Recap from Networking

- In the Lecture we rushed through the most important computer networking concepts:
  - Layered networking stack with different protocol layers:
    - Physical layer
    - Data Link Layer (layer 2) (e.g. Ethernet, or Wifi)
    - Network Layer (layer 3) (e.g. IP)
    - Transport Layer (layer 4) (e.g. TCP or UDP)
    - Session, Presentation and Application Layer (e.g. HTTP)
  - The following concepts were explained:
    - Router : working on Network Layer (layer 3)
    - Network Switch : working on Data Link Layer (layer 2)
  - We mentioned the hierarchical structure of the Internet divided into a tree of subnets.
  - We explained the use of private subnets reserved for home/office networks
  - We explained the usage of NAT (Network Address Translation)
  - We explained DNS and DHCP services

Recap from Networking

- We connected the ESP32 to a WIFI network
  - We could read the obtained IP address, the network mask, the gateway and the DNS from the ESP32.
  - All these IP addresses and the mask are obtained from the DHCP server.

Microcontrollers

Part V

Asyncio in Micropython

(Christoph.Schwick@cern.ch)

Preamble

A very sad story...

Python and modern computer hardware

- Modern Computing hardware
  - Many cores (~100 in new CPUs)
  - Progress: more cores at constant clock speed
  - Software: multiple threads executed concurrently

Compatible ?

- The Python programming language
  - No support for concurrent execution
  - The famous GIL (Global Interpreter Lock) makes sure that only one thread at a time is running
  - **??? Why on earth did and do the python developers go down this road ???**
    - Easier and more lightweight memory management
    - More efficient (=fast) single threaded programs
    - Easily interface to many C-libraries which itself are not thread-safe
      - (A function is thread-safe if it can be entered by a second thread of execution before a previous execution has left the function)

Python and concurrency

- What can we do about this?
  - You CAN program **multiple threads** in python, but they are only one thread at a time is executed. This can be useful when **a thread waits for IO** (e.g. Keyboard input, a new value from a sensor measurement, …)
  - You can use **multiprocessing**:
    - This means **multiple processes** (=programs) are launched on the computer
    - Different instances of python interpreters are created in this case → each one has its own GIL → they can run at the same time on different cores of the CPU
    - Drawbacks:
      - Spawning a process is more expensive (= takes longer) than spawning a new thread
      - More memory is needed
      - Exchanging information between two processes is clumsy/expensive:
        - No shared memory resources between different processes exists
        - You need to create files, Pipes, shared memory regions, sockets or other complex constructs to transfer information from one process to the other (better than nothing though...)

After this shock...

Asyncio

Asyncio: a convenience

- **ALWAYS REMEMBER**: Asyncio is not a magic solution to the problems discussed above

- What is Asyncio:
  - It is a convenient way to write a program when you have to do different things in your program but often you need to wait for Input from outside (or for being able to transmit your output)
    - Many of the programs with a GUI (Graphical User Interface) fit in this category:
      The user gives some input and then something is calculated or processed and finally the result is provided
      - A pocket calculator
      - A program to collect data from sensors and display then (may be after some calculations)
      - A WEB server
      - A GPS for a bike or a car
      - …
  - Asyncio can manage multiple activities or tasks and decides what to run next. It only executes one task at a time. But when the running task starts waiting for input or decides to pause, then Asyncio chooses another task to run during this pause (in a fair way so that every task gets the possibility to run): This technique is called "**Cooperative Scheduling**"

Elements of Asyncio programming (1)

- Task
  - A task in Asyncio is similar to a stand-alone program in some aspects
    - it has it's own stack and it's own local variables.
  - However, it is part of a main program (with other tasks).
    - It can share variables with other tasks of the same program. Hence data exchange between tasks is easy
  - An asyncio program contains multiple tasks
    - They implement the various activities which need to be done by the program

- Event Loop
  - The Event Loop is the "main program" of an asyncio program.
  - It is not programmed by the user but it is part of the Asyncio machinery
  - It is deciding which task is running next (it manages the tasks)
    - This activity is called "**scheduling**"

Elements of Asyncio (2)

- Coroutine
  - Similar to a function
  - The corouting implements the activities to be done in a task.
    - It is the python code running in the task.
    - A task can contain multiple co-routines (but at least one)
  - Uses the stack of the task it is running in (like a function)
  - **Asyncio feature:** A coroutine can decide to pause execution ("yield the executing control")
    - In this case the Event Loop gives the execution control to a different task and resumes execution in that task where ever it stopped the last time it ran.
    - A co-routine always resumes execution (if the associated task is selected by the EventLoop) at the statement following the pause.
  - A coroutine can exit and return results (like a "normal" function)
  - Technical:
    - A Coroutine in python is defined with the async keyword

```
async def mycoroutine( par1, par2):
```

Elements of Asyncio (2)

- Calling or executing a coroutine
  - There are 2 ways to execute a co-routine:
    - You execute it in the context of the current task
    - You create a new task which executes the coroutine

- Calling a coroutine within a tasks

```
await mycoroutine(par1, par2)
```

  - **Attention**: If you call a coroutine like a normal function “nothing happens”:

    mycoroutine(par1, par2)
    - This call returns an “instance-object” of the coroutine but nothing is executed
    - Await tells the python interpreter to run the co-routine and wait for it to yield execution

- Create a new task which executes the coroutine

```
asyncio.create_task( mycoroutine(par1,par2) )
```

Elements of asyncio (3)

- Awaitables
  - Awaitables are objects which you can wait for in an asyncio program. During the waiting time the EventLoop will make sure that another task can run.
  - To wait for a awaitable you use

    ```
    await awaitable
    ```

    - This statement always yields the execution control
  - Coroutines are awaitables (as we have seen above)
  - Asyncio provides other “awaitables” like Locks, semaphores, queues
  - Libraries provide coroutines you can wait for. An often used example:

    ```
    await asyncio.sleep(n)
    ```

    - This means: wait at least n seconds (other tasks will run during this time) until you give the execution control back to this task (and this coroutine)
    - The time cannot be precise since after n seconds the currently running task has to pause before the Event loop gets a chance to execute this task again.
  - Also the asyncio library provided in micropython provides many useful networking coroutines some of which we will use in the exercises.

WARNING

Asyncio programming is not easy

You need to get used to a new way of thinking… this takes some time...

(normally we are used to think in sequential program execution)

However:

asyncio programming is easier than programming in a multi threaded environment like usually used in FreeRTOS and similar real-time operating system

asyncio is still a single threaded environment (remember the GIL)

I decided to insert this chapter here since it shows you some basics of Real Time Operating systems like FreeRTOS (concept of tasks, synchronization and a scheduler with cooperative scheduling) but it still much simpler than programming a C-application in FreeRTOS.

Exercises

- We program a little web server using the tools provided by asyncio
  - Reference is the micropython asyncio documentation
    - Navigate the documentation to “Python standard libraries and micro-libraries” → asyncio
  - You find a subset of the standard asyncio features which are implemented in micropython at the beginning of the documentation.
  - In the section “TCP stream connections” you find the components/functions you will need to implement the web server.
    - “start_server”
    - “Stream”
    - “read” or better: “readline”, “write” and “drain”
    - “close” and “wait_close” (for Stream and for the Server)
  - For all routines **note carefully which are co-routines** and hence you need to invoke them with await (or you create a task with them using create_task)
  - The web server should display data from our sensor.
  - You can start from the code template (to make things simple and since we want to learn how to write a server, we just simulate the sensor data with random data, or use a fixed value. Afterwards you can add the real sensor reading code, if your want.)