Lab 2: UART transmission with the ATmega328P/PB

Introduction

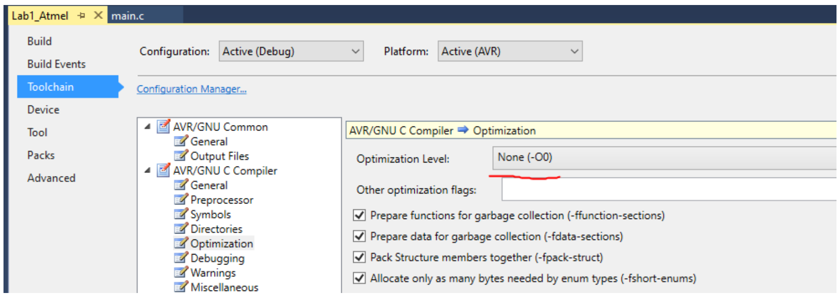
The objective of this lab is to introduce you to data transmission, using the UART protocol and polling. You will also develop experience using Atmel/Microchip Studio and the Proteus Virtual System Modelling (VSM) tool to develop and debug your embedded software. You will test the functionality of your code using Proteus VSM, which emulates a virtual ATmega328P microcontroller and associated peripherals. Subsequently you will test the code you develop on an ATmega328P /ATmega328PB Xplained Mini evaluation board.

Read the entire lab manual before you start. This lab should take you approximately **4 hours**. There is a pre-lab, five compulsory parts and a few optional activities. **You should aim to complete at least the pre-lab before attending the support session. The final answers should be written in this document, with supporting calculations shown in your logbook.** Before starting the lab, pull the lab repository from GitHub as it contains the Proteus model(s). Remember to commit and push your saved work (code and Proteus simulations) to the lab repository on GitHub after completing each task.

Pre-Lab: Using Atmel Studio

In this course we are using the ATmega328P in Proteus VSM (since it only supports this MCU) and the ATmega328P/ATmega328PB when it comes to testing your code on actual hardware. ATmega328P behaves almost the same as the ATmega328PB you already encountered in COMPSYS201. The code written for an ATmega328P could be directly compiled to work on the ATmega328PB (all most all the time). Before the lab, we want to make sure that you are familiar with how to use Atmel/Microchip Studio to write some code, compile, and simulate. Referring to your CS201 coursebook where necessary, complete these tasks:

1) Create a new GCC C Executable Project in Atmel/Microchip Studio, selecting ATmega328P as the device family (note the new version of Atmel Studio is called Microchip Studio).
2) Write a piece of code that creates an array of prime numbers up to 300. HINT: Refer to lecture notes - there should be 62 prime numbers in total. You can use a simple algorithm that iterates through all the numbers up to 300, and checks if they can be exactly divided by numbers smaller than it using the modulo operator. This code should only be executed once, so write your code *before* the while(1) loop. Make sure to use an appropriate variable type for storing all of the numbers in this array.
3) Before building the program, go to Project > [ProjectName] Properties > Toolchain > AVR/GNU C Compiler > Optimization, and set the Optimization level to "None –O0". This ensures that the compiler doesn't remove any of your code that it thinks isn't being used:



Configuration: Active (Debug) Platform: Active (AVR)

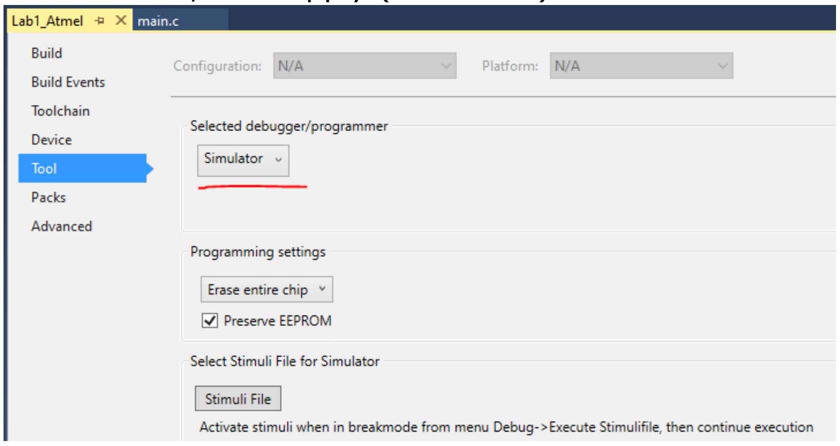
AVR/GNU C Compiler ➡ Optimization

Optimization Level: None (-O0)

Other optimization flags:

☑ Prepare functions for garbage collection (-ffunction-sections)
☑ Prepare data for garbage collection (-fdata-sections)
☑ Pack Structure members together (-fpack-struct)
☑ Allocate only as many bytes needed by enum types (-fshort-enums)

4) Build the program, and then go to Tools > Device Programming, select Simulator for the Tool, click Apply (see below):



Configuration: N/A Platform: N/A

Selected debugger/programmer: Simulator

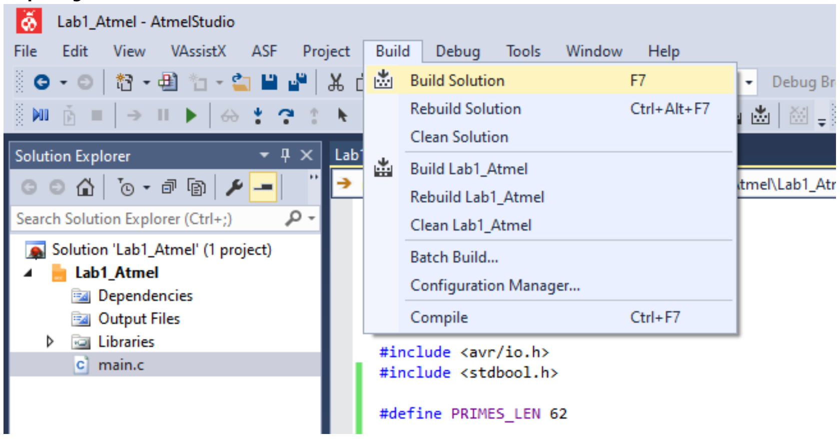
Programming settings: Erase entire chip

☑ Preserve EEPROM

Select Stimuli File for Simulator

Activate stimuli when in breakmode from menu Debug->Execute Stimulifile, then continue execution

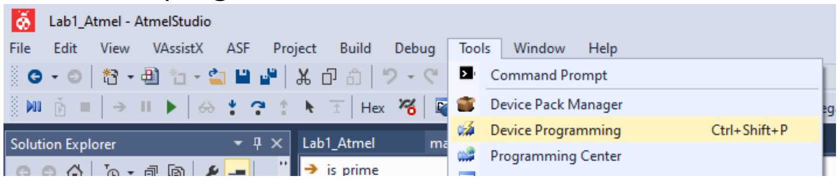
5) Build your project:



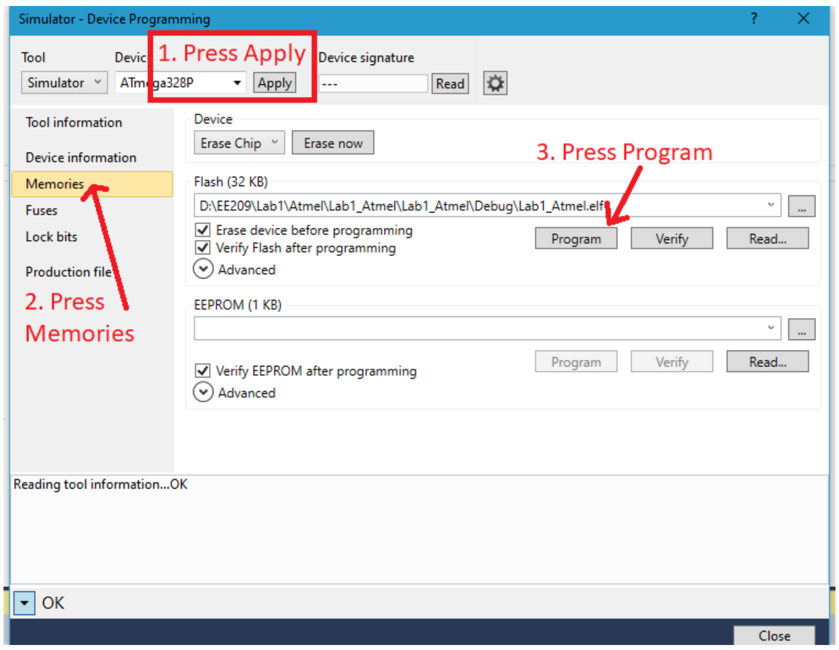
```
#include <avr/io.h>
#include <stdbool.h>

#define PRIMES_LEN 62
```

6) Bring up the device programmer:



7) Press “Apply” with “Simulator” active, press “Memories”, and click “Program”:



Simulator - Device Programming

Tool: Simulator | Device: ATmega328P | Apply | Device signature: --- | Read

Tool information, Device information, Memories, Fuses, Lock bits, Production file

Device: Erase Chip | Erase now

Flash (32 KB): D:\EE209\Lab1\Atmel\Lab1_Atmel\Lab1_Atmel\Debug\Lab1_Atmel.elf

☑ Erase device before programming
☑ Verify Flash after programming
Advanced

Program | Verify | Read...

EEPROM (1 KB)

☑ Verify EEPROM after programming
Advanced

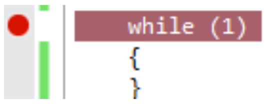
Program | Verify | Read...

```
Reading tool information...OK
```

OK

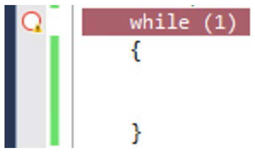
Close

8) Press "Close" after you've pressed Program.
9) Set up a breakpoint next to the "while(1)" loop. You can either click on the left-hand margin next to the code on the appropriate line or right click on the "while(1)" and select Breakpoint>Insert Breakpoint. A dark red circle should appear to indicate that there is a breakpoint here:



```
while (1)
{
}
```

Make sure there are NO blank lines between the two brackets of while(1) as shown above. If you had by mistake inserted blank lines between the two bracket of while(1), when you start debugging as per the next step below, the dark red circle will become a white circle with a red boarder and a warning will be shown as in:

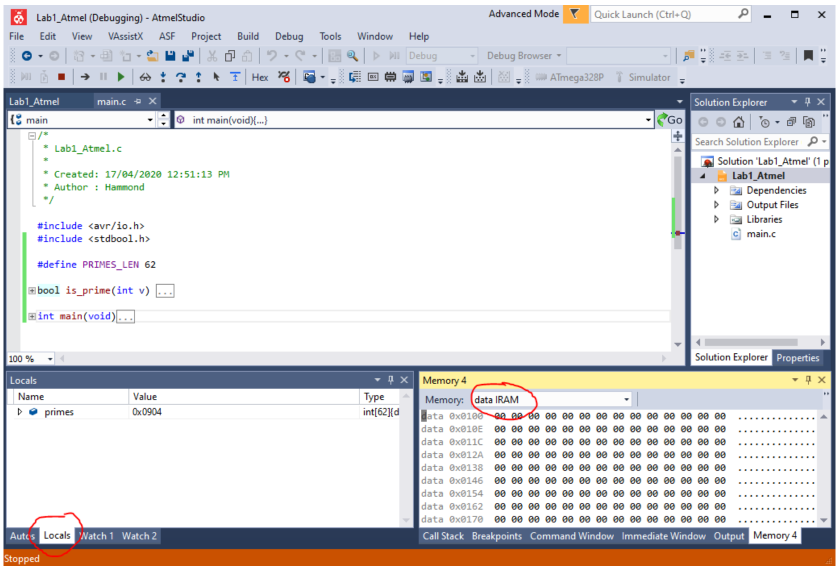


```
while (1)
{


}
```

If this is the case, delete the blank lines, build and program as per previous steps.

10) Press the button to "Start Debugging and Break" or press (Alt + F5). This should execute the code and open a couple of new windows. In the bottom left, you should go to the "Locals" tab to see your local variables, such as your array of prime numbers (our array is named *primes* in this example). On the bottom right, you should change the Memory window from "prog FLASH" to "data IRAM". Here, you can see the raw hexadecimal values of what would be in the memory of the microcontroller:



```
/*
 * Lab1_Atmel.c
 *
 * Created: 17/04/2020 12:51:13 PM
 * Author : Hammond
 */

#include <avr/io.h>
#include <stdbool.h>

#define PRIMES_LEN 62

bool is_prime(int v) ...

int main(void)...
```

Locals

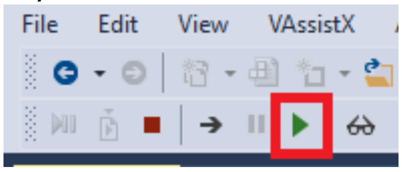
Name	Value	Type
primes	0x0904	int[62]{d

Memory 4

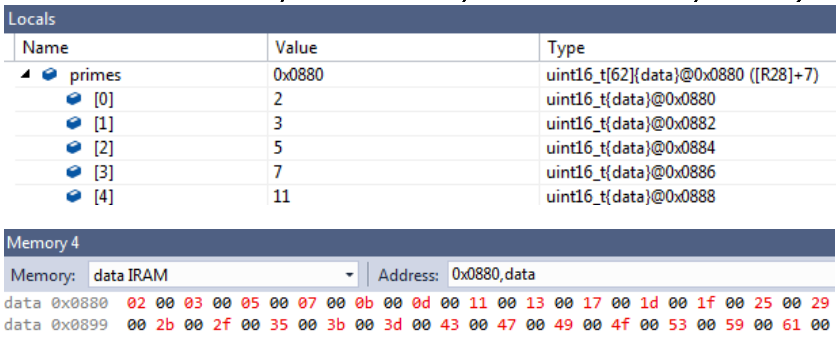
Memory: data IRAM

```
data 0x0100  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
data 0x010E  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
data 0x011C  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
data 0x012A  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
data 0x0138  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
data 0x0146  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
data 0x0154  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
data 0x0162  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
data 0x0170  00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
```

11) Press "Continue" (it may take a few minutes to simulate - so be patient):



12) Using the memory location given in the "Locals" window (under "Value", e.g. for us 0x0880), find the corresponding memory address in the Memory window. Verify that both of the "Locals" and "Memory" windows show you a list of 62 prime numbers up to 300. Confirm that both windows are actually showing you the same numbers in decimal and hex (we include an example here – note that your memory addresses may differ):



Locals

Name	Value	Type
primes	0x0880	uint16_t[62]{data}@0x0880 ([R28]+7)
[0]	2	uint16_t{data}@0x0880
[1]	3	uint16_t{data}@0x0882
[2]	5	uint16_t{data}@0x0884
[3]	7	uint16_t{data}@0x0886
[4]	11	uint16_t{data}@0x0888

Memory 4

Memory: data IRAM Address: 0x0880,data

```
data 0x0880  02 00 03 00 05 00 07 00 0b 00 0d 00 11 00 13 00 17 00 1d 00 1f 00 25 00 29
data 0x0899  00 2b 00 2f 00 35 00 3b 00 3d 00 43 00 47 00 49 00 4f 00 53 00 59 00 61 00
```

13) While the debugger is running, at the top of the code window you should also see a new Disassembly tab. You may be interested in having a look at this, which shows the assembly code that your C code has compiled to. When you are finished, click "Stop Debugging" to return to your code.

Q P.1: What was the largest number stored in the array, in decimal and in hex?

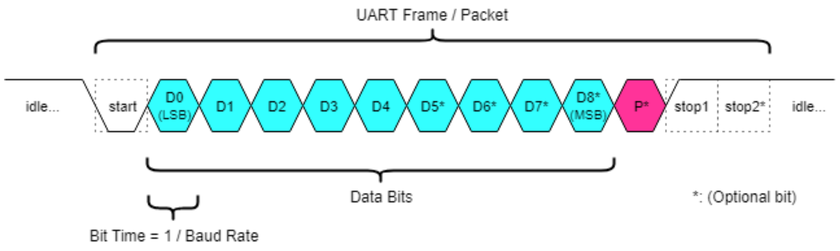
Decimal: 293 Hex: 0x125

Q P.2: Why did the memory window show a lot of 00 elements between some of the numbers in the array?

Because each number in the array is stared in a 16-bit integer, but many of the initial prime numbers are small enough to fit within 8 bits.

Part 1: What is UART?

Let us imagine that we have two computers, and they need to talk to each other. We want to get data from Computer A to Computer B, so we form a connection between them. Great! But what do we send down that connection?
Assume that computer A is trying to talk to computer B. Computer A could send a bunch of 0 and 1 bits over a period of time in a *serial* manner (i.e. one bit at a time). If Computer B doesn't know how long each bit is going to be or how much data there is or when to start reading, then it cannot understand what Computer A is sending to it. So, we need a *communication protocol*, a set of rules that Computer A and Computer B both agree upon in order to know how to transmit and receive the data. UART is one such protocol (among others discussed in lectures), and it has a number of *parameters* which allow the same protocol to be used in slightly different ways for different purposes. This diagram shows what a single UART frame or packet should look like:



Q 1.1: What is the purpose of each of the following parameters and bits?

i) The start bit Signals to the receiver that a data packet is beginning, allowing it to synchronize for reading the incoming bits.

ii) The parity bit An optional bit used for basic error checking during transmission

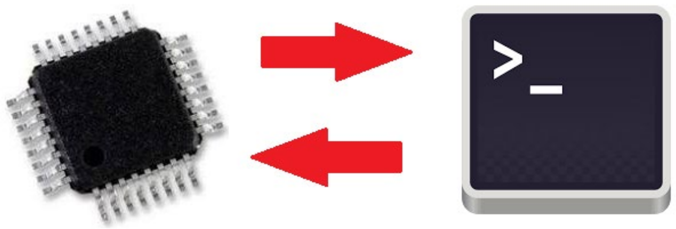
iii) The stop bits Indicates the end of the data packet, allowing the receiver to distinguish between separate packets.

iv) The baud rate Defines the speed of data transmission, ensuring both devices can agree on the length of each bit.

In order for two devices to communicate over UART, all parameters used by both devices need to match. For this lab (and in the project), we will use these values:

Baud Rate	9600	Parity Bit	None
# Data Bits	8	# Stop Bits	1

The aim of this lab is to send some data out of your Microcontroller via the UART. In the real world, any device could be listening, whether that be a computer, a cell phone, another microcontroller, etc. In this initial part of this lab, we'll be sending data to Proteus' *virtual terminal*. After you verify the functionality of your program in Proteus, you load this program to the Xplained Mini that's connected to a computer through the USB cable. The data coming out of the microcontroller is then viewed on the computer using a terminal program such as PuTTY or RealTerm:



On the ATmega328P microcontroller, there is a USART module (the ATmega328PB has an additional USART module, which is one of the very few differences). This is a standalone module whose sole purpose is to transmit and receive data according to the USART protocols. This module knows all of the rules already, so we only need to tell the module two things: what parameter values we want it to use (**control**) and what we want to transmit (**data**).
Note: UART stands for Universal Asynchronous Receiver Transmitter – the extra S in USART stands for Synchronous, because the USART module can be used to also follow a synchronous protocol. However, for the purposes of this project, we only have a single communication line, so we cannot send a synchronising clock along with our data to the receiver side. There are better synchronous communication protocols anyway, like SPI and I^{2}C, but UART is the most popular asynchronous protocol. So wherever you see USART in the datasheet or online, in most cases it is the same thing as UART.

Part 2: How to Control the Microcontroller?

We primarily interact with the I/O peripherals (also known as hardware modules) on a microcontroller through its *registers*. Some registers just store data, while the vast majority are used to control how the peripheral behaves. In order to control a peripheral, there are a bunch of settings that need to be configured. Most settings only need one or two bits to configure, so a single register often manages more than one setting. The ATmega328P registers are 8-bit (hence it is referred to as an "8-bit" microcontroller). There are five 8-bit control registers and one 8-bit data register associated with the USART peripheral on this microcontroller.

Q 2.1: Using the datasheet, state the register and bits that control each of the settings below, and describe the purpose of each of these settings. Identify whether or not we need this setting for this lab, and if we do, determine if the setting is set at initialisation or if it is used at runtime:

Setting Name	Register and Bits	Purpose	Do we need it?	Initialisation/ Runtime
Receive Complete	*UCSR0A RXC0 (Bit 7)*	*A flag indicating when the USART has received a complete packet that is ready to be read*	*No*	-
Tx Data Register Empty	*UCSR0A UDRE0 (Bit 5)*	*A flag indicating that the USART is ready to send another packet and it is safe to load data*	*Yes*	*Runtime*

Transmit Complete	UCSR0A Bit 6 – TXC0	A flag indicating that the entire frame has been shifted out and the transmitter has no new data.	No	
Mode Selection	UCSR0C Bits 7:6 – UMSEL0[1:0]	Selects between Asynchronous, Synchronous, and Master SPI modes.	Yes	Initialisation
Character Size	UCSR0C Bits 2:1 – UCSZ0[1:0] and UCSR0B Bits 2 - UCSZ02	These three bits combined set the number of data bits (5 to 9) in a frame while UCSR0B must me 0	Yes	Initialisation
Clock Polarity	UCSR0C Bit 0 – UCPOL0	Used for synchronous mode only to set the relationship between data and the clock edge.	No	
Baud Rate	`UBRR0H and UBRR0L`	These two registers together hold a 12-bit value that determines the baud rate.	Yes	Initialisation
Receiver Enable	UCSR0B Bit 4 – RXEN0	Enables the USART receiver.	No	
Transmitter Enable	UCSR0B Bit 3 – TXEN0	Enables the USART transmitter.	Yes	Initialisation
Parity Mode	UCSR0C Bits 5:4 – UPM0[1:0]	Enables and selects between no parity, even parity, or odd parity.	Yes	Initialisation
Parity Error	UCSR0A Bit 2 – UPE0	A flag indicating that the next frame in the receive buffer had a parity error.	No	

One of the most important USART parameters is the *baud rate*. Inside the USART peripheral, there is essentially a clock divider that is able to convert the system clock into a clock that matches the desired baud rate. On the ATmega328P, this value is stored in the UBRR register (USART Baud Rate Register).

Q 2.2: Using the datasheet, what is the appropriate UBRR value? Note that our system clock is 2MHz. Binary: 000000001100, Decimal: 12

Q 2.3: On the ATmega328P, the UBRR is divided up between two registers, UBRR0H and UBRR0L. Why do you think this is the case?

because the ATmega328P is an 8-bit microcontroller, but the baud rate calculation requires up to a 12-bit value.

Other than the two UBRR registers, there are three more control registers. We need to decide what values to set them to so that we get the behaviour that we expect/need from the USART peripheral. This is referred to as initialising a peripheral and is typically done at the beginning of your embedded program.

Q 2.4: Using your answers from Q 2.1 and the datasheet, for each of the three control registers below, fill out the table with the short name for each bit, and the value we need to set it to during initialisation. Your options are 0 or 1 – if you don't care what the value is, set it to 0.

UCSR0A

RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
0	0	0	0	0	0	0	0

UCSR0B

RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
0	0	0	0	1	0	0	0

UCSR0C

UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0
0	0	0	0	0	1	1	0

During initialisation, we can directly write these binary values to the three USART control registers. The following example shows how we could set UCSR0A to have 00100101, assuming we have <avr/io.h> included in the project.

```
UCSR0A = 0b00100101; //This is an arbitrary value
```

The 0b means that the number is in binary. Sometimes during system operation, we may need to read or write to a specific bit. Thinking back to COMPSYS201, there were three types of instructions that we needed to implement – reading, writing 1s, and writing 0s. Thanks to the AVR SDK (Software Development Kit), we have access to a bunch of macros in avr/io.h, which can make our life slightly easier.
Instead of using integers that don't really mean anything to a human reader, we can use the actual names of the bits instead. For example, we can use (1<<RXC0) instead of (1<<7); as the human-readable name is more *meaningful*. Here RXC0 refers to the position of the RXC0 bit in the UCSR0A control register, which is 7.

Q 2.5: For each of these instructions, finish the C macro expression:

i)	*Test if the UDRE0 bit is 1:*	**if(**UCSR0A **& (1 <<** UDRE0 **)) {**
ii)	*Write a 1 (set) to the TXEN0 bit:*	UCSR0B **|= (**1 **<<** TXEN0 **);**
iii)	*Write a 0 (clear) to the UCPOL0 bit:*	UCSR0C &= ~(1 << UCPOL0);

Part 3: What Data to Send?

In the datasheet, we can find information about the UDR0 register, and how it acts as a buffer for transmitting and receiving data. In order to send data through UART, we just need to write some data to this register. Once you write data to UDR0, the USART peripheral immediately starts sending that data according to

the UART protocol, with the settings that you have already initialised the peripheral to.
Before we can do that, we probably need to store the data we are sending in a variable of some sort, and then set the UDR0 register to the value of that variable.

Q 3.1: On the ATmega328P, how many bits of data can be stored in a variable of each of these types:

char: 8 bits *uint8_t:* 8 bits

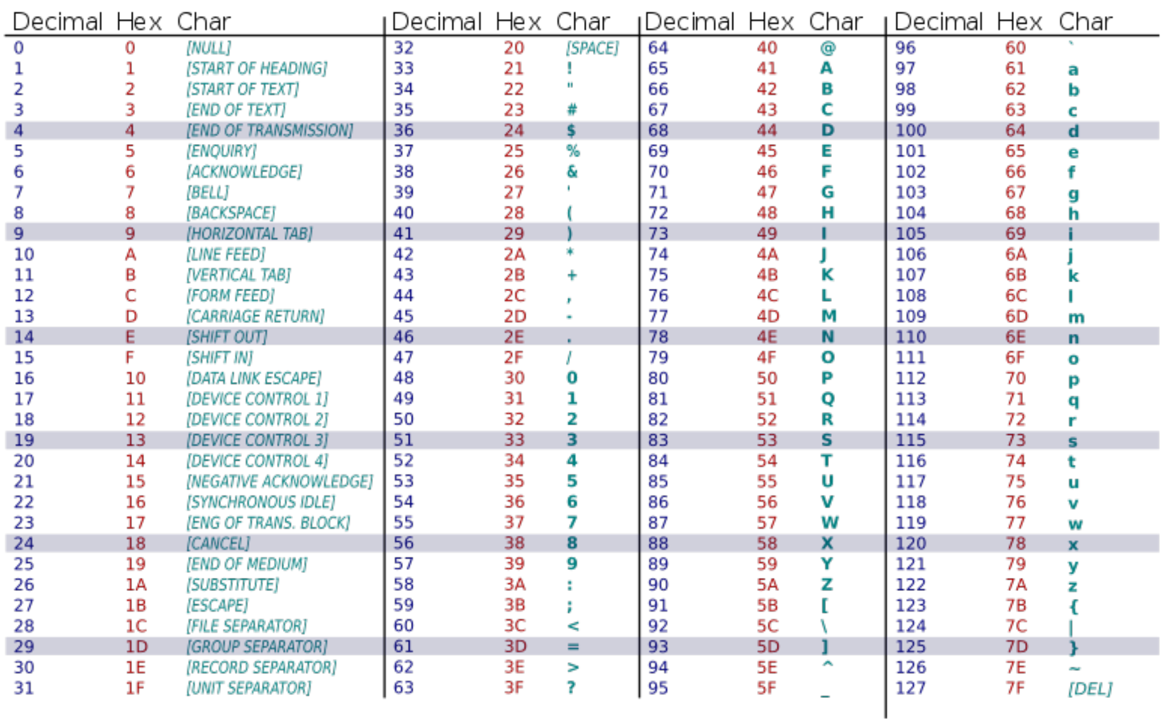
int: 16 bits *uint16_t:* 16 bits

int8_t: 8 bits *float:* 32 bits

Q 3.2: What is the size of the UDR0 register? 8 bits

Q 3.3: What is the largest number we can transmit? 255

Given these limitations, it might seem like we can't send very much at once. There's another element to consider though – we have already talked about how we need both the transmitter and receiver to agree on a set of parameters that dictate how the data is formatted. Actually, this only determines the physical layer – how are the bits physically sent and received. It does not consider what those bits actually mean! At this stage, they are just a bunch of 0s and 1s.
Of course, we could just interpret these as integers. But what if we wanted to send a word, or a song, or a picture? Then we need *character encoding*, which is like having a dictionary that says certain integers should mean different things. You can also think of this like a truth table – if the input is integer X, then the output should be data Y. One of the most popular encodings is ASCII:



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

In the table, the numbers on the left are the integer values that correspond to the character outputs on the right in each column.

Q 3.4: What decimal numbers correspond to the word "HELLO"? 72, 69, 76, 76, 79

ASCII is a very old character encoding standard (from the 1960s!), originally developed for telegraphs. This is why there are a bunch of characters in the first column that don't seem to mean much, and are largely ignored in modern computing. You might notice that the decimal numbers go up to 127 – this is intentional, as it allows us to represent the characters within one byte.
Most *terminal* programs speak ASCII. When you send data from your ATmega328P /ATmega328PB to a computer or a phone, if using a terminal program on the receiving end it is going to try to decode the numbers that you send with ASCII. This means that if you want the terminal program to show the number 63, then you can't just send the number 63 – you have to send a 54 (i.e. "6") followed by a 51 (i.e. "3") in order to display the characters 6 and 3.

Q 3.5: Fill in the blanks below to describe how to send a number to a terminal: We can extract the individual digits of a number by using the Modulo *operator. Then we encode an individual digit into its ASCII equivalent by simply adding the decimal number* 48*. Then we iterate through each digit of the number using a* loop*, writing each character to the* UDR0 *register.*

Note: While terminal programs generally use ASCII, in some projects, you may find that you want to use other encodings – even something custom!

Part 4: Transmitting Data

Now that we understand both how to control the USART peripheral and how to format our data correctly, we can send some data to another computer. We will need to write some C code that does this. Our eventual goal is to send the list of prime numbers from the pre-lab to the terminal on a computer, but let's break that up into a few smaller tasks.

Q 4.1: Assuming that we can begin transmitting a UART packet immediately after the previous one ends, work out the length of time required to send each of these:

A single character: 1.0416 ms

A 3-digit number: 3.0438 ms

Q 4.2: Now, also assuming that we need to send a comma character and a space character between each number, work out the length of time to send each of these:

Three 3-digit numbers (there are 2 commas and 2 spaces): 13.5408 ms

The entire primes list from pre-lab (assume all 3-digits): 320.8128 ms

Our C code will have two main parts: initialisation and transmission. The initialisation is only run once, and configures the USART peripheral to behave the way we need it to. The transmission part is run as many times as needed. Therefore, it probably makes sense to turn this into two functions. Modify the Atmel Studio project you created to answer the pre-lab questions to answer the following questions.

Q 4.3: Write a usart_init(uint16_t ubrr) function which sets up the USART peripheral as determined in Part 2. Of the five control registers, how many could be left with their initial values? 1 (UCSR0C)

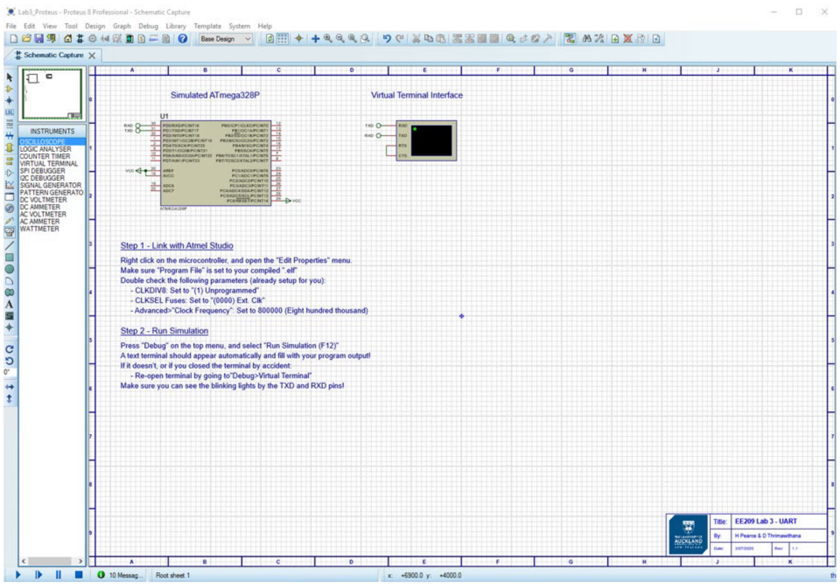
Q 4.4: Write a usart_transmit(uint8_t data) function which handles transmission of a single number through the USART peripheral. The important steps are to:

1) *Check bit* UDRE0 *of register* UCSR0A *and wait if* It is not set to 0
2) *Load* Byte *variable into the* UDR0 *register*

Q 4.5: Construct a program that will allow you to send the character "3" to be printed on a terminal program window every 0.5s infinitely, following these steps:

1) *Call usart_init(ubrr) using the UBRR value calculated in Q2.2*
2) *In the while(1) loop, every 0.5s call the usart_transmit(character) function, where character is the ASCII value for the character "3". You can use <util/delay.h> and _delay_ms(time) to create the 0.5s delay.*

Once your code is complete, we can test it in Proteus. First lets turn-on optimisation for the Atmel Studio project. Go to Project > [ProjectName] Properties > Toolchain > AVR/GNU C Compiler > Optimization, and set the Optimization level to "Optimize –O1". Then make sure to compiled your code in Atmel Studio. Now navigate to your Atmel Studio project folder using Windows Explorer and find the folder called "debug". Inside there will be two compiled files, a ".hex" and a ".elf". These represent the compiled binary of your project code – your actual program, in machine code! When you download your program to the real board, these are the files that would be sent. Proteus can use these files in simulation. In your local lab repository, go to the folder "Lab2_UART\Lab2_Proteus", then open the "Lab2_Proteus.pdsprj" file. You should see something like the following:



Step 1 - Link with Atmel Studio

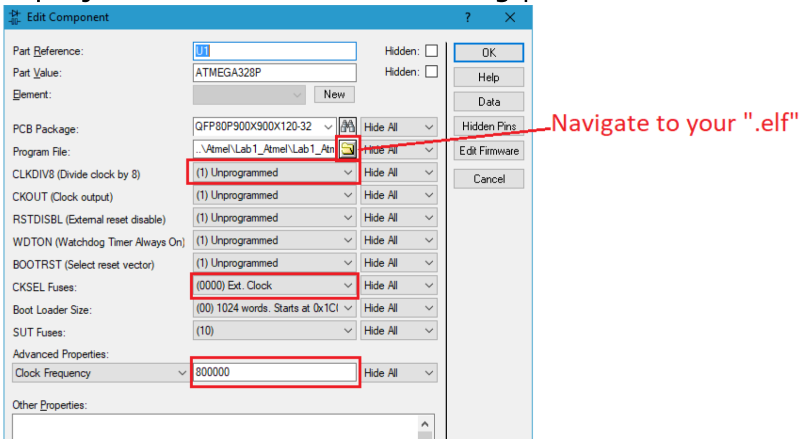
Right click on the microcontroller, and open the "Edit Properties" menu.
Make sure "Program File" is set to your compiled ".elf"
Double check the following parameters (already setup for you):
- CLKDIV8: Set to "(1) Unprogrammed"
- CLKSEL Fuses: Set to "(0000) Ext. Clk"
- Advanced>"Clock Frequency": Set to 800000 (Eight hundred thousand)

Step 2 - Run Simulation

Press "Debug" on the top menu, and select "Run Simulation (F12)"
A text terminal should appear automatically and fill with your program output!
If it doesn't, or if you closed the terminal by accident:
- Re-open terminal by going to"Debug>Virtual Terminal"
Make sure you can see the blinking lights by the TXD and RXD pins!

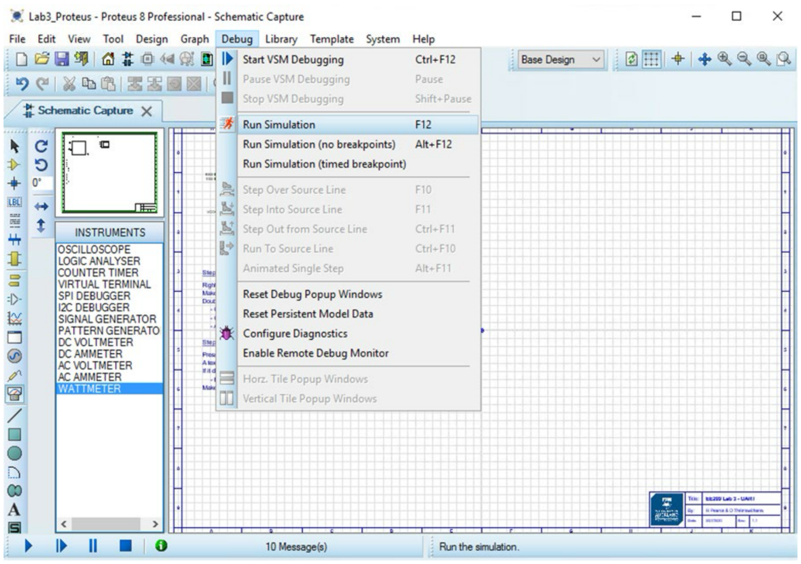
Title: EE209 Lab 3 - UART
By: H Pearce & D Thrimawithana

This Proteus project is made up of only two components, the emulated ATmega328P microcontroller, and the virtual terminal interface. To link the Atmel Project and the Proteus projects, double click on the ATmega328P microcontroller in the Proteus project and ensure the following parameters are set:

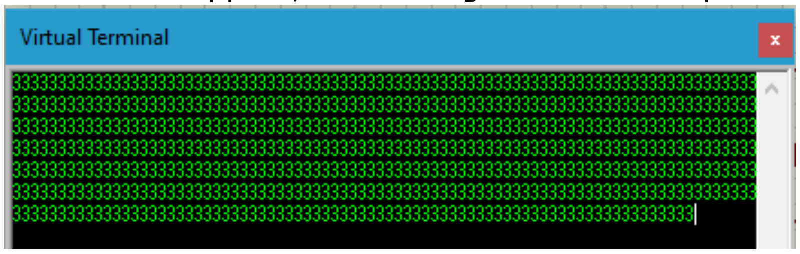


Part Reference:	U1	Hidden: ☐
Part Value:	ATMEGA328P	Hidden: ☐
Element:		
PCB Package:	QFP80P900X900X120-32	Hide All
Program File:	..\Atmel\Lab1_Atmel\Lab1_Atm	Hide All
CLKDIV8 (Divide clock by 8)	(1) Unprogrammed	Hide All
CKOUT (Clock output)	(1) Unprogrammed	Hide All
RSTDISBL (External reset disable)	(1) Unprogrammed	Hide All
WDTON (Watchdog Timer Always On)	(1) Unprogrammed	Hide All
BOOTRST (Select reset vector)	(1) Unprogrammed	Hide All
CKSEL Fuses:	(0000) Ext. Clock	Hide All
Boot Loader Size:	(00) 1024 words. Starts at 0x1C	Hide All
SUT Fuses:	(10)	Hide All
Advanced Properties:		
Clock Frequency	800000	Hide All
Other Properties:		

Program File: Set to your compiled ".elf" in the "debug" directory (from earlier)
– oppose to ".hex" the ".elf" file contains debug information.
CLKDIV8: Set to "(1) Unprogrammed"
CLKSEL Fuses: Set to "(0000) Ext. Clock"
Advanced Properties>Clock Frequency: Set to 2000000 (i.e. 2MHz)
What we have now done is set the virtual microcontroller to run at 2MHz. We choose this speed because the Xplained Mini 328P/328PB boards you will use during this course and in the project is setup to use a 2MHz system clock (as per design specifications).
Note that the Xplained Mini boards can be configured to run at 16MHz (this is in fact the default setting). However, the Xplained Mini boards given to you have been modified according to your design specifications to run at 2MHz (by setting the CKDIV8 fuse bit).
Now let's run the simulation. On the top menu, press Debug>Run Simulation (alternatively you can press F12 or the blue play button found in lower left corner):

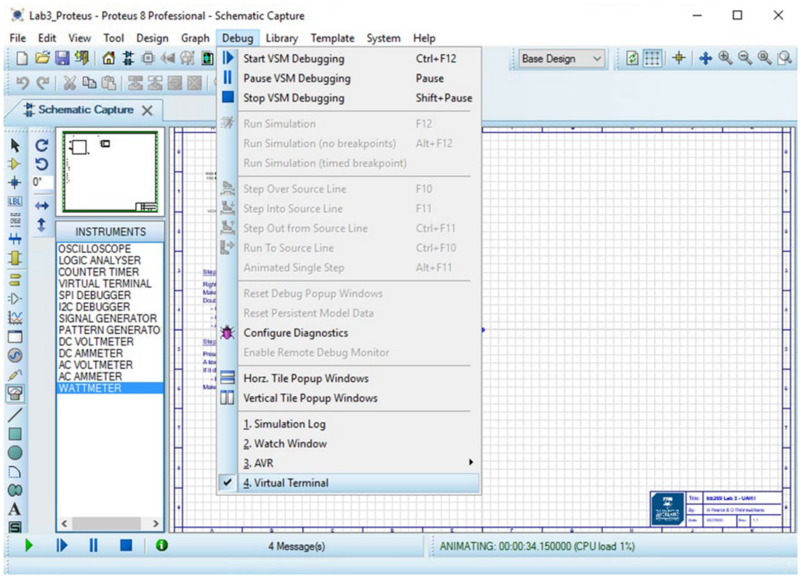

Lab3_Proteus - Proteus 8 Professional - Schematic Capture
File Edit View Tool Design Graph Debug Library Template System Help
Start VSM Debugging Ctrl+F12
Pause VSM Debugging Pause
Stop VSM Debugging Shift+Pause
Run Simulation F12
Run Simulation (no breakpoints) Alt+F12
Run Simulation (timed breakpoint)
Step Over Source Line F10
Step Into Source Line F11
Step Out from Source Line Ctrl+F11
Run To Source Line Ctrl+F10
Animated Single Step Alt+F11
Reset Debug Popup Windows
Reset Persistent Model Data
Configure Diagnostics
Enable Remote Debug Monitor
Horz. Tile Popup Windows
Vertical Tile Popup Windows
Base Design
Schematic Capture
INSTRUMENTS
OSCILLOSCOPE
LOGIC ANALYSER
COUNTER TIMER
VIRTUAL TERMINAL
SPI DEBUGGER
I2C DEBUGGER
SIGNAL GENERATOR
PATTERN GENERATO
DC VOLTMETER
DC AMMETER
AC VOLTMETER
AC AMMETER
WATTMETER
10 Message(s)
Run the simulation.

A terminal window should appear, and the digit "3" should repeatedly fill it:



```
33333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333
```

If your terminal window didn't appear after you start the simulation, or you close it accidentally, then, while a simulation is running, you may recall it by pressing Debug>Virtual Terminal:


Lab3_Proteus - Proteus 8 Professional - Schematic Capture
File Edit View Tool Design Graph Debug Library Template System Help
Start VSM Debugging Ctrl+F12
Pause VSM Debugging Pause
Stop VSM Debugging Shift+Pause
Run Simulation F12
Run Simulation (no breakpoints) Alt+F12
Run Simulation (timed breakpoint)
Step Over Source Line F10
Step Into Source Line F11
Step Out from Source Line Ctrl+F11
Run To Source Line Ctrl+F10
Animated Single Step Alt+F11
Reset Debug Popup Windows
Reset Persistent Model Data
Configure Diagnostics
Enable Remote Debug Monitor
Horz. Tile Popup Windows
Vertical Tile Popup Windows
1. Simulation Log
2. Watch Window
3. AVR
4. Virtual Terminal
Base Design
Schematic Capture
INSTRUMENTS
4 Message(s)
ANIMATING: 00:00:34.150000 (CPU load 1%)

If your terminal is present but empty, check that you've correctly initialised the UART peripheral and setup the transmit function in your Atmel Studio project. If your terminal is showing garbled characters check; baud rate register values are correct, control registers are correctly setup and system clock in Proteus is set to 2000000.

Press Debug>Stop Simulation when you're finished (alternatively you may press the blue stop button found in lower left corner).

BACK TO WORK
Let's do something a bit more complicated now – transmitting a number with multiple digits. We need to separate these out into individual characters, as described in Part 3 of this lab.

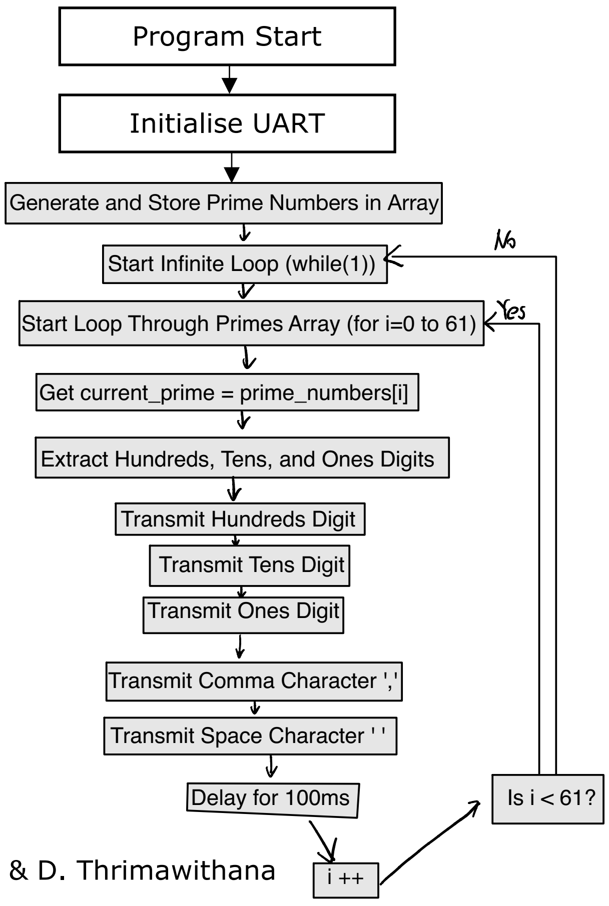
Q 4.6: Modify your program so that you transmit the number "345" to the virtual terminal on Proteus every 0.5s. Inside the while(1) loop, implement your answer from Q3.5 in code, so that the individual 3, 4, and 5 characters are extracted and transmitted to display 345 every 0.5s. If you wish, you may add a space too.

Once you've got this working, it's time to progress to the main task – transmitting a long list of prime numbers! For this lab, we want to transmit each of those numbers so that they appear on a terminal screen, with a comma and a space after each number. To make this task easier, you may print all prime number as 3-digit numbers (e.g., prime number '2', can be printed on terminal as '002')

Q 4.7: Modify your program so that it follows these steps:

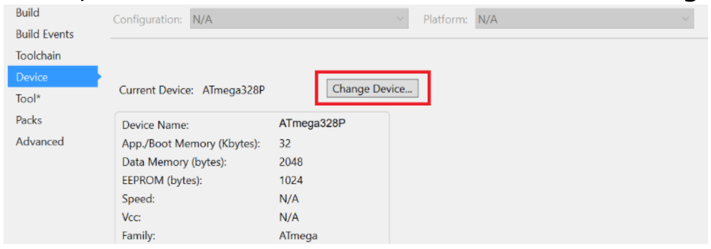
1) *Call usart_init(ubrr) using the UBRR value calculated in Q2.2*
2) *Using the pre-lab code, create an array with the prime numbers*
3) *Inside the while(1) loop, set up a* For Loop
   a. *Extract the individual characters of each prime number*
   b. *Call usart_transmit(character) as needed*
   c. *Call usart_transmit(character) for the comma and space as needed*
   d. *Increment* loop counter Generate and Store Prime Numbers in Array

Q 4.8: Complete the flowchart below that describes the flow of your program code:


Program Start
Initialise UART
Generate and Store Prime Numbers in Array
Start Infinite Loop (while(1))
No
Start Loop Through Primes Array (for i=0 to 61)
Yes
Get current_prime = prime_numbers[i]
Extract Hundreds, Tens, and Ones Digits
Transmit Hundreds Digit
Transmit Tens Digit
Transmit Ones Digit
Transmit Comma Character ','
Transmit Space Character ' '
Delay for 100ms
i ++
Is i < 61?

Part 5: Programming the Xplained Mini

Now that we have developed our program and verified its functionality using Proteus, we are ready to load it on to the Xplained Mini 328P/328PB. Depending on availability, most of you will be provided with a Xplained Mini 328P this year. However, as the semester progress and when we replace damaged boards we may have to issue you a Xplained Mini 328PB. In case you have a Xplained Mini 328PB, remember to recompile the code to run on an ATmega 328PB (so far we have been compiling the code to run on an ATmega328P). To do this, in Atmel/Microchip Studio, go to Project>'Project Name' Properties (here 'Project Name' will be the name you gave your Atmel/Microchip Studio project). A new tab will open and from the menu on left, select "Device" and then click on "Change Device":

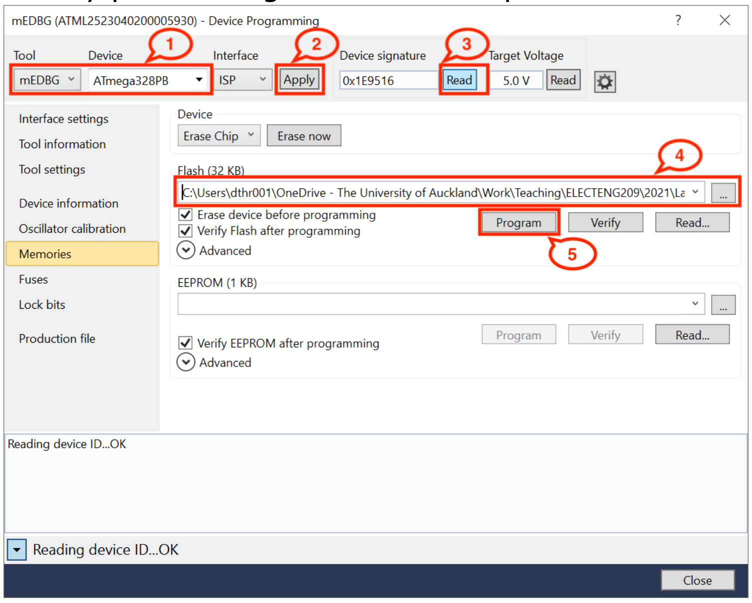


Configuration: N/A
Platform: N/A
Current Device: ATmega328P

Device Name:	ATmega328P
App./Boot Memory (Kbytes):	32
Data Memory (bytes):	2048
EEPROM (bytes):	1024
Speed:	N/A
Vcc:	N/A
Family:	ATmega

A window will open allowing you to pick a microcontroller. Select ATmega328PB as the microcontroller and click "Ok". Now your project is modified so that it will compile the code to run on the Xplained Mini, which has a ATmega328PB on board. Save the project and "Build" your project again.

Connect the Xplained Mini 328P/328PB to your PC using a USB cable. When doing for the first time it may take a few minutes to install drivers. Make sure to wait until drivers are installed to avoid damaging the board. Now to load the recompiled ".elf" file to the ATmega328P/ATmega328PB on the Xplained Mini, bring up the device programmer from Tools>Device Programming. Select "Tool" as "mEDBG …" and "Device" as "ATmega328P" or "ATmega328PB" depending on the device you have. Press "Apply". To check if your Xplained Mini can now communicate with Atmel/Microchip Studio, click "Read" under "Device Signature". If everything is well it will show device signature as a hex number. Now click on "Memories" and check if the file name under "Flash" is set to the ".elf" file (normally this is setup automatically). Finally press "Program". An example is shown here:



mEDBG (ATML2523040200005930) - Device Programming

Tool: mEDBG
Device: ATmega328PB
Interface: ISP
Apply
Device signature: 0x1E9516 Read
Target Voltage: 5.0 V Read

Flash (32 KB): C:\Users\dthr001\OneDrive - The University of Auckland\Work\Teaching\ELECTENG209\2021\La
☑ Erase device before programming
☑ Verify Flash after programming

EEPROM (1 KB)
☑ Verify EEPROM after programming

Reading device ID...OK

Reading device ID...OK

Now on the PC open a terminal program like PuTTY or RealTerm. Setup this terminal program to received the messages sent by the Xplained Mini by telling it the baud rate (i.e., 9600), the number of data bits (i.e., 8), the number of stop bits (i.e., 1) and the serial port number. You can find the serial port number using the "Device Manager". You should be able to now see the prime numbers printed on the terminal.
You should not put the Xplained Mini in to the debugWIRE mode. If you do then stop debug and disable the debugWIRE mode before closing to avoid damaging the Xplaiend Mini.

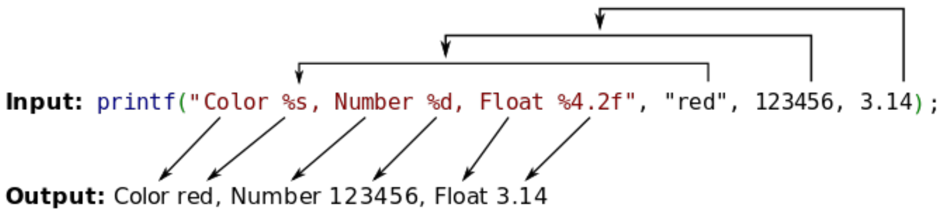
Optional Task: Advanced String Processing

In this lab you broke apart numbers into their individual digits and converted them manually into ASCII. However, this is not the only way to do this process! You may have learned in ENGGEN101 about the *printf*() function in C, stored in **<stdio.h>**, which lets you format strings to the standard output in C, e.g.:

```
#include <stdio.h>

int main() {
      printf("Hello, world!\r\n");
}
```

We can use *printf*, and the closely related *sprintf()* on our microcontroller as well! Let's first have a refresher on how these functions work. Firstly, both printf and sprintf take a *format string* which defines the output, and then a varying number of arguments to be passed in as the values to be placed in the format string. Format specifiers stating with % sign tell where to put these arguments and how to format them. For example:


Input: printf("Color %s, Number %d, Float %4.2f", "red", 123456, 3.14);
Output: Color red, Number 123456, Float 3.14

There are many format specifiers, but the most important are the following:

Specifier	Used for	Example

%c	Individual characters	`char c = 'a';`
		`printf("Char 1: %c", c);`
		➔ `Char 1: a`

%s	Strings of characters	`char *name = "Hammond";`
		`printf("My name is %s", name);`
		➔ `My name is Hammond`

%d	Integers	`int x = 10;`
		`printf("The number is %d", x);`
		➔ `The number is 10`

%f	Floats	`Float e = 2.71828`
		`Printf("Euler's number is %f", e);`
		➔ `Euler's number is 2.71828`

You may also place arguments around the format specifiers for the purposes of rounding as well as leading spaces and zeros on numbers (if you want to produce a neatly formatted table, for instance). The full format specifier prototype is %[flags][width][.precision][length]specifier; and you can find more detail on this at link. Note that Atmel/Microchip Studio is set up to using a lightweight string printing library by default which doesn't support float formatting.
The function *printf* writes the output of the string to your *standard output*. On a computer, this is the terminal. On a microcontroller, we would need to define exactly what our "standard output" is. However, it's easier, and tidier, to use *sprintf* instead. When using *sprintf* the output is written into a character array, which we can then use however we'd like. For instance:

```
//we need to #include <stdio.h>
//and
//#include <string.h>

char* name = "Hammond";
char string_buffer[30]; //we can create strings up to 30 characters long

//this function creates "My name is Hammond" and stores it into string_buffer
sprintf(string_buffer, "My name is %s\n", name);

//we can now send string_buffer to the Uart
for(int i = 0; i < strlen(string_buffer); i++)
{
    USART_Transmit(string_buffer[i]);
}
```

This code will create a nicely formatted string, and save it into the character array called "string_buffer". We can then iterate over that array, sending the characters one by one to our USART peripheral. Note if using above code we also need to include `<string.h>` for the strlen function as indicated in the comments.

Q O.1) Save your code from earlier to a different project. Now, rewrite your code from Q 4.6 to format the prime numbers as strings using sprintf before sending them to the UART. Note that you now no longer need to perform the manual conversion to ASCII – sprintf handles this for you automatically!

Printf and sprintf have a major downside when operating on a simple device like a microcontroller. Consider the complexity that printf/sprintf have inside them in order to achieve all those formatting functions!

Q O.2) Compile both the sprintf and the original version of your code. Now, compare the size of the generated code. Which is larger? __________ By how many times is it larger? ___________

To get signed off for the lab:

- Record all the workings in hardcopy or in digital format so we can give you marks for workings if final answers are not correct
- Modify the simulation models provided as per the lab tasks and save
- Commit and push your saved work to the lab repository on GitHub after completing each task and make sure it is up to date
- Complete this document summarising your final answers, commit and push to lab repository on GitHub
- Update logbook indicating yours and teams progress, meeting notes, etc.
- Go to your assigned interview session to check-in with a TA who will check your solutions and ask a few questions
- Report on your weekly progress