prac_4_lphtum003_jhrmoh002

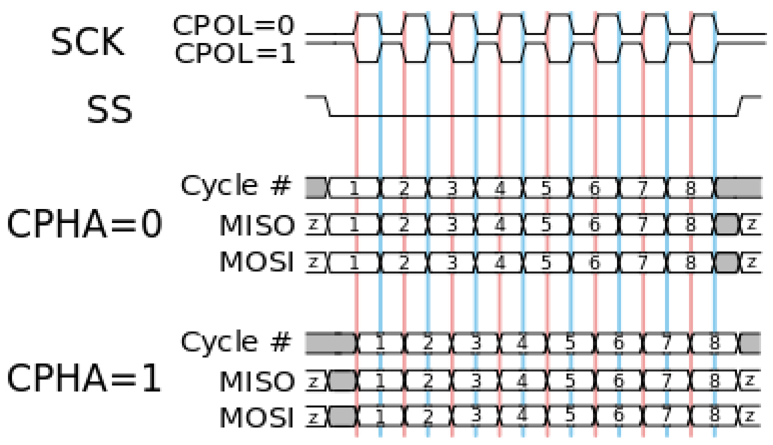
Explain the SPI communication protocol with a timing diagram.

In addition to setting the clock frequency, the master must also configure the clock polarity and phase with respect to the data. Motorola SPI Block Guide names these two options as CPOL and CPHA respectively, and most vendors have adopted that convention.

- CPOL determines the polarity of the clock. The polarities can be converted with a simple inverter.
    - CPOL=0 is a clock which idles at 0, and each cycle consists of a pulse of 1. That is, the leading edge is a rising edge, and the trailing edge is a falling edge.
    - CPOL=1 is a clock which idles at 1, and each cycle consists of a pulse of 0. That is, the leading edge is a falling edge, and the trailing edge is a rising edge.
- CPHA determines the timing of the data bits relative to the clock pulses. It is not trivial to convert between the two forms.
    - For CPHA=0, the "out" side changes the data on the trailing edge of the preceding clock cycle, while the "in" side captures the data on (or shortly after) the leading edge of the clock cycle. The outside holds the data valid until the trailing edge of the current clock cycle. For the first cycle, the first bit must be on the MOSI line before the leading clock edge.
    - An alternative way of considering it is to say that a CPHA=0 cycle consists of a half cycle with the clock idle, followed by a half cycle with the clock asserted.
    - For CPHA=1, the "out" side changes the data on the leading edge of the current clock cycle, while the "in" side captures the data on (or shortly after) the trailing edge of the clock cycle. The outside holds the data valid until the leading edge of the following clock cycle. For the last cycle, the slave holds the MISO line valid until slave select is de-asserted.
    - An alternative way of considering it is to say that a CPHA=1 cycle consists of a half cycle with the clock asserted, followed by a half cycle with the clock idle.

The MOSI and MISO signals are usually stable (at their reception points) for the half cycle until the next clock transition. SPI master and slave devices may well sample data at different points in that half cycle.

This adds more flexibility to the communication channel between the master and slave.



Define interrupt and threaded call-back in the context of an embedded system.

Interrupt controlled system

Some embedded systems are predominantly controlled by interrupts. This means that tasks performed by the system are triggered by different kinds of events; an interrupt could be generated, for example, by a timer in a predefined frequency, or by a serial port controller receiving a byte.

These kinds of systems are used if event handlers need low latency, and the event handlers are short and simple. Usually, these kinds of systems run a simple task in a main loop also, but this task is not very sensitive to unexpected delays.

Sometimes the interrupt handler will add longer tasks to a queue structure. Later, after the interrupt handler has finished, these tasks are executed by the main loop. This method brings the system close to a multitasking kernel with discrete processes.

Pre-emptive multitasking/multithreading

In this type of system, a low-level piece of code switches between tasks or threads based on a timer (connected to an interrupt). This is the level at which the system is generally considered to have an "operating system" kernel. Depending on how much functionality is required, it introduces more or less of the complexities of managing multiple tasks running conceptually in parallel.

As any code can potentially damage the data of another task (except in larger systems using an MMU) programs must be carefully designed and tested, and access to shared data must be controlled by some synchronization strategy, such as message queues, semaphores or a non-blocking synchronization scheme.

Because of these complexities, it is common for organizations to use a real-time operating system (RTOS), allowing the application programmers to concentrate on device functionality rather than operating system services, at least for large systems; smaller systems often cannot afford the overhead associated with a generic real-time system, due to limitations regarding memory size, performance, or battery life. The choice that an RTOS is required brings in its own issues, however, as the selection must be done prior to starting to the application development process. This timing forces developers to choose the embedded operating system for their device based upon current requirements and so restricts future options to a large extent.[16] The restriction of future options becomes more of an issue as product life decreases. Additionally the level of complexity is continuously growing as devices are required to manage variables such as serial, USB, TCP/IP, Bluetooth, Wireless LAN, trunk radio, multiple channels, data and voice, enhanced graphics, multiple states, multiple threads, numerous wait states and so on. These trends are leading to the uptake of embedded middleware in addition to a real-time operating system.

Write a function that converts a 10-bit ADC reading from the potentiometer to a 3V3 limited voltage output.

```
from gpiozero import MCP3008

pot = MCP3008(channel=0)

while True:
    print(pot.voltage)
```

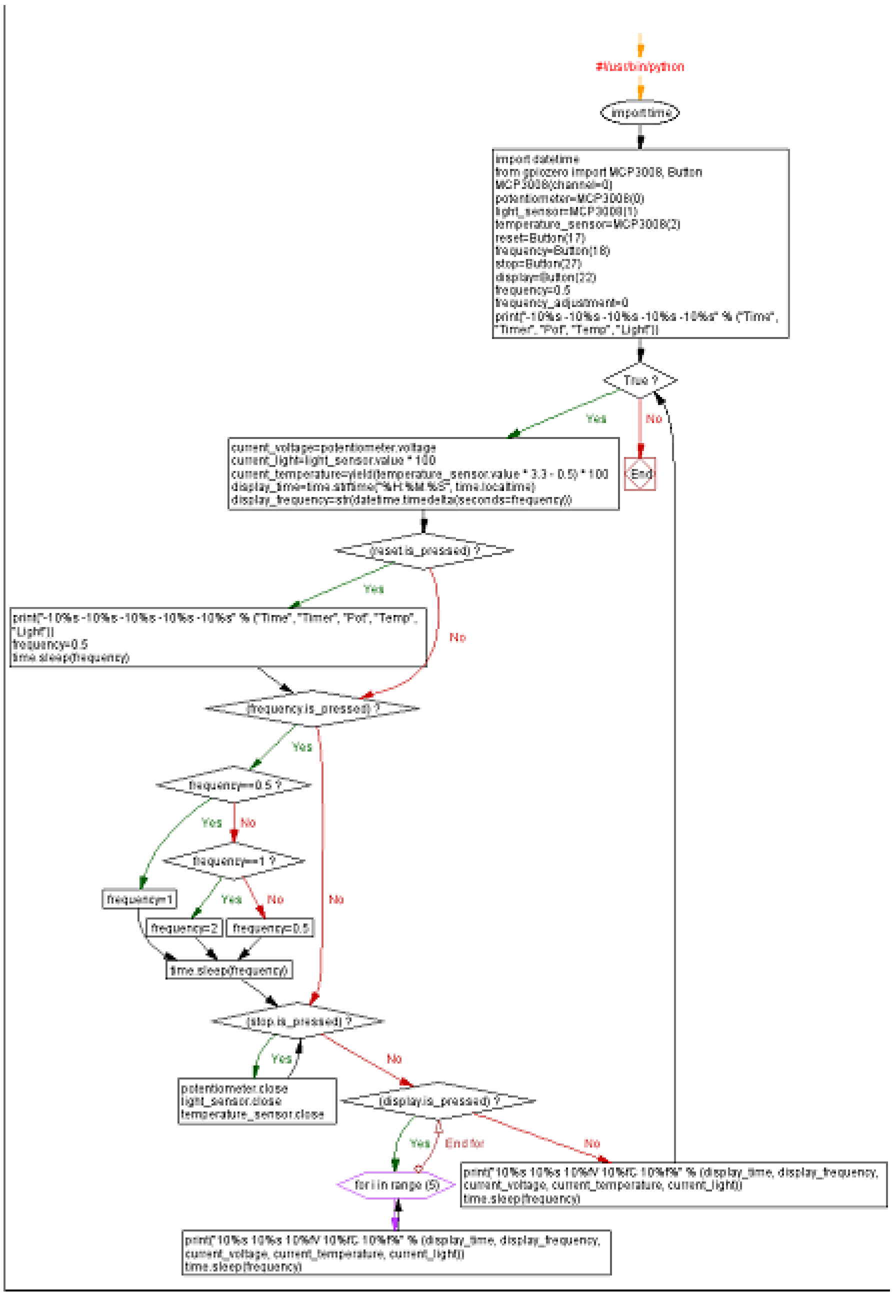
Write a function that converts a 10-bit ADC reading from the temperature sensor to a reading in degree Celsius (Have a look at the datasheet).

```
from gpiozero import MCP3008
from time import sleep

def convert_temp(gen):
    for value in gen:
        yield (value * 3.3 - 0.5) * 100

adc = MCP3008(channel=0)

for temp in convert_temp(adc.values):
    print('The temperature is', temp, 'C')
    sleep(1)
```

Write a function that converts a 10-bit ADC reading from the LDR to a percentage representing the amount of light received by the LDR.

```
from gpiozero import MCP3008

ldr = MCP3008(0)
ldr_percentage=ldr.value*100
```

Draw a flowchart of the system.


```
#!/usr/bin/python
```

```
import time
```

```
import datetime
from gpiozero import MCP3008, Button
MCP3008(channel=0)
potentiometer=MCP3008(0)
light_sensor=MCP3008(1)
temperature_sensor=MCP3008(2)
reset=Button(17)
frequency=Button(18)
stop=Button(27)
display=Button(22)
frequency=0.5
frequency_adjustment=0
print("-10%s -10%s -10%s -10%s -10%s" % ("Time",
"Timer", "Pot", "Temp", "Light"))
```

```
current_voltage=potentiometer.voltage
current_light=light_sensor.value * 100
current_temperature=yield(temperature_sensor.value * 3.3 - 0.5) * 100
display_time=time.strftime("%H:%M:%S", time.localtime)
display_frequency=str(datetime.timedelta(seconds=frequency))
```

```
(reset.is_pressed) ?
```

```
print("-10%s -10%s -10%s -10%s -10%s" % ("Time", "Timer", "Pot", "Temp",
"Light"))
frequency=0.5
time.sleep(frequency)
```

```
(frequency.is_pressed) ?
```

```
frequency==0.5 ?
```

```
frequency==1 ?
```

```
frequency=1
```

```
frequency=2
```

```
frequency=0.5
```

```
time.sleep(frequency)
```

```
(stop.is_pressed) ?
```

```
potentiometer.close
light_sensor.close
temperature_sensor.close
```

```
(display.is_pressed) ?
```

```
for i in range (5)
```

```
print("10%s 10%s 10%fV 10%fC 10%f%" % (display_time, display_frequency,
current_voltage, current_temperature, current_light))
time.sleep(frequency)
```

```
print("10%s 10%s 10%fV 10%fC 10%f%" % (display_time, display_frequency,
current_voltage, current_temperature, current_light))
time.sleep(frequency)
```

References

https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

https://en.wikipedia.org/wiki/Embedded_system#Interrupt-controlled_system

https://en.wikipedia.org/wiki/Embedded_system#Cooperative_multitasking

https://gpiozero.readthedocs.io/en/stable/api_spi.html#analog-to-digital-converters-adc

https://gpiozero.readthedocs.io/en/stable/recipes.html#measure-temperature-with-an-adc