Week 3 Lab

CC2511

This week, you will write your first microcontroller program. Your objective is to control some light-emitting diodes (LEDs) on a breadboard. You will also learn how to send and receive over a UART.

Your task

Build an application that allows LEDs to be controlled using the serial port input.

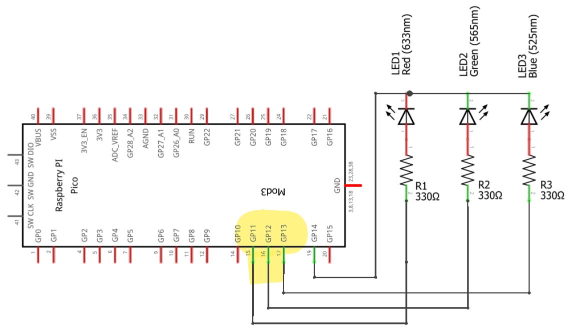
- When the "r" key is received, toggle the red LED.
- When the "g" key is received, toggle the green LED.
- When the "b" key is received, toggle the blue LED.

Whenever your program changes the state of a LED, write a line to the serial port output.

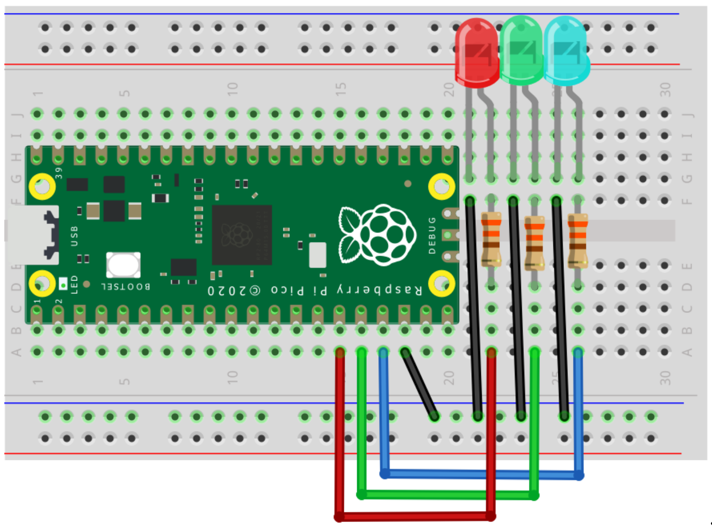
Build a breadboard

Using a breadboard, your Pico, three 330 Ohm resistors and three colored LEDs, create the circuit shown below:

A LED will only work if it is connected in the right direction - the D stands for Diode. Make sure that the long leg (the anode) is connected to the positive voltage, and the short leg (the cathode) is connected to ground.



GP11, 12 and 13 for R G and B respectively



Identify the pins that drive the LEDs

Referring to the schematic above, identify the microprocessor pins that drive the LEDs (in the form GPIOnn, where nn is the pin number, e.g. GPIO08, GPIO22 etc):

LED Colour	**Microcontroller pin**
Red	GPIO 11
Green	GPIO12
Blue	GPIO13

Identify the UART pins universal asynchronous receiver-transmitter

To understand the specification of the UART, inspect the **RP2040 Datasheet** under *Peripherals > UART (Section 4.2)*. Also refer to the GPIO muxing (multiplexing) table in Section 2.19.2, which we used in Lab 3 to specify the LED pins. Here we see that there are several different GPIO pins which can be used for each UART channel.

We are going to use UART0, so we can choose between GPIO 0/1, GPIO 12/13, GPIO 16/17 or GPIO 28/29. There will be two pins: RX and TX.

UART Function	**Microcontroller pin**
Receive (RX)	GPIO0 ?
Transit (TX)	GPIO1 ?

Writing the software

Now that you have identified which GPIO pins need to be set, you can write the software to achieve this.

Projects

A project for the CC2511 course uses a typical modern software project configuration: all the files are held in a single folder (or directory) and a well-defined set of subfolders.

The standard setup for building Raspberry Pi Pico 'C' language projects uses the **cmake** build system. A cmake project needs at least two files:

1. A 'C' source file, typically **main.c**.
2. A make file, which contains instructions as to how to build the project, named **CMakeLists.txt**. It should usually be in the same directory as **main.c**.

To create a better development experience, there may also be some configuration files.

Install or update the project generator

The generator consists of a python script and some template files, which simplify the process of creating a new development project for the Pico board.

Open a Developer Command Prompt for VS 2022.

If you have not yet installed the CC2511 project generator, type the following at the command line:

```
cd /d "%PICO_SDK_PATH%\.."
git clone https://github.com/JCU-CC2511-2021/ppgen
```

If you have already installed the generator, get the latest version as follows:

```
cd /d "%PICO_SDK_PATH%\..\ppgen"
git pull origin main
```

Create the project

To create the new project, open a Developer Command Prompt for VS 2022 and navigate to the folder containing your main git repository. Type the following at the command line:

```
python "%PICO_SDK_PATH%\..\ppgen\ppgen.py" Lab3
```

HINT—If you want to know more about what the project generator can do for you, add -h to the commad line.

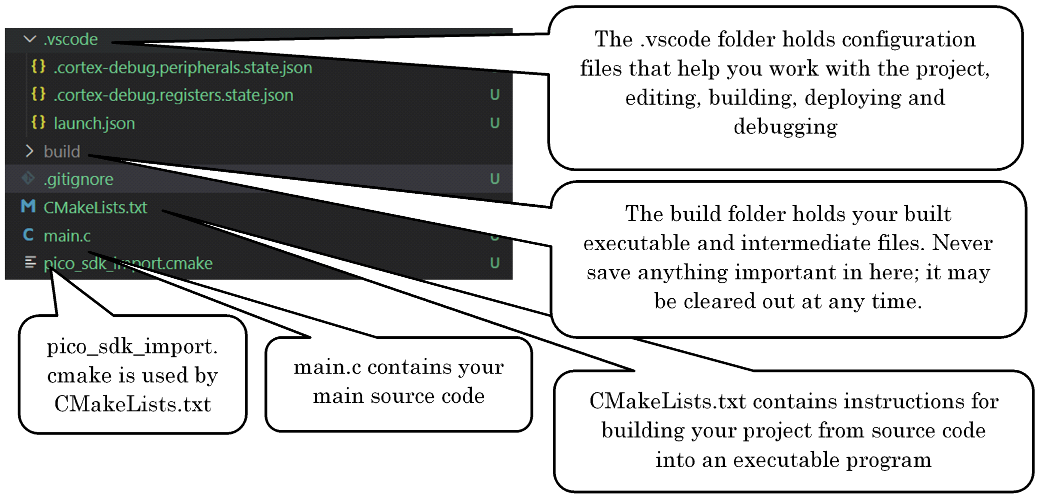
This will create a new folder, Lab3, and populate it with source files and project files.

Open the project

Now open the project, by typing the following at the same command line:

```
code Lab3
```

This instructs the VS Code editor to start, and to open the new folder.
If you inspect the files and folders produced by the project creation tool, you will see some or all of the following:


.vscode
.cortex-debug.peripherals.state.json
.cortex-debug.registers.state.json
launch.json
build
.gitignore
CMakeLists.txt
main.c
pico_sdk_import.cmake
U
The .vscode folder holds configuration files that help you work with the project, editing, building, deploying and debugging
The build folder holds your built executable and intermediate files. Never save anything important in here; it may be cleared out at any time.
pico_sdk_import.cmake is used by CMakeLists.txt
main.c contains your main source code
CMakeLists.txt contains instructions for building your project from source code into an executable program

Inspect the code

Open main.c, and you will find the barest of programs:

```
#include "pico/stdlib.h"
#include <stdbool.h>
int main(void) {
  // TODO - Initialise components and variables
  while (true) {
    // TODO - Repeated code here
  }
}
```

The Raspberry Pi Pico C/C++ SDK

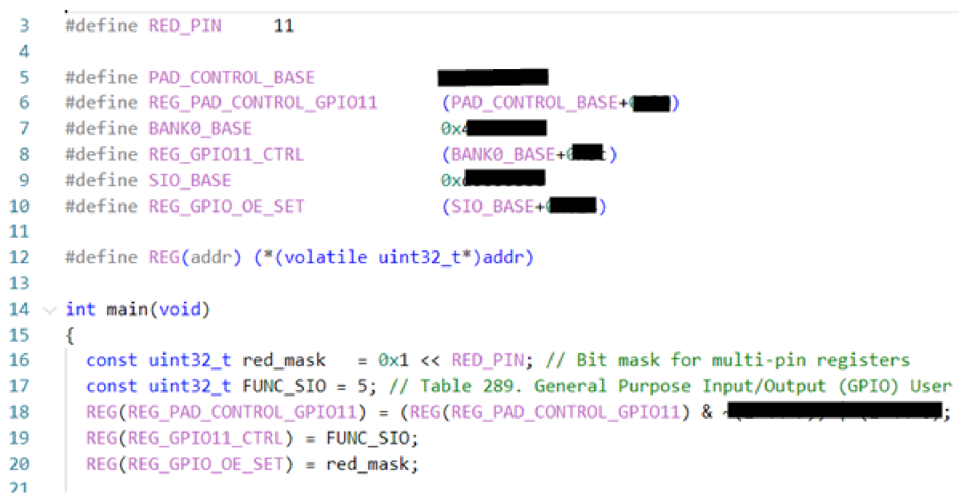
Note that the top line includes the stdlib.h file from the Raspberry Pi Pico SDK.

```
#include "pico/stdlib.h"
```

This will give you access to a multitude of helper functions and structures for many common operations.

Open the documentation for the Raspberry Pi Pico C/C++ SDK, which can be found at https://www.raspberrypi.com/documentation/pico-sdk.

Use the SDK functions to save time: they offer an abstraction layer that handles much of the hard work of programming with the Pi Pico. In this course we will be focusing mostly on the Hardware APIs (Application Programming Interfaces) described in the Pi Pico C SDK at https://www.raspberrypi.com/documentation/pico-sdk/hardware.html. For example, look at *Raspberry Pi Documentation* > *Pico C SDK* > *Hardware APIs* > *hardware_gpio* at https://www.raspberrypi.com/documentation/pico-sdk/hardware.html#hardware_gpio. This section contains functions that can handle all the work neededto set up the GPIO pin that controls a LED. Without the SDK, it looks something like this:



```
#define RED_PIN     11

#define PAD_CONTROL_BASE            [redacted]
#define REG_PAD_CONTROL_GPIO11      (PAD_CONTROL_BASE+[redacted])
#define BANK0_BASE                  0x4[redacted]
#define REG_GPIO11_CTRL             (BANK0_BASE+[redacted])
#define SIO_BASE                    0x[redacted]
#define REG_GPIO_OE_SET             (SIO_BASE+[redacted])

#define REG(addr) (*(volatile uint32_t*)addr)

int main(void)
{
  const uint32_t red_mask   = 0x1 << RED_PIN; // Bit mask for multi-pin registers
  const uint32_t FUNC_SIO = 5; // Table 289. General Purpose Input/Output (GPIO) User
  REG(REG_PAD_CONTROL_GPIO11) = (REG(REG_PAD_CONTROL_GPIO11) & [redacted];
  REG(REG_GPIO11_CTRL) = FUNC_SIO;
  REG(REG_GPIO_OE_SET) = red_mask;
```

With the SDK, you will be able to achieve all this in two lines of code.

Configure the USB serial

In a normal production environment our serial signals would enter and leave the microcontroller directly through a pair of GPIO pins, such as GPIO00/GPIO01. However, modern PCs do not conveniently support direct connections to their serial ports, so we will redirect the serial signals via the USB port.

We can achieve this by linking to a SDK library pico_stdio_usb. See Pico C SDK > Runtime Infrastructure > pico_stdio > pico_stdio_usb for a description of this library. To link our project to it, open the file CMakeLists.txt and locate the following line:

```
target_link_libraries(${projname} pico_stdlib)
```

Amend it to read as follows:

```
target_link_libraries(${projname} pico_stdlib pico_stdio_usb)
```

Look in Pico C SDK > Runtime Infrastructure > pico_stdio for an introduction to the high-level functions available. You will need to initialise the stdio ("Standard input/output") subsystem.

Write to the Serial

There are many ways to write to the serial output, with various different libraries available within the SDK that help you to write in as much or as little detail as you prefer. For this practical, you will find that **printf(...)** supplies all your needs. You will need to include the appropriate standard header file:

```
#include <stdio.h>
...
printf("Some text\r\n");
```

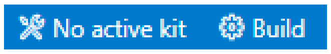
Add some very simple **repeated** code that will test whether your connection to the PC's serial port is working.

At the end of every line of text that you send to the serial, use "\r\n" to generate a Windows-style newline for display in your terminal emulator.

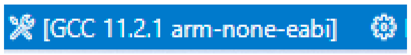
Building the project

Configure the cmake project

When you open your project in VS Code, you will be prompted to select a build kit. If you have not already selected one, then you will see "No active kit" on the status bar:



Click on the tool icon to select a kit, and from the dropdown list select "GCC [version] arm-none-eabi". The kit on the status bar will be updated, e.g.:



This may take some time; eventually you will something like the following in the VS Code Output pane:

```
[build] -- Configuring done
[build] -- Generating done
[build] -- Build files have been written to: E:/SomeFolder/Lab3/build
```

Build the project

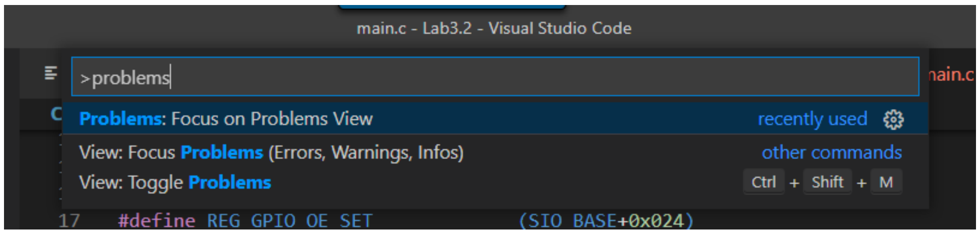
Each time you have made significant changes to your program and you are ready to test it, build it by pressing function key **F7**.

The first time you build your project can take quite a long time, because various SDK library files need to be compiled. After the first time the process is usually much quicker.

If there is an error in your code, you may see a compiler error message:

```
[build] NMAKE : fatal error U1077: ... return code '0x2' [build] Stop.
[build] Build finished with exit code 2
```

Open the *Problems* pane to see the error. If you can't see the *Problems* pane, then press function key **F1**, then type *Problems*, and select *Problems: Focus on Problems View*.


main.c - Lab3.2 - Visual Studio Code

>problems

Problems: Focus on Problems View — recently used

View: Focus Problems (Errors, Warnings, Infos) — other commands

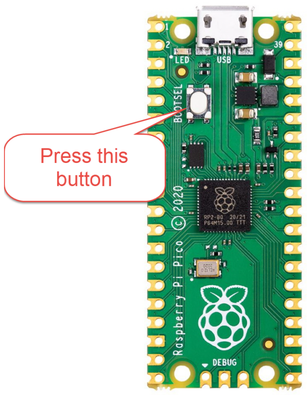
View: Toggle Problems — Ctrl + Shift + M

```
#define REG_GPIO_OE_SET                  (SIO_BASE+0x024)
```

If the build process completes successfully, you can open the **build** folder in Windows Explorer by right-clicking on it in the VS Code Folders window and selecting *Reveal in File Explorer.*

Once the folder is opened, locate the executable binary file, which will have the extension UF2, e.g. Lab3.uf2.

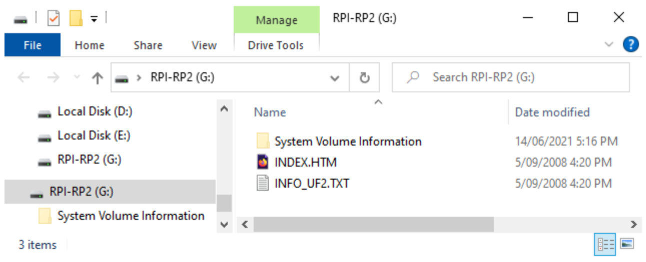
Delivering the code



To make the code run on the pico, you need to deliver it.

Connect the Raspberry Pi Pico to your Windows computer using a micro-USB cable, making sure that you hold down the BOOTSEL button to force it into USB Mass Storage Mode.

The board should automatically appear as an external drive.



Name	Date modified
System Volume Information	14/06/2021 5:16 PM
INDEX.HTM	5/09/2008 4:20 PM
INFO_UF2.TXT	5/09/2008 4:20 PM

You can now drag-and-drop the UF2 binary onto the external drive. The Raspberry Pi Pico will reboot, and unmount itself as an external drive, and start running the flashed code.

Testing the program

Deliver the program

Deliver your initial program to the Pico board the same way as in the last lab:

1. Remove the USB cable from the Pico board.
2. Holding the BOOTSEL button down, reconnect the USB cable.
3. Release the BOOTSEL button.
4. A new Windows Explorer window will open, displaying a virtual drive.
5. In your project's build subdirectory, locate the file Lab3.uf2 and copy it the virtual drive.

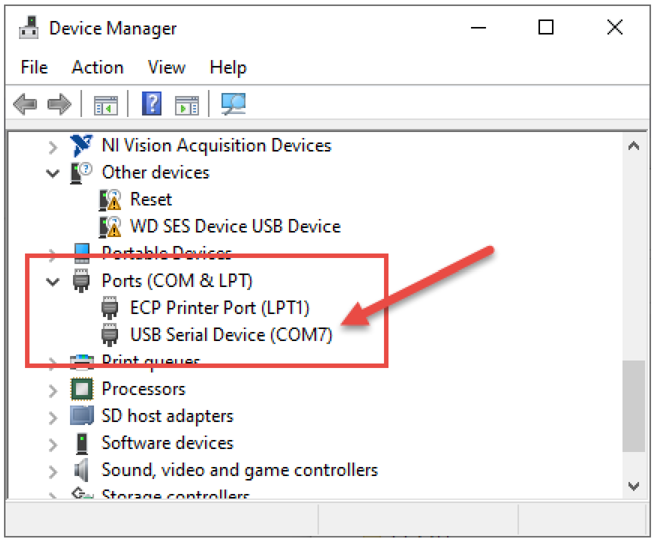
The program will be delivered to your Pico board, which will immediately run the program.

Set up the Terminal Emulator

Now determine which COM port the PC is receiving serial data through.

1. Return to your Developer Command Prompt for VS 2022 and enter the following:
   `devmgmt.msc`
2. In the Device Manager window, locate the Ports node and expand it if necessary.

Alternatively, you can locate the Device Manager in the Windows System menu by right-clicking the Windows menu.

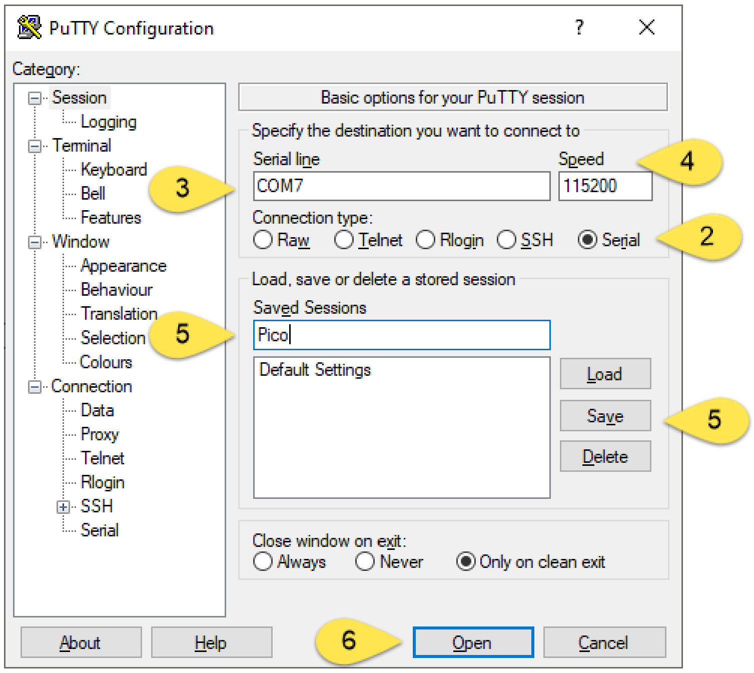


3. If the connection is successful, you wlll see a node named **USB Serial Device (COM?)**.

4. Note the number of the COM port, e.g. **COM7**.

Troubleshooting:- If you don't see the device in the Ports list, make sure that you have (i) linked to the `pico_stdio_usb` library and (ii) initialised the stdio sub-system.

Now start the Terminal Emulator:



Specify the destination you want to connect to

Serial line: COM7

Speed: 115200

Connection type: Raw Telnet Rlogin SSH [✓ Serial]

Load, save or delete a stored session

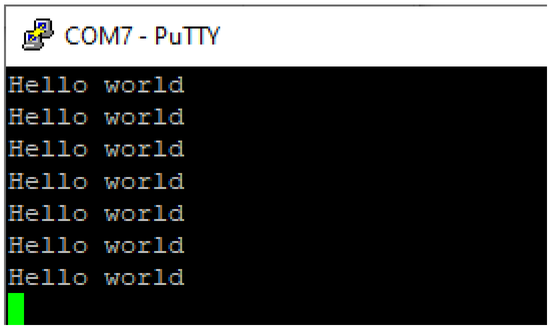
Saved Sessions: Pico

Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

1. Start the Putty application.
2. Set *Connection type* to **Serial**.
3. In *Serial line* enter the COM port name.
4. Set *Speed* to **115200**.
5. Optionally, set the Saved Sessions name to Pico and Save the Session. This will save you time next time you run Putty: you'll be able to load these settings.
6. Click **Open** to start the session.

You should now see your message repeating in the terminal window:



```
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
```

Work with the LED pins

Write code to initialise the three GPIO pins. Ensure that they are set to output. You may find it useful to define the three LED pins using macros, e.g.:

```
#define RED_LED   28 // these are NOT the correct numbers!
#define GREEN_LED 29
#define BLUE_LED  30
```

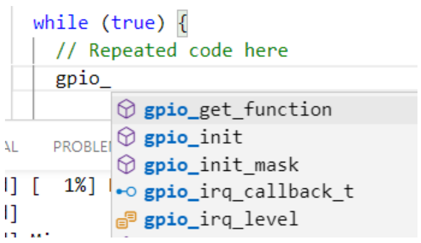
Using macros for this pupose has two major advantages:

1. Your code is much more readable - RED_LED is far more meaningful than 28.
2. If you change the wiring of your circuit, e.g. by moving the red LED to a different GPIO pin, your code only needs to be changed **in one place**. This corresponds to the software DRY principle.

For initialisation, you may find the following functions useful: `gpio_init()` and `gpio_set_dir()`.

In your repeated code, you will want to respond to user input by setting the GPIO pin high or low. Find the appropriate `gpio_` function for this by looking at Pico C SDK > Runtime Infrastructure > pico_stdio > Functions.

Hint: Try typing **gpio_** and see what suggestions the editor makes for you. This *IntelliSense* feature is very useful for looking up functions and for avoiding typos.

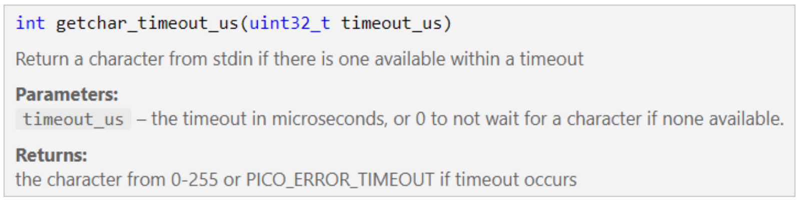


```
while (true) {
  // Repeated code here
  gpio_
```

Monitoring input

There are several different library functions which can be used to monitor input on the serial port, offering different levels of control and complexity. On an embedded system it is usually important that the system does not wait for input: if there is no input available, then the program should immediately carry on doing some other task. If a character is available, it should be processed immediately. Often that simply means adding it to a buffer, unless it is a character that signals the completion of a message or a command.

For this lab, we can use the function `getchar_timeout_us()`.



```
int getchar_timeout_us(uint32_t timeout_us)
```

Return a character from stdin if there is one available within a timeout

Parameters:
`timeout_us` – the timeout in microseconds, or 0 to not wait for a character if none available.

Returns:
the character from 0-255 or PICO_ERROR_TIMEOUT if timeout occurs

There are two interesting things about this function. First, even though its task is to get a character - an 8-bit integer - it returns a full integer. This allows it to use an out-of-range value to signal that no input is available. So it returns -1, which is **not** a legal character code, to indicate that no input was returned.

The second interesting thing is the `timeout_us` parameter. As the help says: use 0 to avoid waiting.

Notice that us stands for microsecond - you'll see this a lot, because μ - the greek character mu - is not legal in 'C'.

So you can inspect input with code like this:

```
while (true) {
  int ch = getchar_timeout_us(0);
  if (ch != PICO_ERROR_TIMEOUT) {
    // Handle the various interesting values of ch here...
  }
  // Wait for a while, or run another task
  // e.g. sleep_ms(20);
}
```

Solve simpler problems first

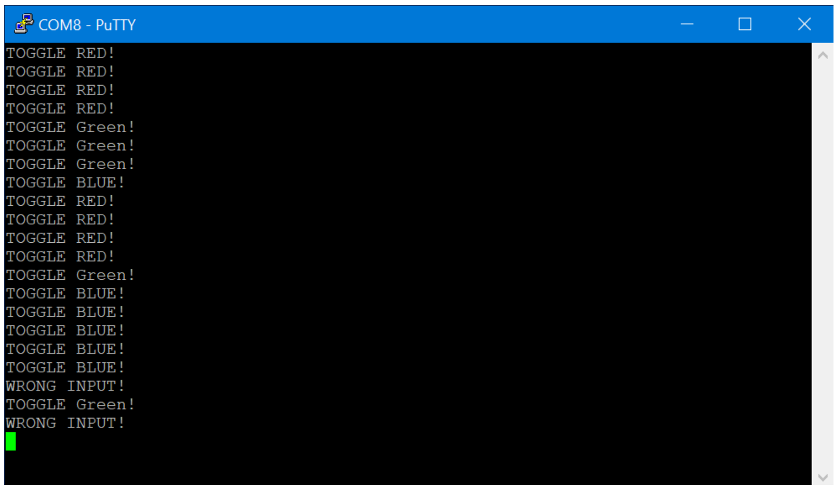
Some suggestions to help you build up to the task requirements:

- Write a program that transmits a string over the serial port repeatedly. This will confirm whether your code is connecting to the port and whether the terminal emulator is set up correctly.
- Write a program that receives a character and then echoes it back to the computer. This will enable you to test that you receive characters correctly.

Implement the task

You are now ready to implement the task requirements. Some tips:

- Use "`\r\n`" at the end of your messages to generate a Windows-style newline for display in your terminal emulator.
- The image below demonstrates how your user interface could operate. You do not need to replicate this exactly!



```
TOGGLE RED!
TOGGLE RED!
TOGGLE RED!
TOGGLE RED!
TOGGLE Green!
TOGGLE Green!
TOGGLE Green!
TOGGLE BLUE!
TOGGLE RED!
TOGGLE RED!
TOGGLE RED!
TOGGLE RED!
TOGGLE Green!
TOGGLE BLUE!
TOGGLE BLUE!
TOGGLE BLUE!
TOGGLE BLUE!
TOGGLE BLUE!
WRONG INPUT!
TOGGLE Green!
WRONG INPUT!
```

Committing your code to GitHub

You must upload your lab work to GitHub. Open up a **Git Bash** prompt, and use:

1. **git status** to see the modified files.
2. **git add <files>** to stage the modified files.
3. **git commit** to commit the previously staged files.
4. **git push origin master** to upload your work to GitHub.

HINT—Use of version control software such as Git is an essential part of professional software development. It is a mandatory requirement of this subject. If you are unsure of how to use Git, please ask your prac tutor or lecturer.

On a personal laptop, you might prefer to use a graphical user interface to commit your code to GitHub instead of the bash prompt, or alternatively you may prefer to use the excellent GitHub extension in VS Code. This is fine; it doesn't matter which tools you use.

Assessment

To finish this lab, you must:

- Demonstrate to your prac tutor a working program that can use the serial communication to individually control each of the three LED colours.
- Show your prac tutor your GitHub webpage with your Lab 3 code visible to demonstrate that you have uploaded it.