CG2271 Real Time Operating Systems

Lab 7 – UART Programming

1. Introduction

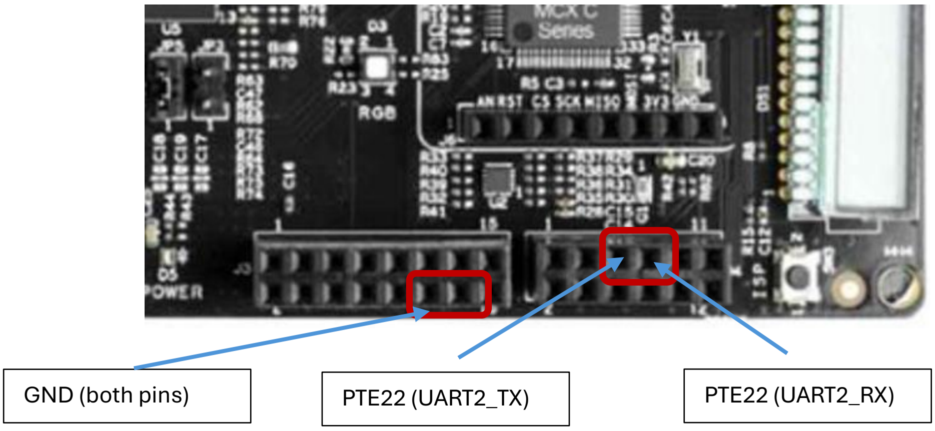
The UART interface is one of several common and important serial interfaces on the MCXC444, the others being SPI (Serial Peripheral Interface) and TWI (Two-Wire Interface). UART is the simplest to program, and in this lab we will explore how to program UART together with FreeRTOS, and use this to integrate the FRDM-MCXC444 with the ESP32, giving the FRDM-MCXC444 some WiFi and Bluetooh capability, as well as allowing us to use devices like the

2. Submission

The report for this lab is due on Friday 31 October 2025 at 2359 hours. Please upload as usual to Canvas.

3. UART Programming on the FRDM-MCXC444

The PTE22 (UART2_TX) and PTE23 (UART2_RX) pins are at the bottom of the FRDM-MCXC444 board, near the LCD:



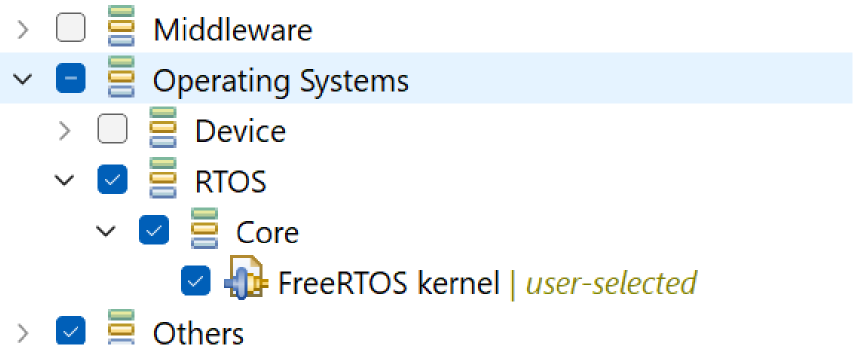
For easy testing:

Connect UART2_RX (PTE23) to UART2_TX (PTE22). This creates what is known as a “local loopback”

The idea behind a local loopback is very simple; we connect TX to RX, so everything that is sent over UARTx_D will be received back on UARTx_D (Recall: Though they have the same name, the transmit end of UARTx_D and and the receive end actually read from and write to different shift registers)

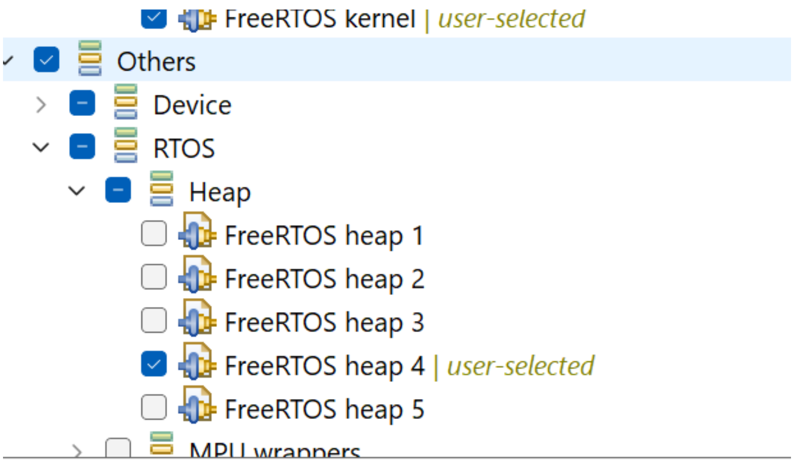
You are provided with a file called “uart_rtos.c”. Create a new C/C++ project called “2271UART” for FreeRTOS, ensuring that in the SDK Components:

i. Under Operating Systems you choose “RTOS”:



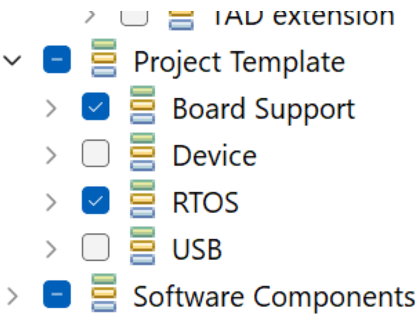
☐ Middleware
⊟ Operating Systems
  ☐ Device
  ☑ RTOS
    ☑ Core
      ☑ FreeRTOS kernel | *user-selected*
☑ Others

ii. Under Others->RTOS->Heap you choose FreeRTOS heap 4 – This is very important. The default heap 5 will hang your code.



☑ FreeRTOS kernel | *user-selected*
☑ Others
  ⊟ Device
  ⊟ RTOS
    ⊟ Heap
      ☐ FreeRTOS heap 1
      ☐ FreeRTOS heap 2
      ☐ FreeRTOS heap 3
      ☑ FreeRTOS heap 4 | *user-selected*
      ☐ FreeRTOS heap 5
    ☐ MPU wrappers

iii. Under Project Template, RTOS is chosen:



iv. Delete all the code in 227UART.c, and copy over all the code from uart_rtos.c

Let’s now examine the code:

i. We add additional includes to bring in our FreeRTOS and task-related calls:

```
#include “FreeRTOS.h”
#include “task.h”
```

ii. Additionally we include queue.h as we are using queues today.

```
#include “queue.h”
```

iii. Our initUART2 is exactly the same as the Lab Lecture, so we will not explain this any further.

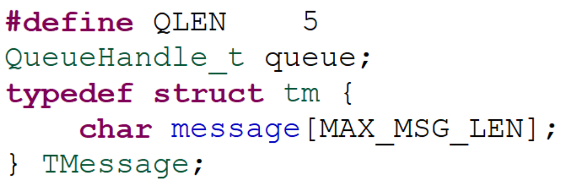
We will use the FreeRTOS queues to pass messages from the UART2_FLEXIO_IRQHandler ISR to a receiver task, which will print messages it receives to the console

Question 1. (3 MARKS) Explain why we pass messages received over queues to a task to print it out, rather than call PRINTF directly inside the ISR.

We begin declaring a send buffer of 256 characters:

```
#define MAX_MSG_LEN     256
char send_buffer[MAX_MSG_LEN];
```

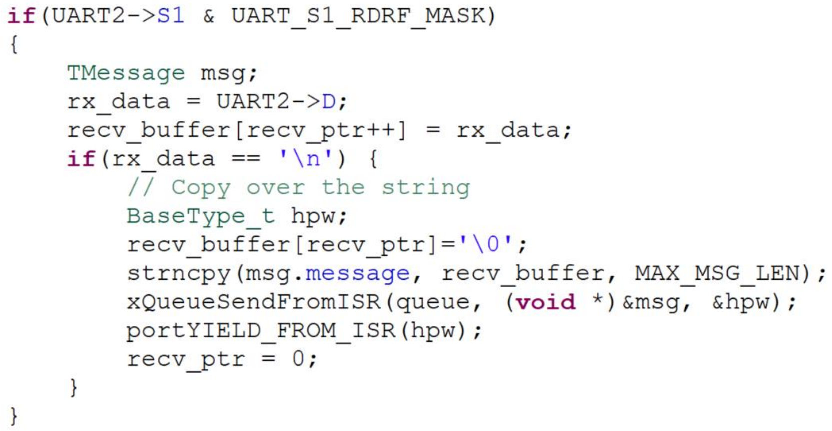
We also declare a FreeRTOS queue, of type QueueHandle_t, a maximum queue length of 5 elements, and a structure for passing messages.



```
#define QLEN     5
QueueHandle_t queue;
typedef struct tm {
    char message[MAX_MSG_LEN];
} TMessage;
```

Processing Received Data

Let’s examine the receive part of the ISR:



```
if(UART2->S1 & UART_S1_RDRF_MASK)
{
    TMessage msg;
    rx_data = UART2->D;
    recv_buffer[recv_ptr++] = rx_data;
    if(rx_data == '\n') {
        // Copy over the string
        BaseType_t hpw;
        recv_buffer[recv_ptr]='\0';
        strncpy(msg.message, recv_buffer, MAX_MSG_LEN);
        xQueueSendFromISR(queue, (void *)&msg, &hpw);
        portYIELD_FROM_ISR(hpw);
        recv_ptr = 0;
    }
}
```

Question 2. (2 MARKS) Why do we use:

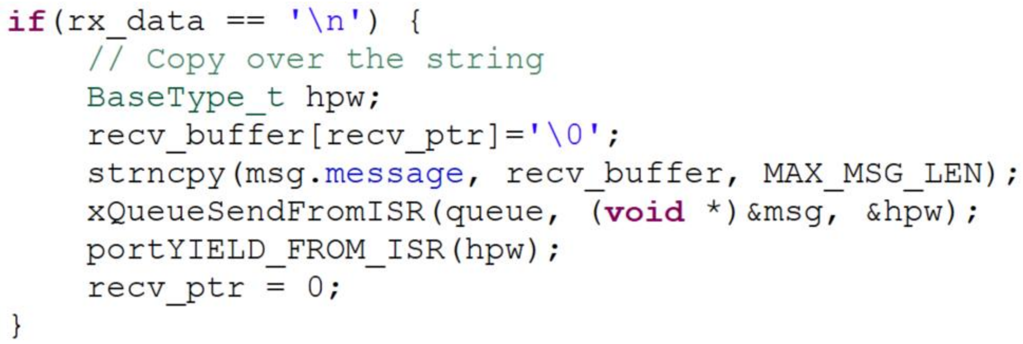
if(UART1->S1 & UART_S1_RDRF_MASK)

to test for whether we have received data? Explain how this works.

Question 3. (3 MARKS) In past labs where we tested flags in ISRs, we reset the flag by writing a 1 to it. Explain why, in this case, there is no need to reset the RDRF flag.

In this code we first read from the UART2 data register UART2->D and store it into rx_data, then we put it into the recv_buffer, using recv_ptr to tell us where to insert the next character to.

We come to this part of the receive part of the USR:



```
if(rx_data == '\n') {
    // Copy over the string
    BaseType_t hpw;
    recv_buffer[recv_ptr]='\0';
    strncpy(msg.message, recv_buffer, MAX_MSG_LEN);
    xQueueSendFromISR(queue, (void *)&msg, &hpw);
    portYIELD_FROM_ISR(hpw);
    recv_ptr = 0;
}
```

Here we detect the carriage return at the end of the line, then:

i. Properly NULL-terminate recv_buffer by writing '\0' to the end of the buffer.

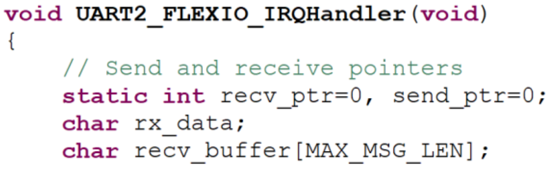
ii. Copy the buffer into a variable msg of type TMessage.

iii. Call xQueueSendFromISR with the queue, a pointer to msg, and a pointer to hpw of type BaseType_t. From our earlier examination of xSemaphoreGiveFrromISR, we know that the hpw flag is set to pdTRUE if the queue send results in a higher priority task becoming READY.

iv. We pass the hpw flag to portYIELD_FROM_ISR . If hpw is pdTRUE, FreeRTOS will prepare to context switch to the higher priority task once the ISR exits.

Question 4. (5 MARKS) do not pass a pointer to recv_buffer to xQueueSendFromISR because recv_buffer is a local variable, and when the ISR exits, the contents are not guaranteed to be correct. We instead copy recv_buffer into msg and send msg over. Why does this work? (Hint: As per the documentation at https://www.freertos.org/Documentation/02-Kernel/04-API-references/06-Queues/04-xQueueSendFromISR, items are queued by copy and not reference)

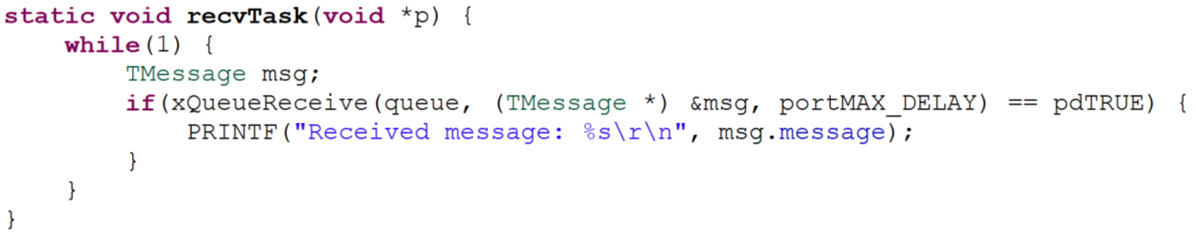
Question 5. (2 MARKS) Looking at the start of the ISR we see that send_ptr and recv_ptr are declared as static int:



```
void UART2_FLEXIO_IRQHandler(void)
{
    // Send and receive pointers
    static int recv_ptr=0, send_ptr=0;
    char rx_data;
    char recv_buffer[MAX_MSG_LEN];
```

Explain what "static" means and why it is important here.

To understand this ISR further, let's look at the corresponding recvTask:



```
static void recvTask(void *p) {
    while(1) {
        TMessage msg;
        if(xQueueReceive(queue, (TMessage *) &msg, portMAX_DELAY) == pdTRUE) {
            PRINTF("Received message: %s\r\n", msg.message);
        }
    }
}
```

We see that this task calls xQueueReceive, which will block until the ISR sends the message using xQueueSendFromISR.

Sending Data

Let's now look at how we send data. Firstly notice that in initUART2, we enable the receiver (and receiver interrupt), but NOT the transmitter!

```
//Enable RX interrupt
UART2->C2 |= UART_C2_RIE_MASK;

// Enable the receiver
UART2->C2 |= UART_C2_RE_MASK;
```

This is because we DO NOT want the transmit interrupt to trigger unless we have data to transmit.

We create a dummy sendTask to continually send data over the UART every 2 seconds:

```
static void sendTask(void *p) {
    int count=0;
    char buffer[MAX_MSG_LEN];
    while(1) {
        sprintf(buffer, "This is message %d\n", count++);
        sendMessage(buffer);
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

This task calls send_message, which copies the message over to send_buffer, and switches on the transmitter and transmitter interrupt:

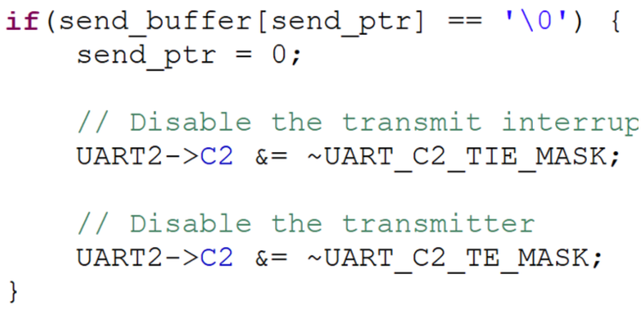
```
void sendMessage(char *message) {
    strncpy(send_buffer, message, MAX_MSG_LEN);

    // Enable the TIE interrupt
    UART2->C2 |= UART_C2_TIE_MASK;

    // Enable the transmitter
    UART2->C2 |= UART_C2_TE_MASK;
}
```

Question 6. (2 MARKS) Explain why we use strncpy and not strcpy to copy the string over.

Finally let’s look at the send portion of the ISR:

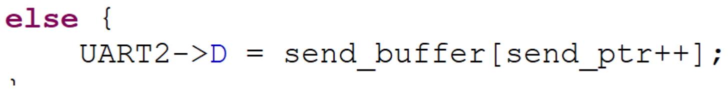
```
if(UART2->S1 & UART_S1_TDRE_MASK) // Send data
{
    if(send_buffer[send_ptr] == '\0') {
        send_ptr = 0;

        // Disable the transmit interrupt
        UART2->C2 &= ~UART_C2_TIE_MASK;

        // Disable the transmitter
        UART2->C2 &= ~UART_C2_TE_MASK;
    }
    else {
        UART2->D = send_buffer[send_ptr++];
    }
}
```

We see that when we detect the transmit interrupt, this part checks if we have reached the end of the string:



```
if(send_buffer[send_ptr] == '\0') {
    send_ptr = 0;

    // Disable the transmit interrup
    UART2->C2 &= ~UART_C2_TIE_MASK;

    // Disable the transmitter
    UART2->C2 &= ~UART_C2_TE_MASK;
}
```

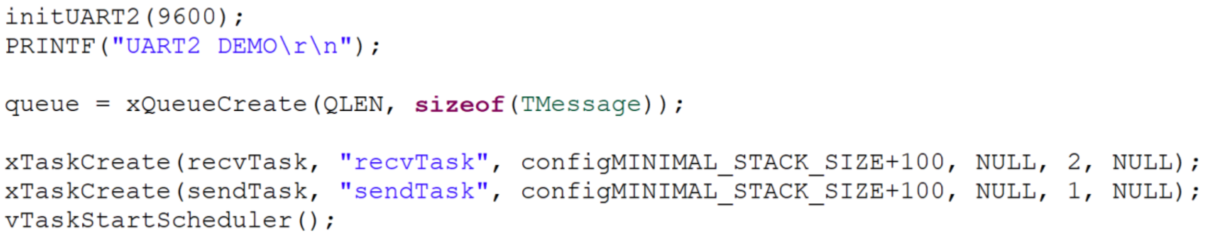
If so, we reset send_ptr, and turn off the transmitter interrupt and the transmitter itself. Otherwise we do:



```
else {
    UART2->D = send_buffer[send_ptr++];
```

Which sends the next character over the UART by writing to UART2->D, which is the data register to hold the data to be sent.

Finally we look at main, where we call iniUART2 to initialize UART2 to 9600 bps, create the queue, the send and receive tasks, and start the RTOS:



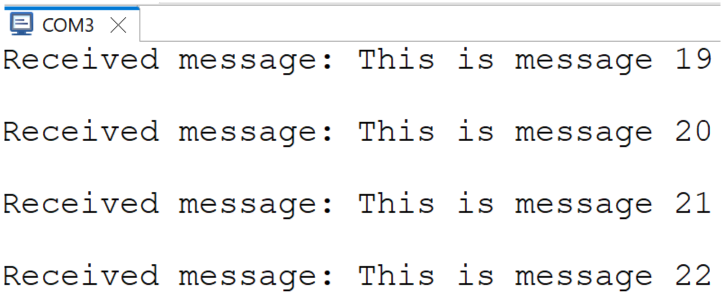
```
initUART2(9600);
PRINTF("UART2 DEMO\r\n");

queue = xQueueCreate(QLEN, sizeof(TMessage));

xTaskCreate(recvTask, "recvTask", configMINIMAL_STACK_SIZE+100, NULL, 2, NULL);
xTaskCreate(sendTask, "sendTask", configMINIMAL_STACK_SIZE+100, NULL, 1, NULL);
vTaskStartScheduler();
```

Notice that xQueueCreate takes just two arguments; the maximum length of the queue, and the size of each queue item.

Now build and debug the project, and open a new UART terminal to monitor the output of the project. You will see the board output this line every 2 seconds:

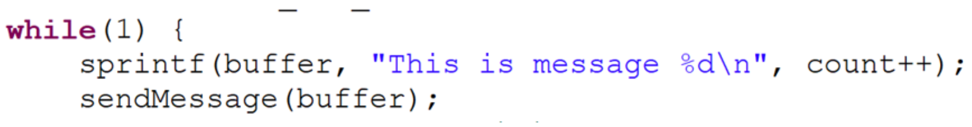
Received Message: This is message xx

Here xx is a running count. The actual output looks like this:



```
Received message: This is message 19

Received message: This is message 20

Received message: This is message 21

Received message: This is message 22
```

Notice that “This is message xx” is actually sent by sendTask:



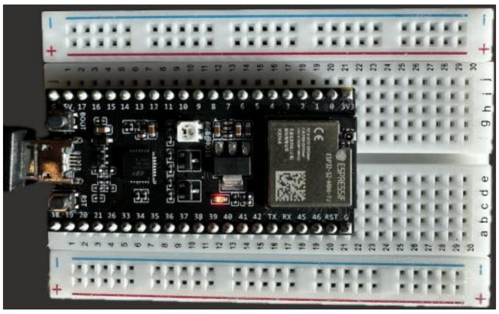
```
while(1) {
    sprintf(buffer, "This is message %d\n", count++);
    sendMessage(buffer);
```

However it is being received by UART2 because of the jumper that we connected between PTE22 and PTE23 (UART2_TX to UART2_RX).

4. Interfacing with the Espressif DevKitM-1 Board

   The Espressif ESP32-S2 DevkitM-1 board is based on the ESP32 chip, which combines a powerful single-core XTensa 240 MHz microcontroller with WiFi (but no bluetooth) capability. For comparison the MCXC444 is single-core clocked at 48 MHz, though the MCXC444 comes with many more features like Direct Memory Access and a true digital to analog converter (not PWM but an actual DAC). Additionally ARM is far more commonly used than ESP32 for commercial applications.

   The board comes in a 42-pin dual-in-line package (DIP) format which sits just nicely on a single breadboard, leaving one row of pins free on both sides as shown below:

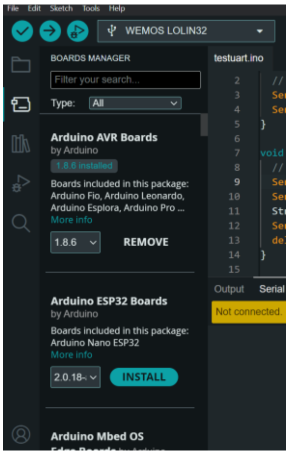


While you will have access to all pins, you will need to mount any components you want to use on a separate breadboard.

i. Setting up the Arduino IDE

   a. Download and install the latest version of the Arduino IDE.

   b. Go to Tools->Board->Boards Manager and click on it. You will get a dialog box on the left like this:

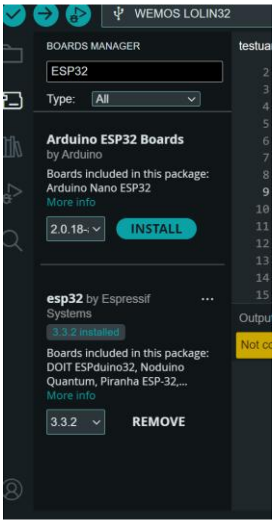


Arduino AVR Boards
by Arduino
1.8.6 installed
Boards included in this package:
Arduino Fio, Arduino Leonardo, Arduino Esplora, Arduino Pro ...
More info
1.8.6 REMOVE

Arduino ESP32 Boards
by Arduino
Boards included in this package:
Arduino Nano ESP32
More info
2.0.18- INSTALL

Arduino Mbed OS

c. In the "Filter your search" box, look for "ESP32":



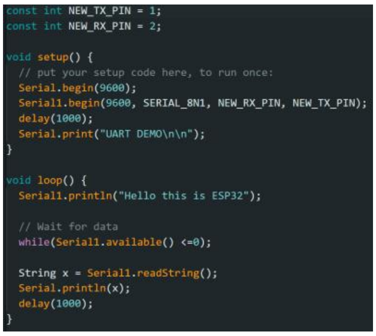
ESP32

Arduino ESP32 Boards
by Arduino
Boards included in this package:
Arduino Nano ESP32
More info
2.0.18- INSTALL

esp32 by Espressif Systems
3.3.2 installed
Boards included in this package:
DOIT ESPduino32, Noduino Quantum, Piranha ESP-32,...
More info
3.3.2 REMOVE

d. Choose esp32 by Espressif and click INSTALL (here it says "Remove" but you should see INSTALL in a green button if you've never installed this board before). Note: DO NOT install "Arduino ESP32 Boards" as these are not compatible with the Lolin32 boards we are using.

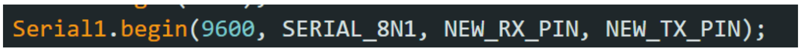
ii. Navigate to where you've unzipped the Lab 7 files to, and open the testuart Arduino Sketch. You will see code like this:



```
const int NEW_TX_PIN = 1;
const int NEW_RX_PIN = 2;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial1.begin(9600, SERIAL_8N1, NEW_RX_PIN, NEW_TX_PIN);
  delay(1000);
  Serial.print("UART DEMO\n\n");
}

void loop() {
  Serial1.println("Hello this is ESP32");

  // Wait for data
  while(Serial1.available() <=0);

  String x = Serial1.readString();
  Serial.println(x);
  delay(1000);
}
```

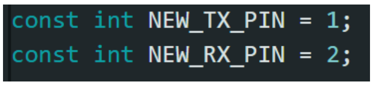
Note how the ESP32 S2 has two UART ports. UART0 (Serial) goes to the USB interface and you can Serial.Print information to the Serial Monitor.

UART1 is free for our use but we need to “remap” the current UART pins as UART1_RX is used by the LED (see below). This is done at this line when we initialize UART1:



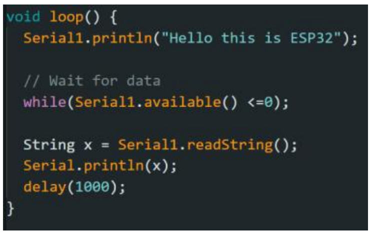
```
Serial1.begin(9600, SERIAL_8N1, NEW_RX_PIN, NEW_TX_PIN);
```

This line sets the bit rate (9600 bps), the frame format (8N1), then the new TX and RX pins, which were earlier defined:



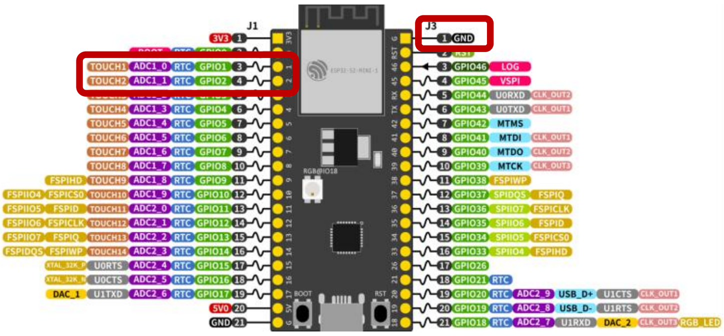
```
const int NEW_TX_PIN = 1;
const int NEW_RX_PIN = 2;
```

Other than this, the code is quite straightforward. It sends “Hello this is ESP32” to the MCXC444, then waits for a reply, which it reads and prints. One second later we repeat.



```
void loop() {
  Serial1.println("Hello this is ESP32");

  // Wait for data
  while(Serial1.available() <=0);

  String x = Serial1.readString();
  Serial.println(x);
  delay(1000);
}
```

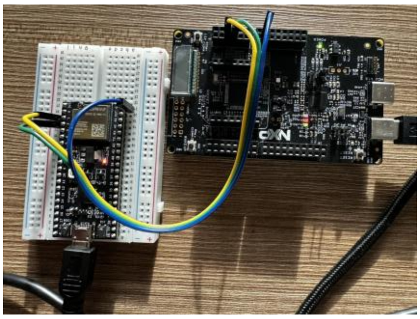
iii. The pinout for the ESP32-S2 DevKitM-1 is shown below:


J1
3V3 1
TOUCH1 ADC1_0 RTC GPIO1 3
TOUCH2 ADC1_1 RTC GPIO2 4
TOUCH4 ADC1_3 RTC GPIO4 6
TOUCH5 ADC1_4 RTC GPIO5 7
TOUCH6 ADC1_5 RTC GPIO6 8
TOUCH7 ADC1_6 RTC GPIO7 9
TOUCH8 ADC1_7 RTC GPIO8 10
FSPIHD TOUCH9 ADC1_8 RTC GPIO9 11
FSPIIO4 FSPICS0 TOUCH10 ADC1_9 RTC GPIO10 12
FSPIIO5 FSPID TOUCH11 ADC2_0 RTC GPIO11 13
FSPIIO6 FSPICLK TOUCH12 ADC2_1 RTC GPIO12 14
FSPIIO7 FSPIQ TOUCH13 ADC2_2 RTC GPIO13 15
FSPIDQS FSPIWP TOUCH14 ADC2_3 RTC GPIO14 16
XTAL_32K_P U0RTS ADC2_4 RTC GPIO15 17
XTAL_32K_N U0CTS ADC2_5 RTC GPIO16 18
DAC_1 U1TXD ADC2_6 RTC GPIO17 19
5V0 20
GND 21
ESP32-S2-MINI-1
RGB@IO18
BOOT
RST
J3
1 GND
3 GPIO46 LOG
4 GPIO45 VSPI
5 GPIO44 U0RXD CLK_OUT2
6 GPIO43 U0TXD CLK_OUT1
7 GPIO42 MTMS
8 GPIO41 MTDI CLK_OUT1
9 GPIO40 MTDO CLK_OUT2
10 GPIO39 MTCK CLK_OUT3
11 GPIO38 FSPIWP
12 GPIO37 SPIDQS FSPIQ
13 GPIO36 SPIIO7 FSPICLK
14 GPIO35 SPIIO6 FSPID
15 GPIO34 SPIIO5 FSPICS0
16 GPIO33 SPIIO4 FSPIHD
17 GPIO26
18 GPIO21 RTC
19 GPIO20 RTC ADC2_9 USB_D+ U1CTS CLK_OUT1
20 GPIO19 RTC ADC2_8 USB_D- U1RTS CLK_OUT2
21 GPIO18 RTC ADC2_7 U1RXD DAC_2 CLK_OUT3 RGB_LED

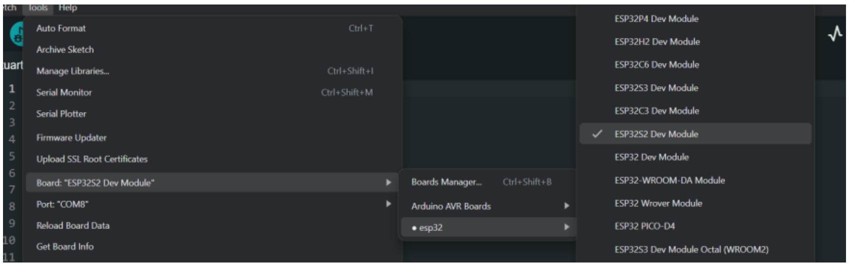
As seen here UART1 TX is on pin 19 on the left (GPIO17) and UART1 RX is on pin 21 on the right (GPIO18). However note that GPIO18 is connected to the RGB LED. This interferes with UART1 working correctly (and annoyingly lights up the LED very brightly). We need to remap GPIO17 to GPIO1 and GPIO18 to GPIO2. GPIO1 and GPIO2 (and GND) are circled above.

Connect PTE22 on the MCXC444 to GPIO2 on the ESP32, and PTE23 to GPIO3. Also connect GND on the MCXC444 to GND on the ESP32. See the picture on Page 1 to locate PTE22, PTE23 and GND on the MCXC444 board.

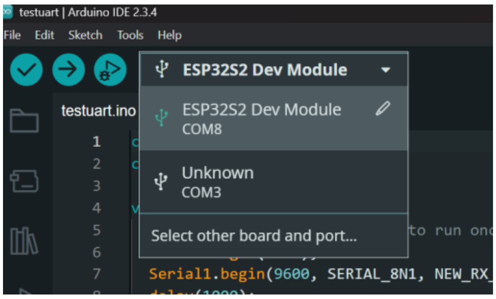
iv. Your connection should look something like this:



v. Connect the ESP32-S2 board to your computer, then click Tools->Board->esp32->ESP32S Dev Module and select it as our board.



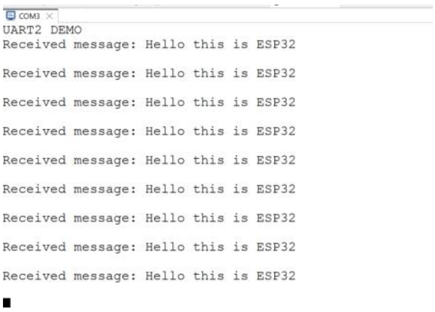
vi. From the ports dropdown, select the port connected to your ESP32 (here it is COM8):


testuart | Arduino IDE 2.3.4
File
Edit
Sketch
Tools
Help
ESP32S2 Dev Module
testuart.ino
ESP32S2 Dev Module
COM8
Unknown
COM3
Select other board and port...
to run onc
Serial1.begin(9600, SERIAL_8N1, NEW_RX_

vii. Navigate to the directory where you unzipped the testuart Arduino sketch and load it. Upload it to the ESP32-S2.

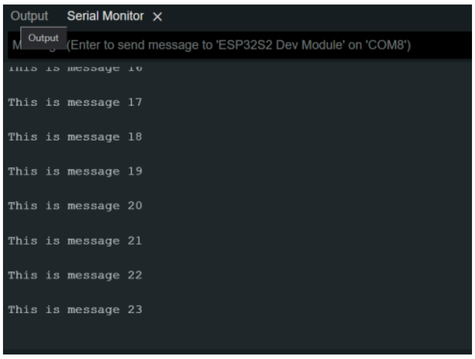
viii. You should see “Hello world!” appear on the UART console in MCUXpresso, and “This is message XX” on the Arduino IDE’s Serial Monitor, where XX is some integer. This shows that the ESP32 and MCXC444 boards are communicating over UART.

On the MCXC444 side:



```
UART2 DEMO
Received message: Hello this is ESP32

Received message: Hello this is ESP32

Received message: Hello this is ESP32

Received message: Hello this is ESP32

Received message: Hello this is ESP32

Received message: Hello this is ESP32

Received message: Hello this is ESP32

Received message: Hello this is ESP32

Received message: Hello this is ESP32
```

On the ESP32-S2 side:



```
This is message 16

This is message 17

This is message 18

This is message 19

This is message 20

This is message 21

This is message 22

This is message 23
```

5. Demo (3 MARKS)

   Demostrate the MCXC444 and ESP32 S2 communicating to your TA.

6. Connecting to the Internet

   As mentioned at the beginning, the ESP32 has WiFi that can connect to the Internet. This is beyond the scope of an RTOS course, but there is sample code provided in the Arduino IDE and on the Internet for this, and you are encouraged to look at that code and experiment.