INTRODUCTIORY QUESTIONS

- How do we really communicate between embedded systems?
- What is a communication protocol?
- What do we have to take into account to communicate?

COMMUNICATION PROTOCOLS 1

Daniel Martínez

I7266 - Programación de Sistemas Embebidos

CUCEI - UDG

COMMUNICATION PROTOCOLS

What are they?

WHAT IS “COMMUNICATION”?

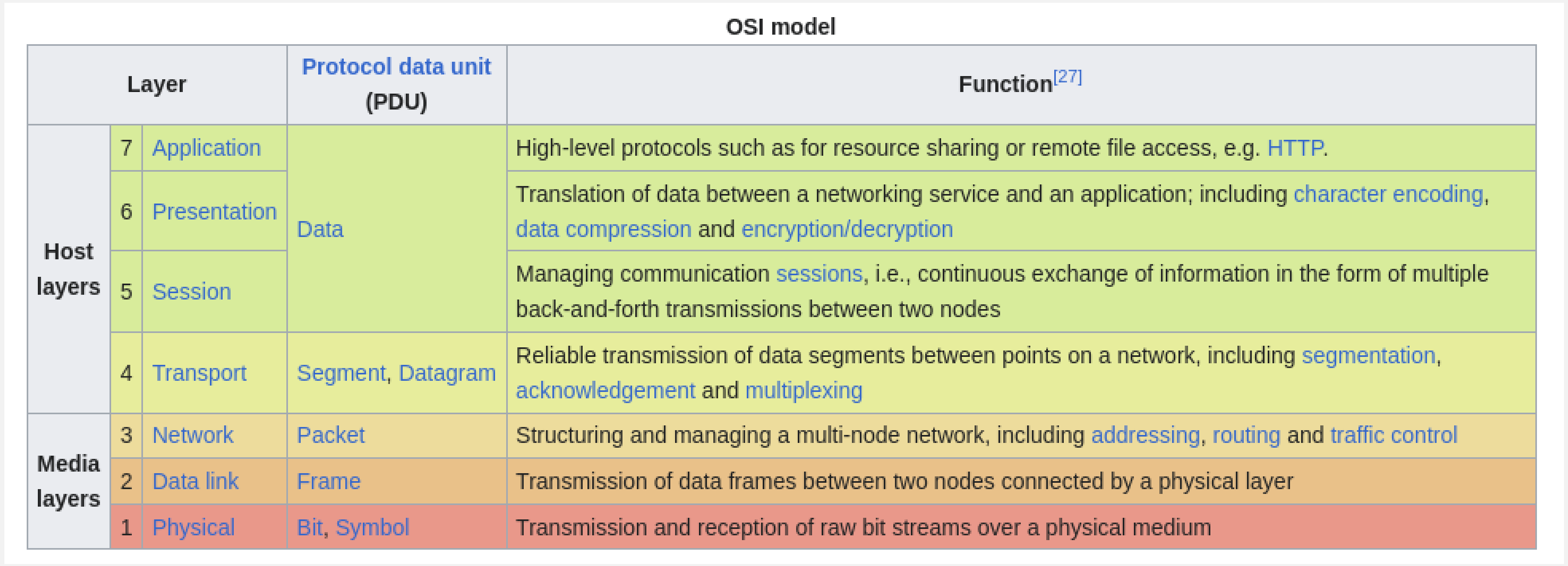
- Communication is usually defined as the exchange of information
  - In the context of microcontrollers, there is an exchange of information ocurring between the internal parts of it, but we usually refer to communication as *external* in the context of embedded systems
- Communication needs some components:
  - These are usually: a sender, a receiver and a medium.

WHAT IS A COMMUNICATION PROTOCOL?

- Apart from the sender, receiver and the medium to communicate, we usually need a set of rules.
  - These rules help us to have an efficient way of communicating
- This set of rules is called a protocol.
- Some of the characteristics are:
  - Synchronous / Asynchronous?
  - Serial / Parallel?
  - Full-Duplex / Half-Duplex?

LAYERS FOR COMMUNICATIONS

- You probably have heard about many parts that comprise the layers that allow communications to coexist.
  - HTTP, HTTPS, TCP/IP, SSH, etc.
- Networks are complex. To keep everything organized we separate its components into layers.
  - Let us analyze the internet and the OSI model.



OSI model

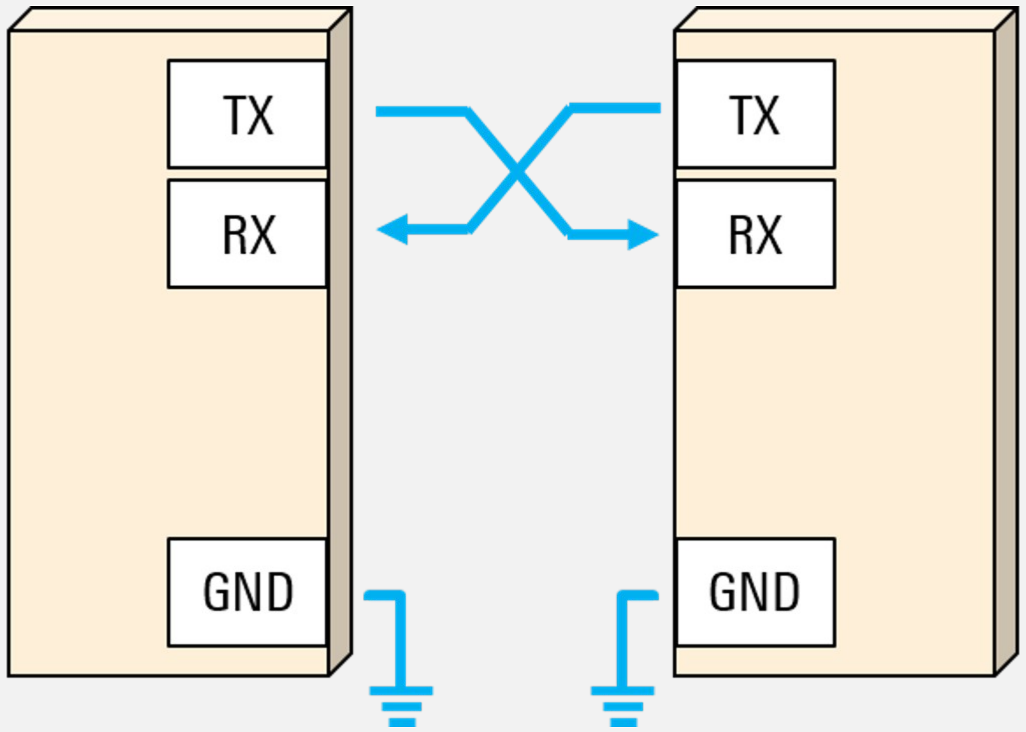
Layer			Protocol data unit (PDU)	Function[27]
Host layers	7	Application	Data	High-level protocols such as for resource sharing or remote file access, e.g. HTTP.
	6	Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption
	5	Session		Managing communication sessions, i.e., continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes
	4	Transport	Segment, Datagram	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing
Media layers	3	Network	Packet	Structuring and managing a multi-node network, including addressing, routing and traffic control
	2	Data link	Frame	Transmission of data frames between two nodes connected by a physical layer
	1	Physical	Bit, Symbol	Transmission and reception of raw bit streams over a physical medium

Taken from Wikipedia.com

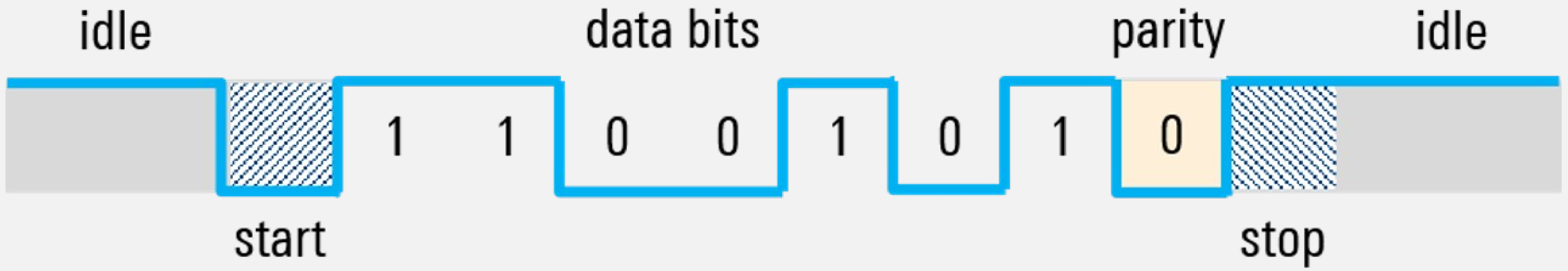
USART

THE USART PROTOCOL

- Stands for:
  - Universal
  - Synchronous
  - Asynchronous
  - Receiver
  - Transmitter



Taken from rohde-schwarz.com



Taken from rohde-schwarz.com

TEAM WORK

20.5 USART Initialization

The USART has to be initialized before any communication can take place. The initialization process normally consists of setting the baud rate, setting frame format and enabling the Transmitter or the Receiver depending on the usage. For interrupt driven USART operation, the Global Interrupt Flag should be cleared (and interrupts globally disabled) when doing the initialization.

Before doing a re-initialization with changed baud rate or frame format, be sure that there are no ongoing transmissions during the period the registers are changed. The TXCn Flag can be used to check that the Transmitter has completed all transfers, and the RXC Flag can be used to check that there are no unread data in the receive buffer. Note that the TXCn Flag must be cleared before each transmission (before UDRn is written) if it is used for this purpose.

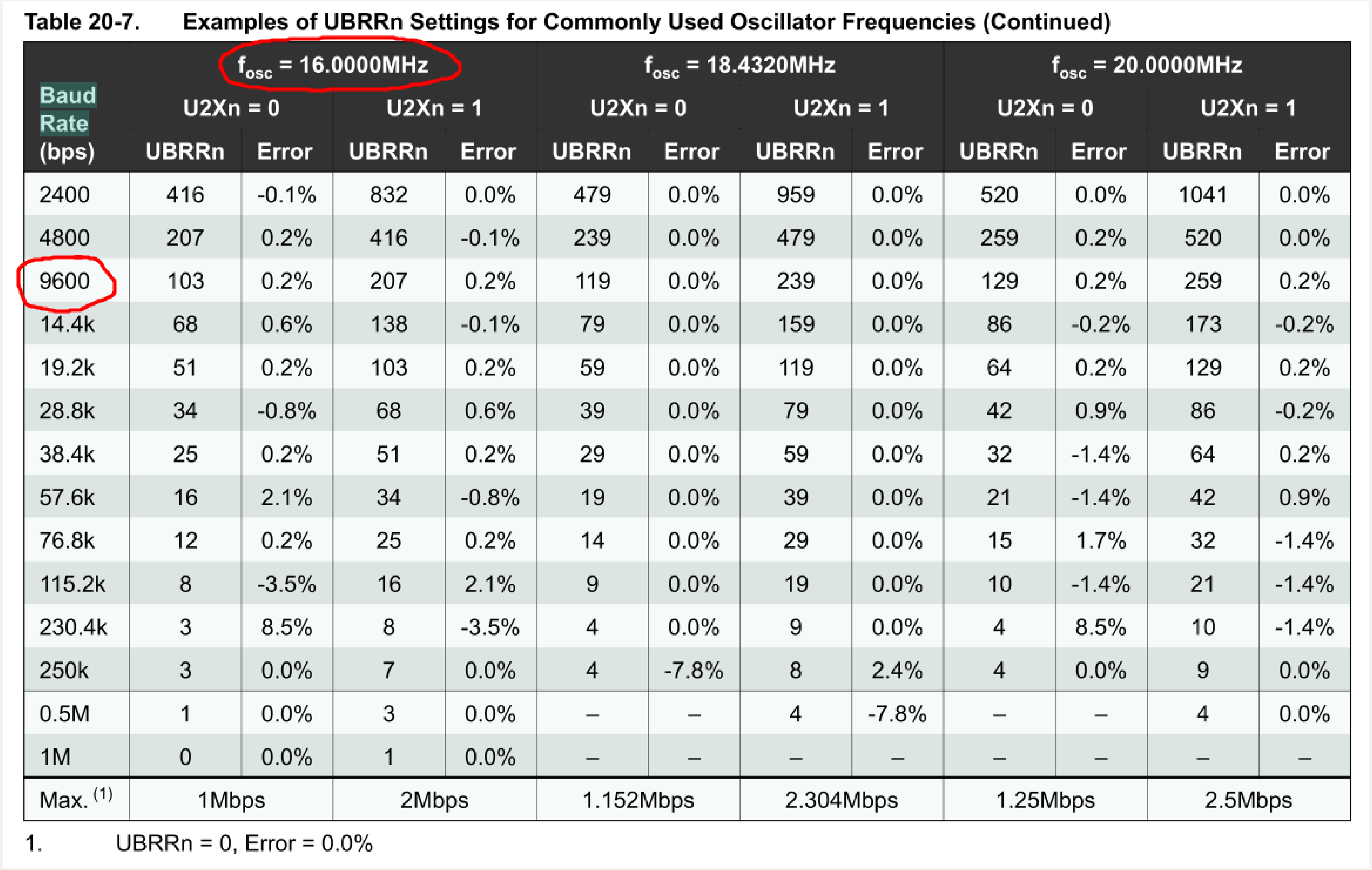


Table 20-7. Examples of UBRRn Settings for Commonly Used Oscillator Frequencies (Continued)

Baud Rate (bps)	f_{osc} = 16.0000MHz				f_{osc} = 18.4320MHz				f_{osc} = 20.0000MHz			
	U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1	
	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error
2400	416	-0.1%	832	0.0%	479	0.0%	959	0.0%	520	0.0%	1041	0.0%
4800	207	0.2%	416	-0.1%	239	0.0%	479	0.0%	259	0.2%	520	0.0%
9600	103	0.2%	207	0.2%	119	0.0%	239	0.0%	129	0.2%	259	0.2%
14.4k	68	0.6%	138	-0.1%	79	0.0%	159	0.0%	86	-0.2%	173	-0.2%
19.2k	51	0.2%	103	0.2%	59	0.0%	119	0.0%	64	0.2%	129	0.2%
28.8k	34	-0.8%	68	0.6%	39	0.0%	79	0.0%	42	0.9%	86	-0.2%
38.4k	25	0.2%	51	0.2%	29	0.0%	59	0.0%	32	-1.4%	64	0.2%
57.6k	16	2.1%	34	-0.8%	19	0.0%	39	0.0%	21	-1.4%	42	0.9%
76.8k	12	0.2%	25	0.2%	14	0.0%	29	0.0%	15	1.7%	32	-1.4%
115.2k	8	-3.5%	16	2.1%	9	0.0%	19	0.0%	10	-1.4%	21	-1.4%
230.4k	3	8.5%	8	-3.5%	4	0.0%	9	0.0%	4	8.5%	10	-1.4%
250k	3	0.0%	7	0.0%	4	-7.8%	8	2.4%	4	0.0%	9	0.0%
0.5M	1	0.0%	3	0.0%	–	–	4	-7.8%	–	–	4	0.0%
1M	0	0.0%	1	0.0%	–	–	–	–	–	–	–	–
Max. [(1)]	1Mbps		2Mbps		1.152Mbps		2.304Mbps		1.25Mbps		2.5Mbps	

1. UBRRn = 0, Error = 0.0%

C Code Example[(1)]

```
#define FOSC 1843200 // Clock Speed
#define BAUD 9600
#define MYUBRR FOSC/16/BAUD-1
void main( void )
{
...
      USART_Init(MYUBRR)
...
}
void USART_Init( unsigned int ubrr)
{
      /*Set baud rate */
      UBRR0H = (unsigned char)(ubrr>>8);
      UBRR0L = (unsigned char)ubrr;
      Enable receiver and transmitter */
      UCSR0B = (1<<RXEN0)|(1<<TXEN0);
      /* Set frame format: 8data, 2stop bit */
      UCSR0C = (1<<USBS0)|(3<<UCSZ00);
}
```

20.3.1 Internal Clock Generation – The Baud Rate Generator

Internal clock generation is used for the asynchronous and the synchronous master modes of operation. The description in this section refers to Figure 20-2.

The USART Baud Rate Register (UBRRn) and the down-counter connected to it function as a programmable prescaler or baud rate generator. The down-counter, running at system clock (f_{osc}), is loaded with the UBRRn value each time the counter has counted down to zero or when the UBRRnL Register is written. A clock is generated each time the counter reaches zero. This clock is the baud rate generator clock output (= f_{osc}/(UBRRn+1)). The Transmitter divides the baud rate generator clock output by 2, 8 or 16 depending on mode. The baud rate generator output is used directly by the Receiver's clock and data recovery units. However, the recovery units use a state machine that uses 2, 8 or 16 states depending on mode set by the state of the UMSELn, U2Xn and DDR_XCKn bits.

USART “LIBRARY”

THERE ARE VARIOUS KINDS

- Libraries, in the context of programming, might mean multiple things
- Mainly, there are pre-compiled static libraries and dynamic libraries
- However, we usually call “library” to a C software module containingone or multiple header files (.h) and, optionally, one or multiple source files (.c)
  - That is not a library, but we are going to create one of these

HOW TO CREATE A C MODULE

The ingredients and recipe

STEP 1

Create the folder structure for a new project

CREATE THE FOLDER STRUCTURE

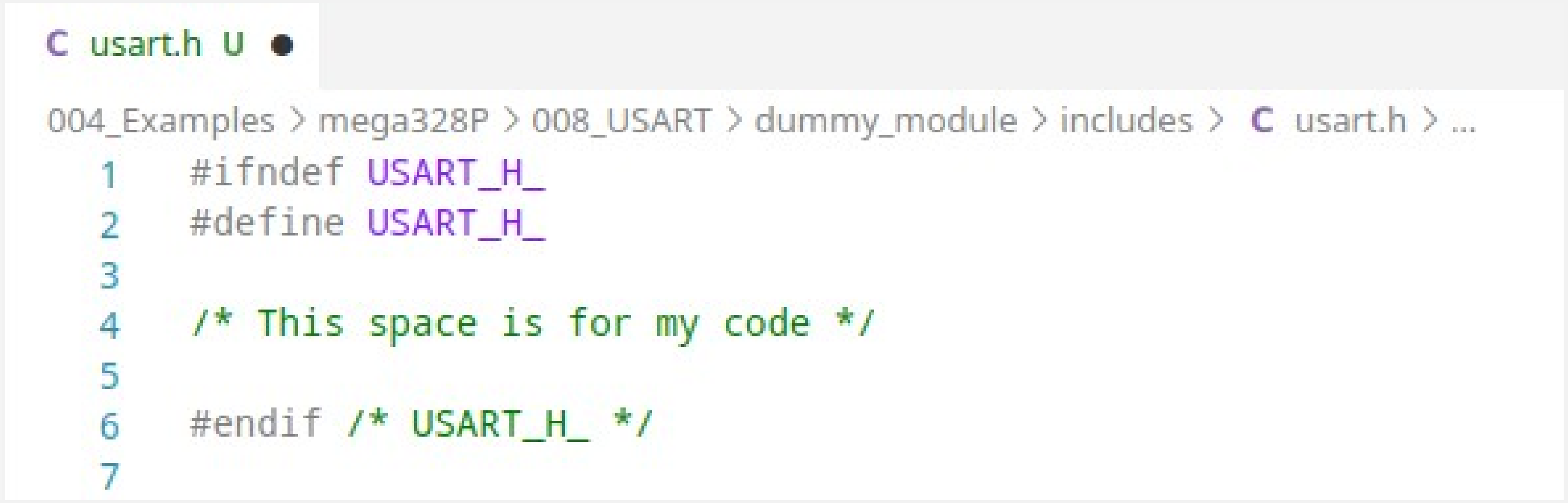
- Choose the path of your preference and create a new folder for your project
  - I created one called “dummy_module”. In this folder, I will create my *main.c* file
- Inside it, create another folder for your module
  - Mine is called “includes” since I will use it to store multiple modules that I will include into my main program

STEP 2

Create the header file

- Inside the “includes” folder, create a new file with the name of your module and the extension “.h”
  - In my case: **usart.h**
- Before writing anything else, create the *ifndef wrapper* or *once-only conditional*.
  - This is to avoid including the same header file multiple times, it is done as follows in the next slides

- Since my header is called ***usart.h***, my wrapper will evaluate if ***USART_H_*** is defined or not


C usart.h U ●
004_Examples > mega328P > 008_USART > dummy_module > includes > C usart.h > ...

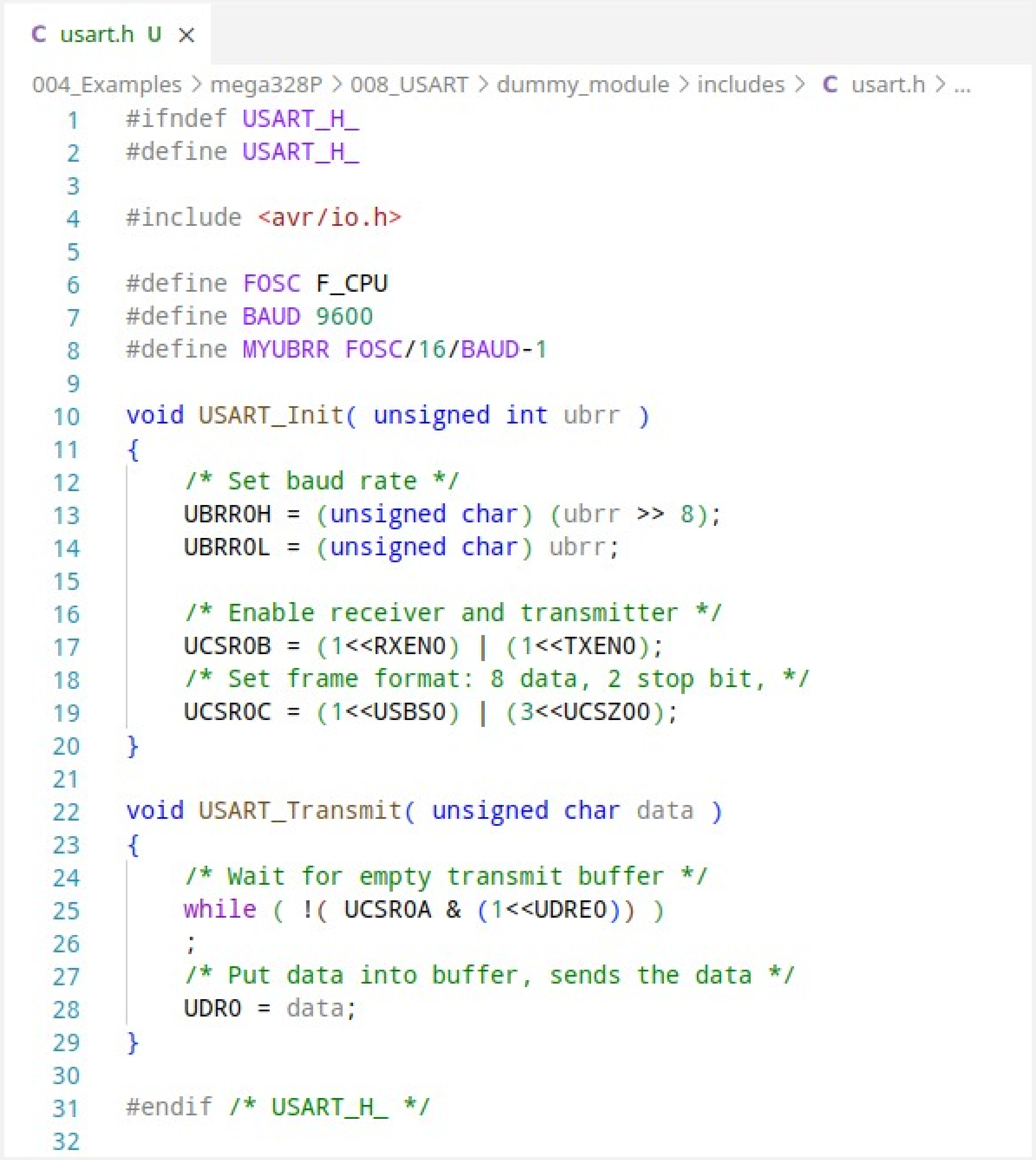
```c
#ifndef USART_H_
#define USART_H_

/* This space is for my code */

#endif /* USART_H_ */

```

- The wrapper does the following:
  - In the first line, the directive ***#ifndef USART_H_*** checks if the macro ***USART_H_*** has <u>not</u> been defined before, being that the case, the following lines are evaluated.
  - In the second line, the directive ***#define USART_H_*** sets the macro ***USART_H_***, so the preprocessor knows for the rest of the code that the file ***usart.h*** has been used already.
  - At the end of the file, the directive ***#endif /* USART_H_ */*** signals the end of the ***#ifndef*** from the first line. The comment is just a good way of telling other people *"this **#endif** corresponds to the one where **USART_H_** is evaluated"*.
- In other words:
  - The wrapper, well, wraps all the code inside the header file, so the contents of the file are **not evaluated** (i.e., they are ignored) if they have already been added before.

STEP 3

Populate the header file

- Rembemer what the preprocessor does to the *includes*?
  - Wherever an ***#include*** is present, the file provided to the directive is “pasted” in that place.
  - This means that if I want to add my USART functions to any other file, all I need to do is define my functions in the ***usart.h*** header file and include it wherever I want to use it.
- Now I will populate my header file


C usart.h U ×

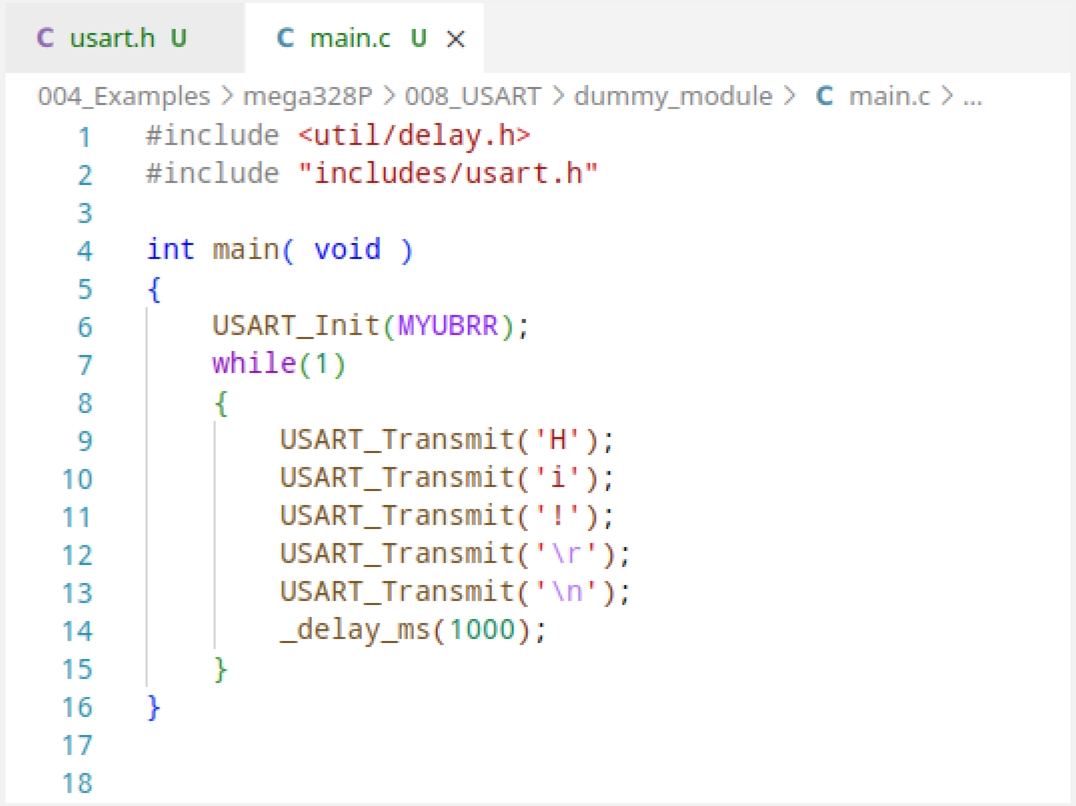
004_Examples > mega328P > 008_USART > dummy_module > includes > C usart.h > ...

```c
#ifndef USART_H_
#define USART_H_

#include <avr/io.h>

#define FOSC F_CPU
#define BAUD 9600
#define MYUBRR FOSC/16/BAUD-1

void USART_Init( unsigned int ubrr )
{
    /* Set baud rate */
    UBRR0H = (unsigned char) (ubrr >> 8);
    UBRR0L = (unsigned char) ubrr;

    /* Enable receiver and transmitter */
    UCSR0B = (1<<RXEN0) | (1<<TXEN0);
    /* Set frame format: 8 data, 2 stop bit, */
    UCSR0C = (1<<USBS0) | (3<<UCSZ00);
}

void USART_Transmit( unsigned char data )
{
    /* Wait for empty transmit buffer */
    while ( !( UCSR0A & (1<<UDRE0)) )
    ;
    /* Put data into buffer, sends the data */
    UDR0 = data;
}

#endif /* USART_H_ */

```

STEP 4

Add it to my main file and compile

- Now I should be ready to use my module
  - I just need to add ***#include*** with the path to my header file between quotes. In my case: ***“includes/usart.h”***
  - **(remember, local includes are wrapped inside double quotes)**
- When I want to use a function or a macro defined in that header file, I can use it just like if it was declared in my main file


C usart.h U
C main.c U
004_Examples > mega328P > 008_USART > dummy_module > C main.c > ...

```c
#include <util/delay.h>
#include "includes/usart.h"

int main( void )
{
    USART_Init(MYUBRR);
    while(1)
    {
        USART_Transmit('H');
        USART_Transmit('i');
        USART_Transmit('!');
        USART_Transmit('\r');
        USART_Transmit('\n');
        _delay_ms(1000);
    }
}


```

- As you might see, this comes with a great advantage:
  - The code looks clean and it is easy to understand.
  - Since there are no declarations in the main file, it is easy to understand its purpose.
  - The code is still there. When I want to see what is going on in the USART functions, I can just go to the header file.
- Let us see if it works
  - I will compile like it any other program and I expect to see no errors, I will use the following command:

```
avr-gcc -mmcu=atmega328p -g -Wall -Os -D F_CPU=16000000 main.c -o main.elf
```

- Just so you remember, this is what each part of the command does.
  - **avr-gcc:** this is the compiler command
  - **-mmcu=atmega328p:** defines the microcontroller we are using
  - **-g:** this argument enables some debugging options
  - **-Wall:** enables all the compiler warnings
  - **-Os:** enables optimization for size
  - **-D F_CPU=16000000:** Sets the clock frequency as a global constant
  - **main.c:** is the input source file
  - **-o main.elf:** defines our desired output file
- These are my personal preferences and the arguments **-g -Wall** are not mandatory.

STEP 5

Upload the program and test it

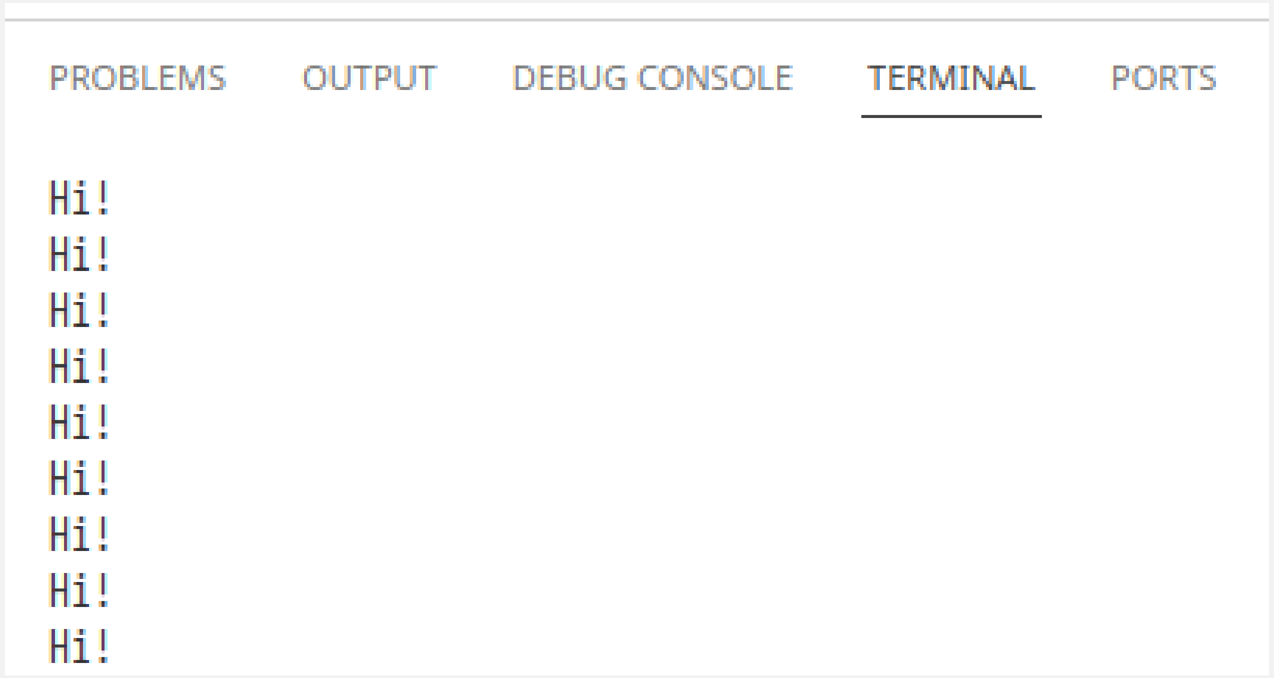
- Once I compiled my code, and prior to uploading it to the microcontroller, I have to convert the **.elf** file to an **.hex** file in *intel hex* format using the following command:

```
avr-objcopy -O ihex main.elf main.hex
```

- And then upload it with the following command:

```
avrdude -p atmega328p -c usbasp -v -U flash:w:main.hex
```

- Once the program has been uploaded, I use the ***screen*** command to read the serial output from my microcontroller.



```
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
```

HOW TO CREATE A C MODULE V2

Corrected and improved

WORKS, BUT CAN BE BETTER

- C modules can provide some options that we couln’t see in this last example.
- From Harvey Mudd College:

  A large C or C++ program should be divided into multiple files. This makes each file short enough to conveniently edit, print, etc. It also allows some of the code, e.g. utility functions such as linked list handlers or array allocation code, to be shared with other programs.

- ChatGPT explains it just fine:

  Splitting a C program with many functions into multiple C files is generally a good practice for multiple reasons:

  - **Modularity:** Each file can focus on a specific aspect of the program. This makes the code easier to understand, maintain, and modify.
  - **Readability:** Developers can quickly locate relevant functions and understand their purpose without having to sift through a large monolithic file.
  - **Encapsulation:** Allows to encapsulate related functions together, as well as limiting the scope of variables.
  - **Compilation Efficiency:** You don't need to recompile the entire program, just the modules you modify. Works best in large projects.
  - **Reusability:** Modularizing your code makes it easier to reuse functions in other projects.
  - **Collaboration:** Different developers can work on different modules concurrently without interfering with each other's code.

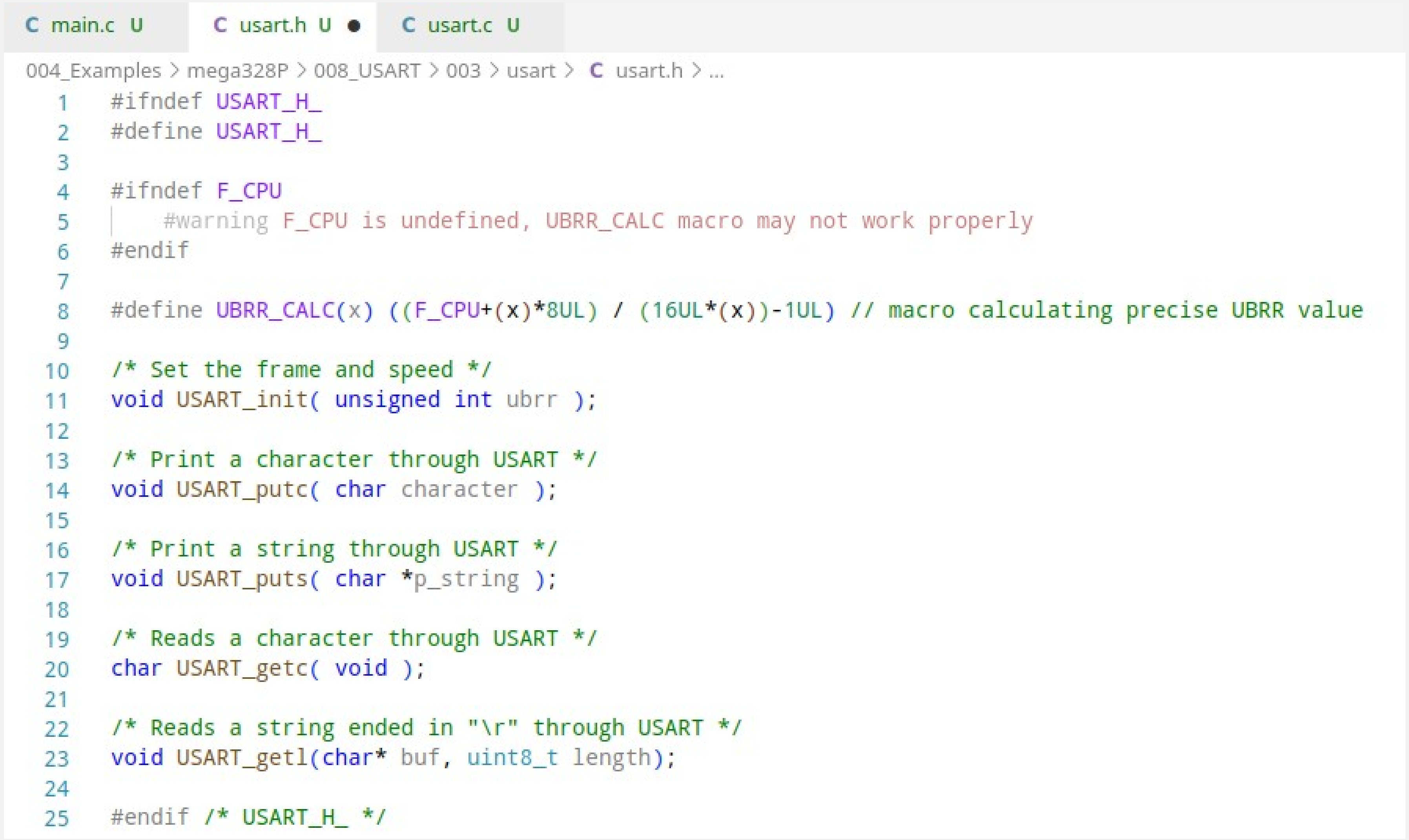
HOW CAN WE MAKE IT BETTER?

- In the last example we defined a header file containing the functions we wanted.
- That means, the code in the header file was integrated into our main file before compiling.
- While this is okay, we can also create multiple C files, each one containing a set of related functionalities.

MY TAKE

A simple USART module

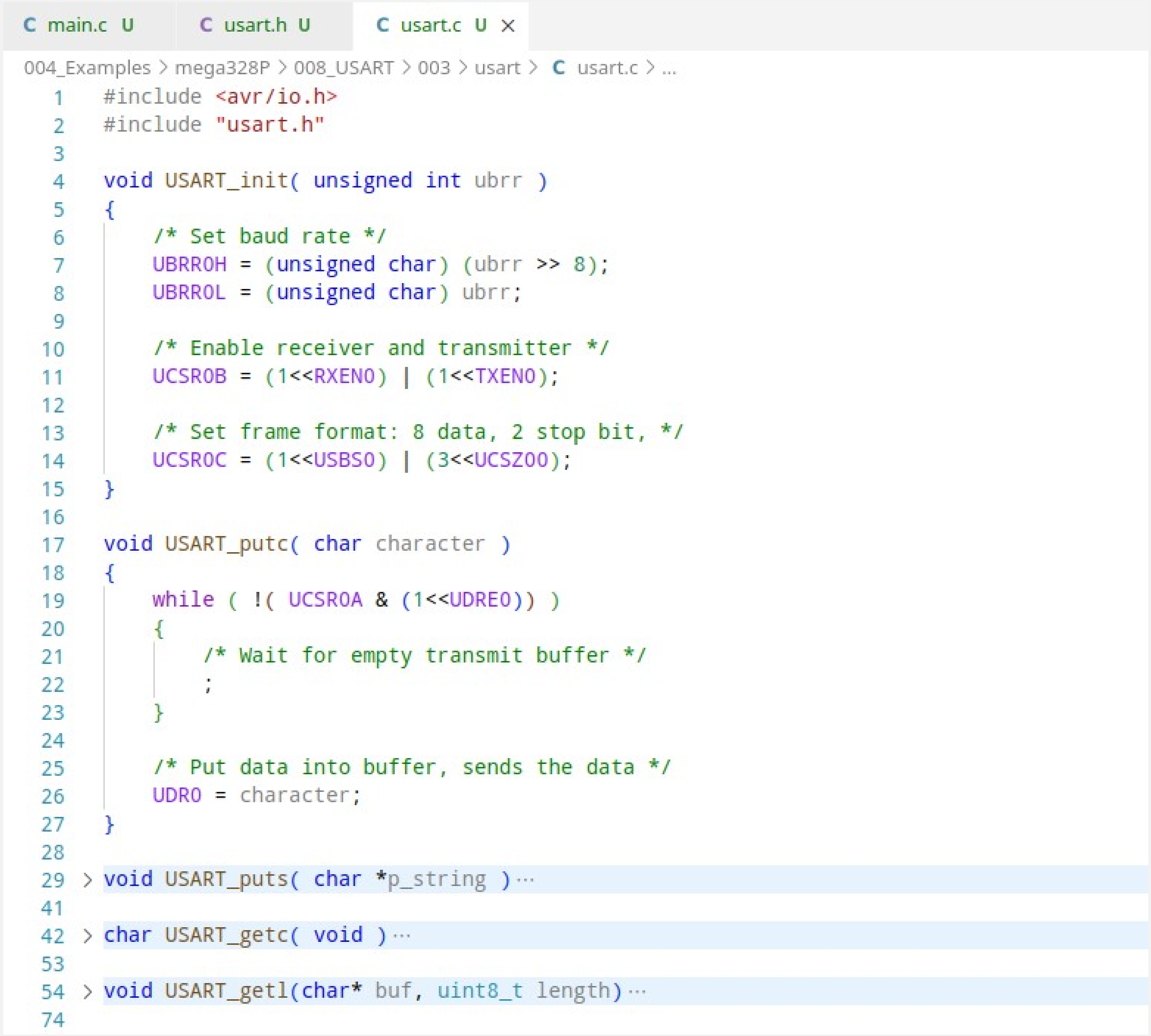
- I will repeat the steps and populate my new header file, but this time it will only contain declarations and prototypes and it will be in the folder **usart**
- I will include the following functions:
  - **USART_init:** to initialize USART communication
  - **USART_putc:** to send a character
  - **USART_puts:** to send a string
  - **USART_getc:** to receive a character
  - **USART_getl:** to receive a string ended in “\r”


C main.c U C usart.h U ● C usart.c U

004_Examples > mega328P > 008_USART > 003 > usart > C usart.h > ...

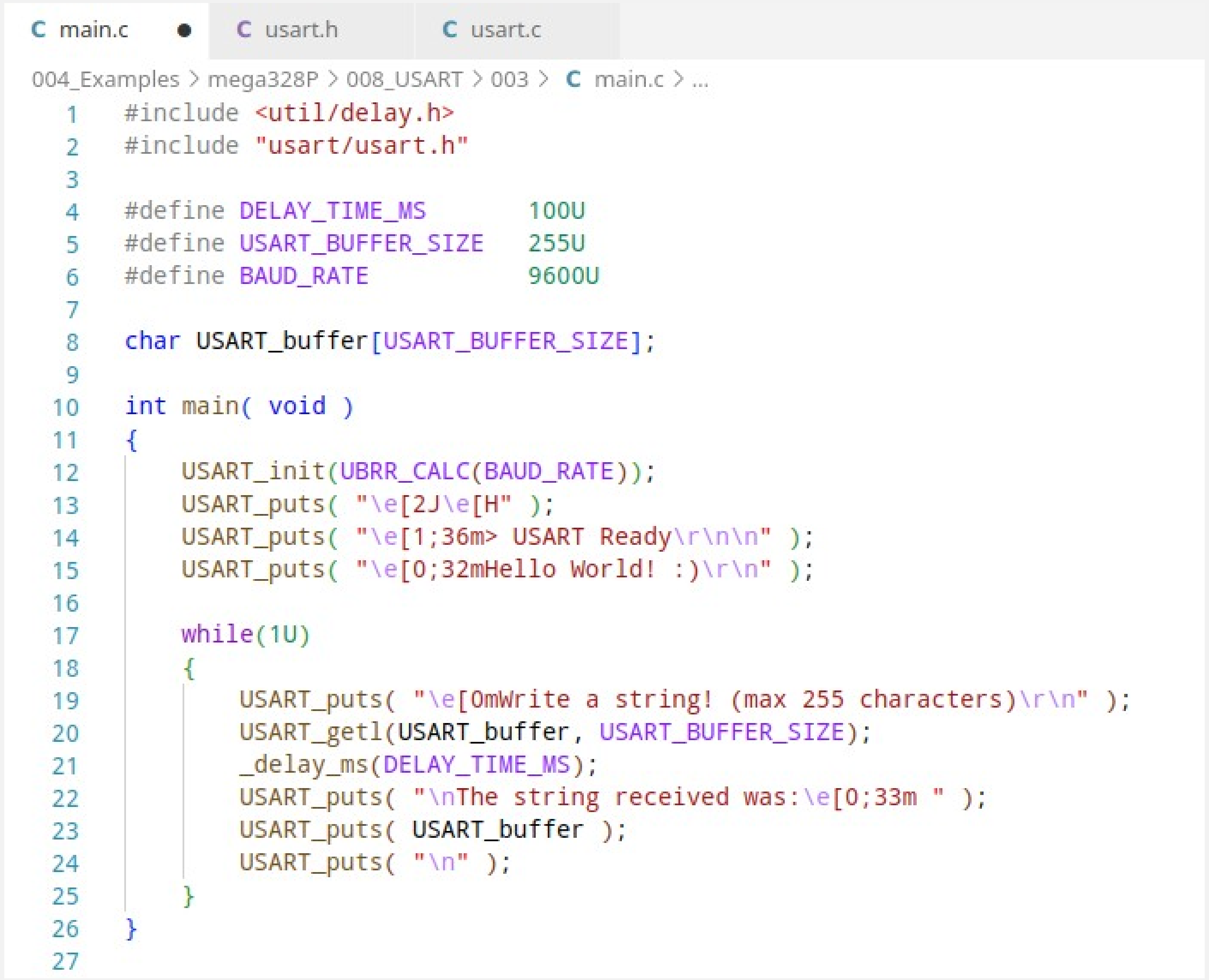
```c
#ifndef USART_H_
#define USART_H_

#ifndef F_CPU
    #warning F_CPU is undefined, UBRR_CALC macro may not work properly
#endif

#define UBRR_CALC(x) ((F_CPU+(x)*8UL) / (16UL*(x))-1UL) // macro calculating precise UBRR value

/* Set the frame and speed */
void USART_init( unsigned int ubrr );

/* Print a character through USART */
void USART_putc( char character );

/* Print a string through USART */
void USART_puts( char *p_string );

/* Reads a character through USART */
char USART_getc( void );

/* Reads a string ended in "\r" through USART */
void USART_getl(char* buf, uint8_t length);

#endif /* USART_H_ */
```

- Now I will define my functions in a C source file with the same name (***usart.c***)
  - And I have to add ***#include*** with the path to my header file between quotes. In this case: ***“usart.h”*** since it is in the same folder as my ***usart.c*** file.
- The reason to add it to the C file is because that is where the definitions exist for the functions I will be defining


C main.c U
C usart.h U
C usart.c U ×
004_Examples > mega328P > 008_USART > 003 > usart > C usart.c > ...

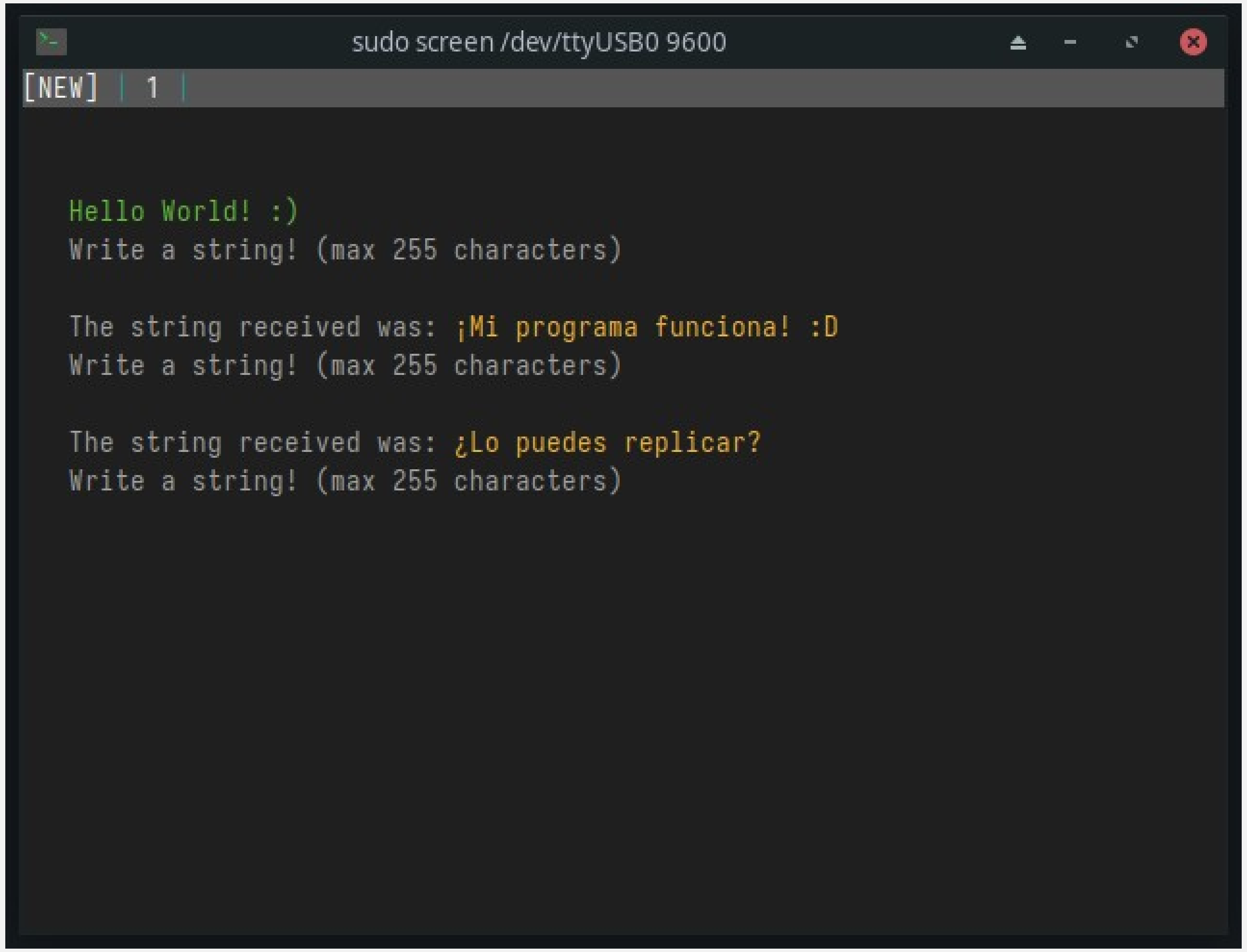
```c
#include <avr/io.h>
#include "usart.h"

void USART_init( unsigned int ubrr )
{
    /* Set baud rate */
    UBRR0H = (unsigned char) (ubrr >> 8);
    UBRR0L = (unsigned char) ubrr;

    /* Enable receiver and transmitter */
    UCSR0B = (1<<RXEN0) | (1<<TXEN0);

    /* Set frame format: 8 data, 2 stop bit, */
    UCSR0C = (1<<USBS0) | (3<<UCSZ00);
}

void USART_putc( char character )
{
    while ( !( UCSR0A & (1<<UDRE0)) )
    {
        /* Wait for empty transmit buffer */
        ;
    }

    /* Put data into buffer, sends the data */
    UDR0 = character;
}

void USART_puts( char *p_string )···

char USART_getc( void )···

void USART_getl(char* buf, uint8_t length)···
```

Finally, my main file:



```c
#include <util/delay.h>
#include "usart/usart.h"

#define DELAY_TIME_MS       100U
#define USART_BUFFER_SIZE   255U
#define BAUD_RATE           9600U

char USART_buffer[USART_BUFFER_SIZE];

int main( void )
{
    USART_init(UBRR_CALC(BAUD_RATE));
    USART_puts( "\e[2J\e[H" );
    USART_puts( "\e[1;36m> USART Ready\r\n\n" );
    USART_puts( "\e[0;32mHello World! :)\r\n" );

    while(1U)
    {
        USART_puts( "\e[0mWrite a string! (max 255 characters)\r\n" );
        USART_getl(USART_buffer, USART_BUFFER_SIZE);
        _delay_ms(DELAY_TIME_MS);
        USART_puts( "\nThe string received was:\e[0;33m " );
        USART_puts( USART_buffer );
        USART_puts( "\n" );
    }
}

```

It asks for a string. After you enter it, it is printed back.



```
sudo screen /dev/ttyUSB0 9600
[NEW] | 1 |

Hello World! :)
Write a string! (max 255 characters)

The string received was: ¡Mi programa funciona! :D
Write a string! (max 255 characters)

The string received was: ¿Lo puedes replicar?
Write a string! (max 255 characters)
```

CAN YOU DO I2C?

THANKS!

Please feel free to ask any related questions at any time.