COMP 2659 Course Project – Stage 10c: 2-Player Mode

Released: Tuesday, April 7, 2015
Target Completion Date: Thursday, April 16, 2015 (at the latest)

Overview

This stage will see you completing the 2-player version of your game. You will connect two STs via their RS-232 serial ports for head-to-head play.

In preparation, this stage involves developing a game-independent, low-level serial communication library. Some guidance for how to complete this stage may be given in class. See also the provided tutorial notes.

Some of the PCs in E203 (in the middle column) have been paired up. The paired computers are connected by a null model cable connecting their respective serial ports. Use these PCs for testing your software.

Important Note: if you attempt this stage but run out of time to get the full 2-player game completed, a working serial library (as demonstrated by working test drivers) will still be worth a significant number of marks.

Requirement: Provision of a Serial Library

Remember that using TOS system calls for serial I/O is strictly disallowed. Your solution must control the RS-232 port hardware directly.

Construct a `serial` module. It must consist of:

`serial.h`	*the header file for the module's interface*
`serial.c`	*the majority of the implementation*
`serial.s`	*the ISRs*

At a minimum, the module must contain the following functions:

- `void serial_setup();`

  Initializes the serial interface and its associated buffers, and installs the needed ISRs.

- `void serial_teardown();`

  Cleans up. Restores all original settings.

- `int serial_write(const char *buff, unsigned int size);`

  Enqueues `size` bytes from `buff` in the serial output buffer. This function is non-blocking, meaning that it returns without waiting for the data to be transmitted. Transmission continues "in the background". The number of bytes actually enqueued is returned (this may be less than `size`, if there was insufficient space in the buffer).

- `int serial_read(char *buff, unsigned int size);`

  Like `serial_write`, except that if dequeues up to `size` bytes from the serial input buffer and stores them in `buff`. The number of bytes actually dequeued is returned.

- `int serial_can_write();`

  Returns the number of bytes which can be successfully written, at the time of call.

- `int serial_can_read();`

  Like `serial_can_write`, but for reading.

- `int serial_tx_pending();`

  Returns 0 if the serial output buffer is empty. Returns non-0 otherwise. In other words, this is used to test if all previously written bytes have been transmitted or not.

- `void tx_empty_isr();`

  The ISR which is invoked whenever the USART's Tx register becomes empty.

- `void rx_full_isr();`

  The ISR which is invoked whenever the USART's Rx register becomes full.

Guidance: Testing the Serial Library

Thoroughly testing your module, using multiple test drivers, will be essential to success.

It is suggested that one of your first tests be a `tx_hello` program paired with an `rx_hello` program.

The `tx_hello` program will do the following:

```
serial_setup();
serial_write("hello\n", 7);     /* x-mit null terminator too */

while (serial_tx_pending())
      ;

serial_teardown();
```

The `rx_hello` will do:

```
char ch;

serial_setup();

while (!serial_read(&ch, 1))
      ;

while (ch != '\0')
{
      printf("%c", ch);

      while (!serial_read(&ch, 1))
            ;
}

serial_teardown();
```

When the above works, ensure you can enqueue additional data in the serial output buffer, even when the previously written data hasn't been fully transmitted yet! For example:

```
serial_write("hello ", 6);
serial_write("world!\n", 8);     /* enqueue more right away */
```

Another useful test is to ensure that entire structures can be transmitted and received. This may be good preparation for completing the 2-player game.

E.g. `tx_struc`:

```
struct TestRecord rec = { ... };

serial_setup();

if (serial_can_write() >= sizeof(struct TestRecord))
{
      serial_write((char *)&rec, sizeof(struct TestRecord));

      while (serial_tx_pending())
            ;
}
else
      printf("serial output buffer not big enough?\n");

serial_teardown();
```

E.g. `rx_struc`:

```
struct TestRecord rec;

serial_setup();

while (serial_can_read() < sizeof(struct TestRecord))
      ;

serial_read((char *)&rec, sizeof(struct TestRecord));
serial_teardown();

/* display contents to ensure successful reception */
```

You should also develop test drivers which confirm that everything works as expected when you try to write more data to the serial output buffer than will fit in the buffer. Similarly, confirm that everything works if you try to read more data from the serial input buffer than has been received.

Background: Game Server vs. Game Client

In the 1-player version of the game, the program maintains a model. The model is a collection of data structures which represent the current state of the game world and of all its component objects.

In the 2-player version, there will be two instances of the game program. *They cannot both maintain an authoritative version of the game model.* Due to differing clocks and the communication latency between the systems, their models would not be consistent. At best, they would be out-of-sync. with each other. At worst, they could be in fundamental disagreement about game events, such as whether or not the game has been won by a particular player.

To solve this problem, only *one* game program will maintain the *authoritative* model. This will be called the "server" (or "master"). The other will be called the "client" (or "slave").

The client will still have a copy of the model, which it will use mainly for rendering purposes. However, it will not base any game event decisions off of its model. The server will make all decisions, and it will notify the client as necessary.

In summary:

The job of the server is:

- to manage all asynchronous events (i.e. user input events from both from the local player and the remote player), and to update the model as necessary;
- to manage all synchronous events related to game play, and to update the model as necessary;
- to make all game play-related decisions;
- to periodically update the client on the current state of the model.

The job of the client is:

- to inform the server of any user input events generated on that machine;
- to receive model updates from the server, and to update its copy as necessary.

Requirement: Finished 2-Player Game

Your splash screen must allow the user the option of a 2-player game. In fact, there must be two menu options for 2-player mode: one to select "server" mode, and one for "client" mode.

Use the serial communications library functions for transmitting and receiving data between the client and the server. One simple technique is to transmit structures between systems.

The following example uses `render_request` to determine how often to update the client. This is because there is little point in updating the client's model any more frequently than 70 times per second, since that is the maximum rate at which it can refresh its screen (without flickering or tearing).

Note also that the example transmits the entire model to the client. Actually, depending on your game, this may not be necessary. If possible, it is better to only transmit any state details which have changed since the last transmission.

E.g. the following skeleton code shows how a server *might* interact with a client:

```
struct Model model;
struct Update_From_Client client_update;

int render_request = 0;

...

while (!quit)
{
       /* check for locally generated user input events,
          process and update model as necessary: */

       ...

       /* check for remotely generated user input events: */

       if (serial_can_read() >= sizeof(struct Update_From_Client))
       {
              serial_read((char *)&client_update,
                          sizeof(struct Update_From_Client));

              ...     /* process and update model as necessary */
       }

       if (render_request)
       {
              if (serial_can_write() >= sizeof(struct Model))
                     serial_write((char *)&model, sizeof(struct Model));

              ...     /* render view of model */
       }
}
```

If your game logic is efficient, the client will only lag the server by a small fraction of a second. This shouldn't be noticeable to the human players.

It is not necessary that music be kept in sync. between machines.

When the server starts, it must await a client connection. When the client starts, it must send a "connection message" to the server. Once the client has "connected" to the server, the game may begin.