ECED3204 – Lab #5

STUDENT NAME(s): _____ .

STUDENT NUMBER(s): B00 _____ .

Pre-Lab Information

It is recommended that you read this entire lab ahead of time. Doing so will save you considerable time during the lab, as you will be required to write some simple C code during this lab!

Overall Objective

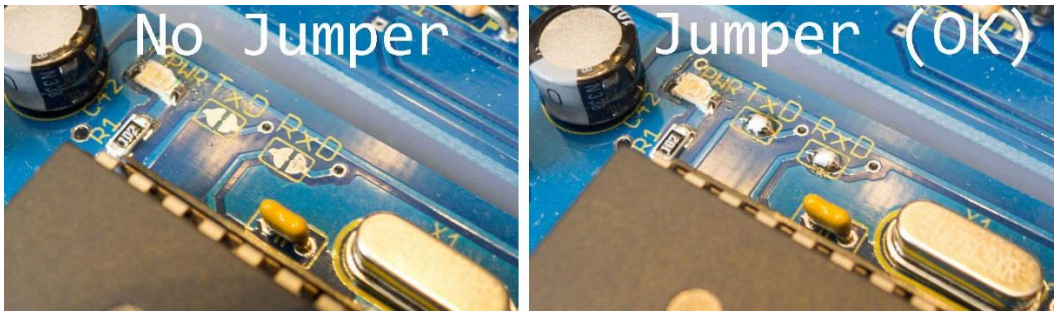
This lab is designed to teach you about using serial communications. You will learn about using the USART module in the AVR, and using the 'printf()' statements from your code for debug and data transfer.

NOTE: There is a video overview at http://www.youtube.com/watch?v=9vljRX5im9o&hd=1 for this lab.

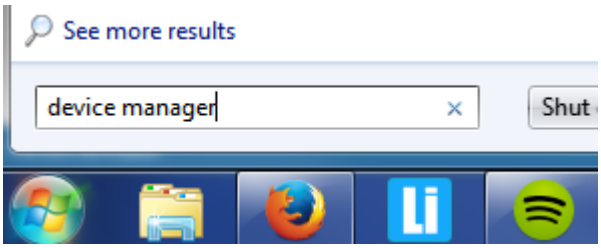
Environment Setup

You will require some setup of your device before being able to work with this lab.

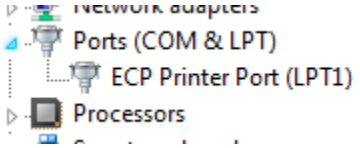
1. Ensure the RXD and TXD jumpers have been bridged with solder. You can look at your board, and if not bridged see the lab technicians to have this fixed:



2. The computer you have plugged your 'programmer' board into needs to be configured to use a USB-Serial converter. You can check if this was already done by opening the device manager (NB: the video link above also shows you how to do this which might be easier to follow):



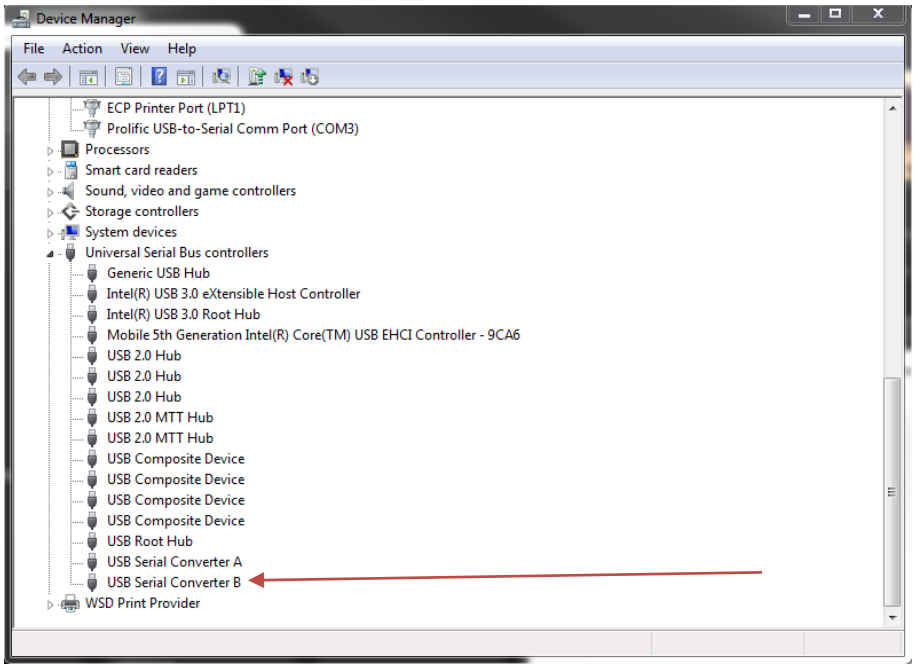
Without the board plugged in, there is no serial ports (you might see a few here too, but we want to look for a change):



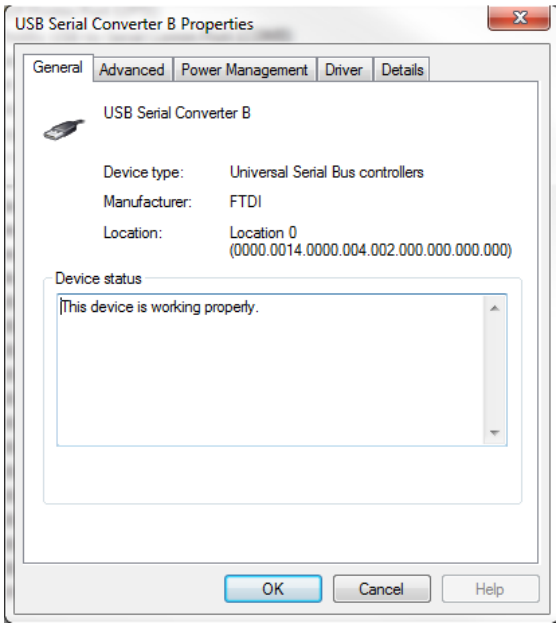
Plugging the 'Programmer Board' into the USB connector gives me a *new* "USB Serial Port":

If this does not happen, perform the following steps. If you DO get the USB Serial Port, ignore these steps:

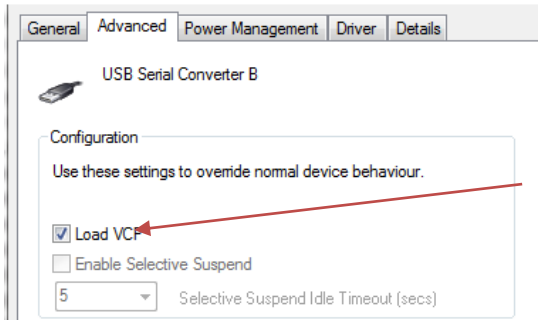
a) Expand the "Universal Serial Bus Controllers" group, and find the "USB Serial Converter B" device:



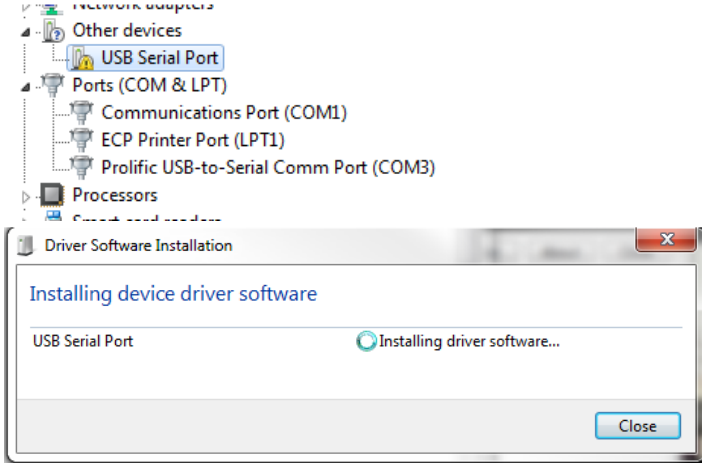
b) Right-click and select ‘Properties’:



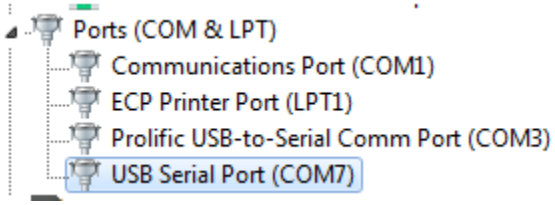
c) Under the ‘Advanced’ tab, select “Load VCP”:



d) Wait for device driver installation to finalize:



e) Confirm the USB-Serial is now appearing.



Part #1: Raw Read/Write of Characters

Objective

- Familiarize yourself with using a timer and interrupt.

Required Materials

- Microprocessor Module with Programmer and TXD/RXD lines shorted.
- Breadboard
- USB Cable
- Power Supply
- Computer with Atmel Studio 6.2 and Programmer Utility installed

Background

Serial communication without an explicit clock is known as *asynchronous* communication. It requires you to configure a number of settings on both sides of the connection, these settings are the 'agreement' between sending and receiving about the meaning of bits.

When debugging your code, this serial communication is an extremely useful tool. It will allow you to print the value of variables, or modify the value of variables without having to recompile your code.

As the first part of this, we will send and receive single characters at a time. This will be extended in Part 2 to send and receive complete strings, and parse those strings into variables.

NOTE: There is a video overview at http://www.youtube.com/watch?v=9vljRX5im9o&hd=1 for this lab.

Procedure

1. Power up the digital trainer board, and connect the programmer. This lab does not use any external devices (no LEDs, etc).

2. Start a new C/C++ project (see Lab #1 for details), copy the following template into it:

```
#include <avr/io.h>
#include <util/delay.h>

void init_uart(void)
{
   UCSR0B = (1<<RXEN0) | (1<<TXEN0);
   UBRR0 = ????
}

void write_char(char c)
{
   //Wait for UDR0 to be ready for writing
   ???

   //Write data byte
   UDR0 = c;
}
```

```
int main(void)
{
   init_uart();


   //STEP1: Just print 'A'
   while(1){
          write_char('A');
          _delay_ms(500);
   }
}
```

3. If you try to compile the code as-is, you will get some errors. This is because there is some stuff you need to do, first figure out the 'baud rate' setting. See the following code:

```
void init_uart(void)
{
   UCSR0B = (1<<RXEN0) | (1<<TXEN0);
   UBRR0 = ????
}
```

This can be found be referring to the ATMega644 Datasheet. See Table 17-11 for example, which gives you settings for the UBRR register. In this case write the value '7' into the UBRR register.

Table 17-11. Examples of UBRRn Settings for Commonly Used Oscillator Frequencies (Continued)

Baud Rate (bps)	f_{osc} = 8.0000 MHz				f_{osc} = 11.0592 MHz				f_{osc} = 14.7456 MHz			
	U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	207	0.2%	416	-0.1%	287	0.0%	575	0.0%	383	0.0%	767	0.0%
4800	103	0.2%	207	0.2%	143	0.0%	287	0.0%	191	0.0%	383	0.0%
9600	51	0.2%	103	0.2%	71	0.0%	143	0.0%	95	0.0%	191	0.0%
14.4k	34	-0.8%	68	0.6%	47	0.0%	95	0.0%	63	0.0%	127	0.0%
19.2k	25	0.2%	51	0.2%	35	0.0%	71	0.0%	47	0.0%	95	0.0%
28.8k	16	2.1%	34	-0.8%	23	0.0%	47	0.0%	31	0.0%	63	0.0%
38.4k	12	0.2%	25	0.2%	17	0.0%	35	0.0%	23	0.0%	47	0.0%
57.6k	8	-3.5%	16	2.1%	11	0.0%	23	0.0%	15	0.0%	31	0.0%
76.8k	6	-7.0%	12	0.2%	8	0.0%	17	0.0%	11	0.0%	23	0.0%
115.2k	3	8.5%	8	-3.5%	5	0.0%	11	0.0%	7	0.0%	15	0.0%
230.4k	1	8.5%	3	8.5%	2	0.0%	5	0.0%	3	0.0%	7	0.0%
250k	1	0.0%	3	0.0%	2	-7.8%	5	-7.8%	3	-7.8%	6	5.3%
0.5M	0	0.0%	1	0.0%	–	–	2	-7.8%	1	-7.8%	3	-7.8%
1M	–	–	0	0.0%	–	–	–	–	0	-7.8%	1	-7.8%
Max. (1)	0.5 Mbps		1 Mbps		691.2 Kbps		1.3824 Mbps		921.6 Kbps		1.8432 Mbps	

1. UBRR = 0, Error = 0.0%

4. We also need to add the block of code to the following section:

```
void write_char(char c)
{
   //Wait for UDR0 to be ready for writing
   ???

   //Write data byte
   UDR0 = c;
}
```

5. You need to loop until the bit 'UDRE0' is set, see this section of the datasheet:

17.10.2 UCSRnA – USART Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

- **Bit 7 – RXCn: USART Receive Complete**

This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (that is, does not contain any unread data). If the Receiver is disabled, the receive buffer will be flushed and consequently the RXCn bit will become zero. The RXCn Flag can be used to generate a Receive Complete interrupt (see description of the RXCIEn bit).

- **Bit 6 – TXCn: USART Transmit Complete**

This flag bit is set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (UDRn). The TXCn Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXCn Flag can generate a Transmit Complete interrupt (see description of the TXCIEn bit).

- **Bit 5 – UDREn: USART Data Register Empty**

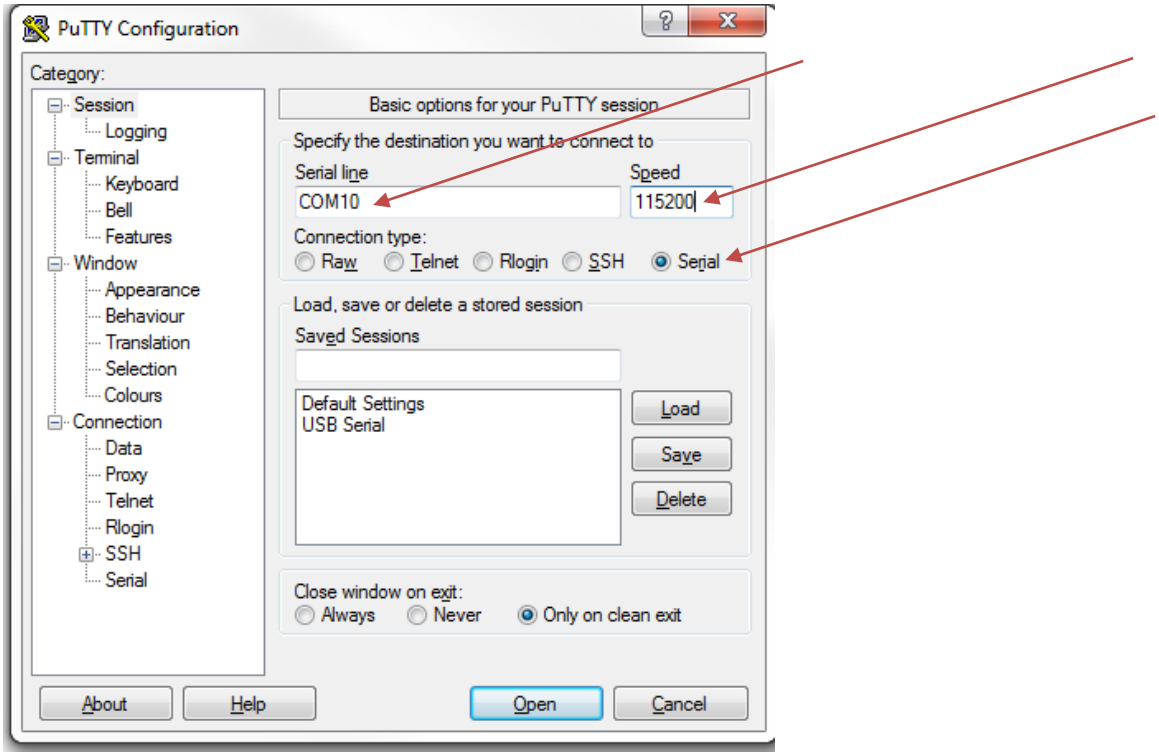
The UDREn Flag indicates if the transmit buffer (UDRn) is ready to receive new data. If UDREn is one, the buffer is empty, and therefore ready to be written. The UDREn Flag can generate a Data Register Empty interrupt (see description of the UDRIEn bit).

You can use the loop_until_bit_is_set() macro if you want for example:

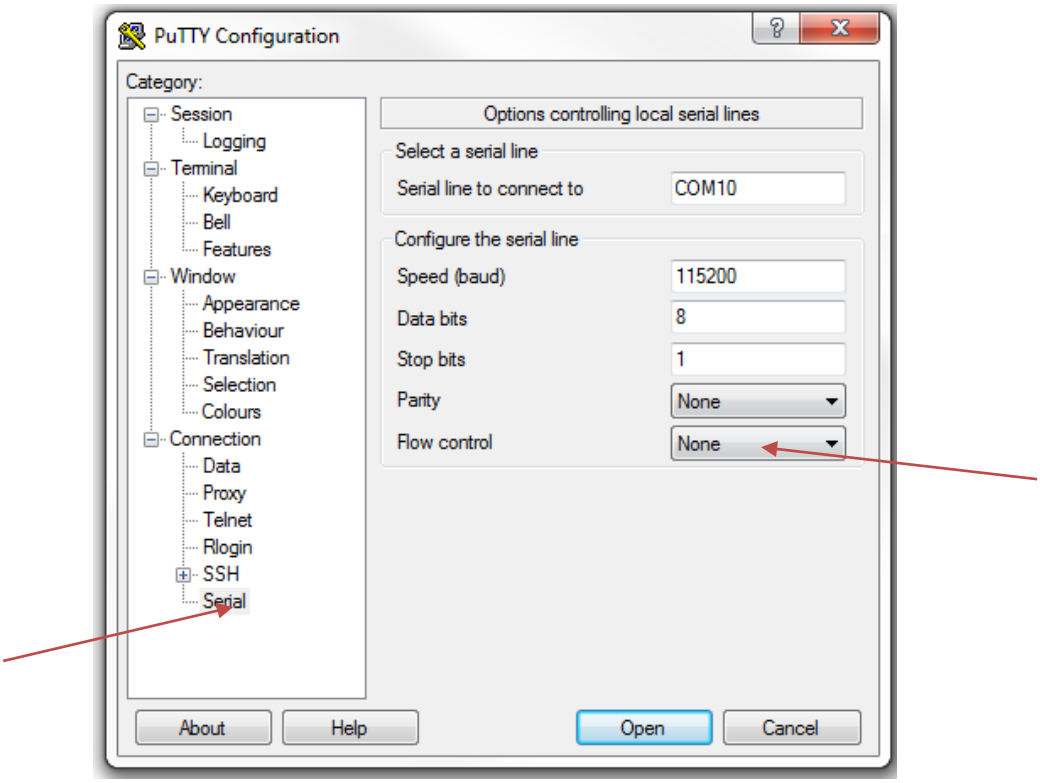
```
void write_char(char c)
{
   //Wait for UDR0 to be ready for writing
   Loop_until_bit_is_set(UCSR0A, UDRE0);

   //Write data byte
   UDR0 = c;
}
```

6. Before continuing, ensure you have performed the required lab setup from the 'prelab' section. This involves soldering your RXD/TXD jumpers and ensuring the proper serial port driver is loaded.
   You will also need to know the COM port number. See the device manager, for example I will be using COM10:

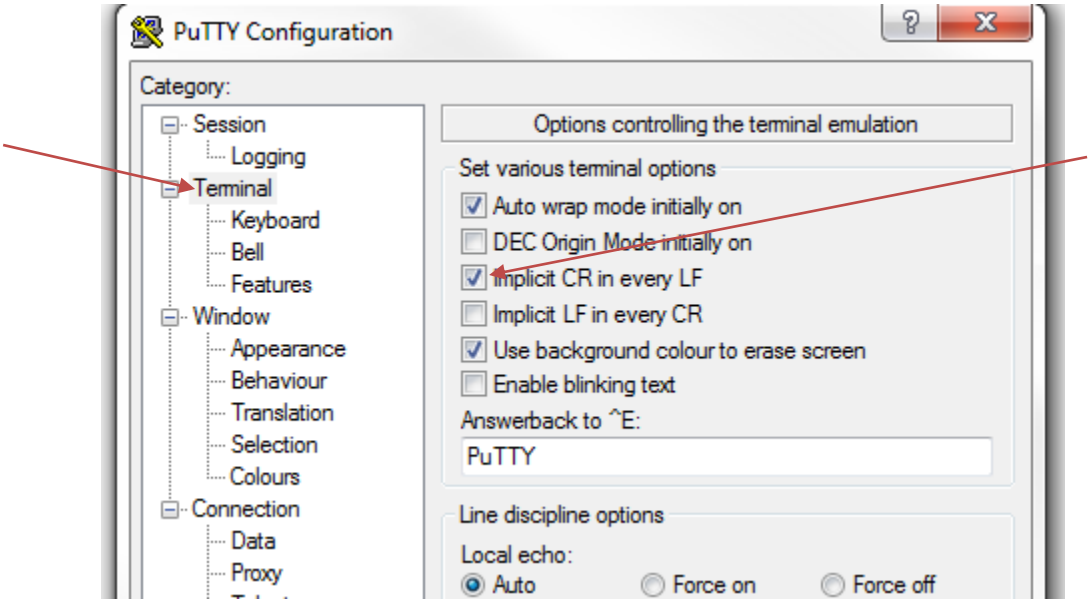
7. Open PuTTY, change the following settings on the main page:
   a) Connection Type = Serial
   b) Serial line = COM10 (on my computer, set as required on your device)
   c) Speed = 115200



8. Under the Connection→Serial setup, change the following settings:
   d) Flow Control: None



9. Under **Terminal**, change the following settings:
   e) Implicit CR in every LF: Checked



10. Hit “Open” to open the serial port. Note you can save the connection as well before hitting Open to avoid you needing to set this all up again each time.

11. Finally, program the AVR device with your code. It should send ‘A’ repeatedly to your screen, if so success! Only once this is working continue onto the next steps.

12. Next, we will wait for a character to be sent to the device. This involves a new version of the code which is capable of checking for a character, see the following example:

```
#include <avr/io.h>
#include <util/delay.h>

void init_uart(void)
{
   UCSR0B = (1<<RXEN0) | (1<<TXEN0);
   UBRR0 = 7;
}

void write_char(char c)
{
   //Wait for UDR0 to be ready for writing
   while((UCSR0A & (1<<UDRE0)) == 0);

   //Write data byte
   UDR0 = c;
}

char check_char(void)
{
   //Check if data ready for reading
   return (UCSR0A & (1<<RXC0));
}

char read_char(void)
{
   //Wait for data
   while((UCSR0A & (1<<RXC0)) == 0);

   //Return data
   return UDR0;
}

int main(void)
{
   init_uart();
   // Check if character received
   while(1){
          if (check_char()){
                 write_char('A');
          } else {
                 write_char('B');
          }
          _delay_ms(500);
   }
}
```

13. Finally, change the main loop to echo every received character back. Confirm on the terminal emulator (PuTTY) that if you write a sentence and hit enter, it is returned to you for example. The following shows such a main loop:

```
int main(void)
{
   init_uart();

   //STEP3: Full echo
   while(1){
          write_char(read_char());
   }

}
```

Part #2: Using <stdio.h>

Objective

- Use printf() and scanf() to send the values of variables to your computer screen, and read in numbers.

Required Materials

- Setup from Part #1

Background

This section extends the basic single-character transmission and reception. In this part of the lab you will use a terminal emulator program such as PuTTY to interact with your device in a more meaningful way.

NOTE: There is a video overview at http://www.youtube.com/watch?v=9vljRX5im9o&hd=1 for this lab.

Procedure

1. Keep the same setup as the previous lab (i.e. still using PuTTY etc)
2. Program the following example code:

```
#include <stdio.h>
#include <avr/io.h>
#include <avr/pgmspace.h>

static int uart_putchar(char c, FILE *stream);
static int uart_getchar(FILE *stream);
FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL, _FDEV_SETUP_WRITE);
FILE mystdin = FDEV_SETUP_STREAM(NULL, uart_getchar, _FDEV_SETUP_READ);

static int uart_putchar(char c, FILE *stream)
{
   loop_until_bit_is_set(UCSR0A, UDRE0);
   UDR0 = c;
   return 0;
}

static int uart_getchar(FILE *stream)
{
   loop_until_bit_is_set(UCSR0A, RXC0); /* Wait until data exists. */
   return UDR0;
}

void init_uart(void)
{
   UCSR0B = (1<<RXEN0) | (1<<TXEN0);
   UBRR0 = 7;
   stdout = &mystdout;
   stdin = &mystdin;
}

int main(void)
{
   init_uart();
   printf_P(PSTR("System Booted, built %s on %s\n"), __TIME__, __DATE__);
```

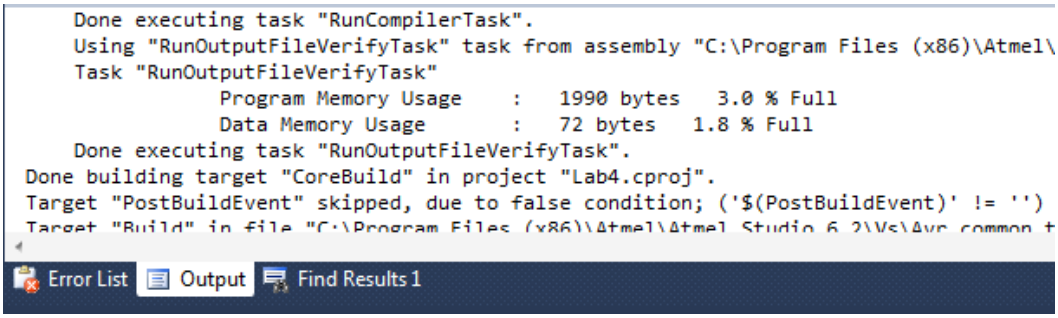
```
    printf_P(PSTR("Hello There. What is your name?\n"));

    char name[32];

    scanf("%s", name);

    printf_P(PSTR("Alright %s. What is a number?\n"), name);

    int number;

    if (scanf("%d", &number) == 1){
        printf_P(PSTR("OK %s, did you pick %d?\n"), name, number);
    } else {
        printf_P(PSTR("Look %s - I said a number. Try again.\n"));
    }
}
```

3. This example code is using the “program memory” macro called PSTR along with special printf_P() calls, which stores data to FLASH memory instead of wasting storage in SRAM.

   As a test, change the main routine to instead use regular printf() statements as below:

```
int main(void)
{
    init_uart();
    printf("System Booted, built %s on %s\n", __TIME__, __DATE__);

    printf("Hello There. What is your name?\n");

    char name[32];

    scanf("%s", name);

    printf("Alright %s. What is a number?\n", name);

    int number;

    if (scanf("%d", &number) == 1){
        printf("OK %s, did you pick %d?\n", name, number);
    } else {
        printf("Look %s - I said a number. Try again.\n");
    }
}
```

   Record the usage of SRAM and FLASH across each version. The SRAM and FLASH usage is printed when Atmel Studio compiles your code. Switch to the ‘Output’ tab and scroll up slightly, for example something like this is printed, which shows Program Memory (FLASH) usage and Data Memory (SRAM) usage:



4. Experiment with the code, and add your own print statement.

Lab Questions

1. What does ‘115200’ in the baud rate setting refer to from Part 1?

2. What was the difference in SRAM and FLASH memory usage when switching to printf_P() instead of regular printf() statements in Part 2?