ // no, pass it on
            TxPutFifo(data); // data is lost if TxFifo is full!!
            SCOCR2=0xAC; // arm transmission
        }
    }
    if(SCORS1 & TDRE){
        if(TxGetFifo(&data))
            SCODRL=data;
        else
            SCOCR2=0x2c;
    }
}

extern void SciHandler();
#pragma abs_address:0xffd6
void (*SCI_interrupt_vector[])( ) = { SciHandler };
#pragma end_abs_address

#include "RxFifo.c"
#include "TxFifo.c"

The above solution only works if the TxFifo is big enough to never fill. Another solution is to write two
TxPutFifo() functions. The one called by the foreground, TxPutFifo1, will leave a couple of spaces for the
background to use. The second put fifo routine, the regular TxPutFifo, is called by the background and allowed
to use the entire fifo. To simplify this effort, we will switch over and use the pointer-counter implementation shown
in Chapter 4. Here is the regular TxPutFifo, called by the ISR. The TxGetFifo routine can be found in chapter 4.

int TxPutFifo(char data){
    char SaveSP;
    if(TxSize == TxFifoSize ){
        return(0);    /* Failed, fifo was full */
    }
    else{
        asm(" tpa\n staa %SaveSP\n sei"); /* make atomic, entering critical*/
        TxSize++;
        *(TxPutPt) = data; /* put data into fifo */
        TxPutPt++;
        if(TxPutPt == &TxFifo[TxFifoSize]){
            TxPutPt = &TxFifo[0]; /* Wrap */
        }
        asm(" ldaa %SaveSP\n tap"); /* end critical section */
        return(-1);  /* Successful */
    }
}

The other TxPutFifo, called by the RingSend is,

int TxPutFifo1(char data){
    char SaveSP;
    if(TxSize+2 >= TxFifoSize ){ // keeps two empty spaces
        return(0); /* Failed, fifo was full */
    }
    else{

asm(" tpa\n staa %SaveSP\n sel");    /* make atomic, entering critical*/
TxSize++;
*(TxPutPt) = data;    /* put data into fifo */
TxPutPt++;
if(TxPutPt == &TxFifo[TxFifoSize]){  
    TxPutPt = &TxFifo[0];    /* Wrap */
}
asm(" ldaa %SaveSP\n tap");  /* end critical section */
return(-1);    /* Successful */
}

Part d) It takes 10 bit times to output a frame. The baud rate is 500,000 bits/sec, so the bit time is 2 µs. Thus is takes 20 µs to send a frame. Assuming a continuous data stream and the computers are fast enough, computer 0 will output a new frame every 20 µs. There are 5 bits of information in every frame, so the bandwidth is 5bits/20µs = 250,000 bits/sec. For a real system the bandwidth is probably limited by the software execution speed of the microcomputers.

(40) Question 2. This Moore FSM controls traffic at an intersection.

const struct State {
    unsigned char Out;    /* Output to Port B */
    unsigned short Time;  /* Time in sec to wait in 10 msec units */
    const struct State *Next[4]; /* Next if input=00,01,10,11*/
};

typedef const struct State StateType;
#define goN   &fsm[0]
#define waitN &fsm[1]
#define goE   &fsm[2]
#define waitE &fsm[3]

StateType fsm[4] = {
    {0x21, 3000, {goN,waitN,goN,waitN}}, /* goN state */
    {0x22, 500, {goE,goE,goE,goE}}, /* waitN state */
    {0x0C, 3000, {goE,goE,waitE,waitE}}, /* goE state */
    {0x14, 500, {goN,goN,goN,goN}}     /* waitE state */
};

Part a) Show the ritual that initializes the traffic light system.

StateType *pt; // pointer to current state
unsigned short Count; // number of 10 ms to wait in this state

void doOutput(void){
    PORTB = pt->Out;   // set traffic light pattern
    Count = pt->Time;  // wait time in 10 ms units
}

void Ritual(void){
    TIOS  |= 0x01; // enable output compare 0
    TSCR  |= 0x80; // enable timer
    TMSK2 = 0x33;  // 1 us clock
    TMSK1 |= 0x01; // arm OC0
    TFLG1 = 0x01; // initially clear C0F
    pt = goN;     // initial state
    DDRB = 0x3F; // PB5, PB4, PB3, PB2, PB1, PB0 are outputs to traffic lights
    DDRC = 0x00; // PCI and PC0 are inputs from car sensors
    doOutput(); // set outputs, delay Count
    asm(" cli");
}
Part b) Show the ISR that runs the controller in the background.

```c
#pragma interrupt_handler OC0Handler
void OC0Handler(void){
    unsigned char sensor;
    TC0 = TC0 + 10000; // interrupt every 10 ms
    TFLG = 0x01; // acknowledge interrupt by clearing C0F
    Count--;
    if(Count == 0){ // is delay over?
        sensor = PORTC&0x03; // read sensors
        pt = pt->Next[sensor]; // go to next state
        doOutput(); // set outputs, delay Count
    }
}

extern void OC0Handler();
#pragma abs_address:0xffee
void (*OC0_interrupt_vector[])() = { OC0Handler};
#pragma end_abs_address
```