Final Report

INFO2055 - Embedded Systems Project

Jérôme BAYAUX - s161630
Antoine MALHERBE - s164138
Tom PIRON - s161415
Chloé PREUD'HOMME - s154552

Academic Year 2020-2021

1 Project goal

The objective of this project consists in measuring and collecting data about the environment in a mushroom box. This mushroom box is an idea of the Pot'Ingé, a group of students and PhD students building a vegetable garden at the University of Liège willing to harvest mushrooms in a mushroom box. To help them monitor their mushroom culture, several environment variables such as temperature, air humidity, light level and CO_2 concentration should be measured with sensors and should be easily accessible for the user. The idea was to store the measurements in an external flash memory module until a user requests them. When this happens, the data is sent through Bluetooth towards a mobile phone in order to display the measured variables in a fashionable way.

During our development, we focused ourselves on making sure that the temperature, humidity and light sensor function properly. We planned to add the gas measurements only if there was some time left once all the other sensors as well as the Bluetooth module operate correctly. Indeed, it is more interesting to have a complete working cycle (i.e data acquisition, storage and sending) than to have a gas sensor that was considered as an added value to our project since it is more difficult to handle and quite expensive.

2 Capabilities of the final version

The final version of the project fulfills the objectives expected by the Pot'ingé except for the measurement of the CO_2 concentration. The implemented circuit is capable of measuring the data about the temperature, the air humidity and the light level in the mushroom box and to store these data in the internal memory storage of the PIC micro-controller. In fact, we encountered several difficulties with the flash memory modules so we needed to find an alternative in order to still have a fully functional mushroom monitoring station (see section 6 for more details). Those data are transmitted from the PIC micro-controller to the user's smartphone by Bluetooth thanks to the presence of an external module. The collected data are given in hexadecimal but conversion formula and tables are furnished to the user for each type of data. The figure 1 shows the final circuitry we deliver to our client.

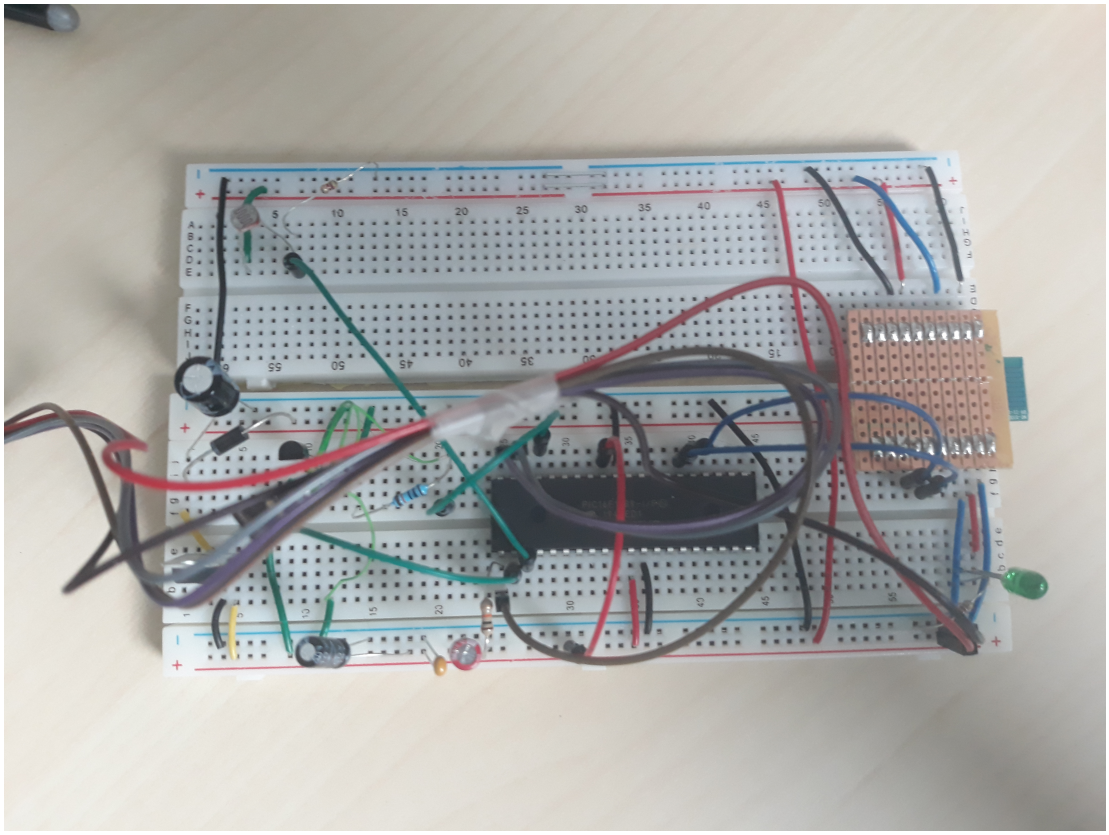


Figure 1: Photo of the delivered system

3 Schematics

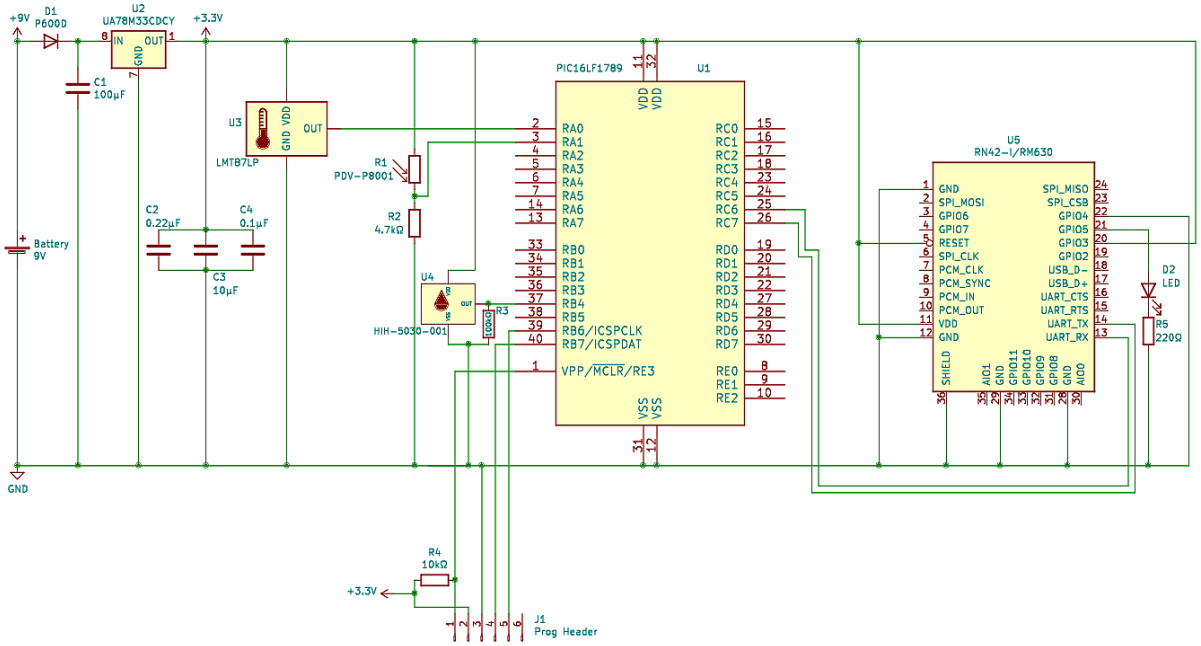


Figure 2: Schematics of the final version.

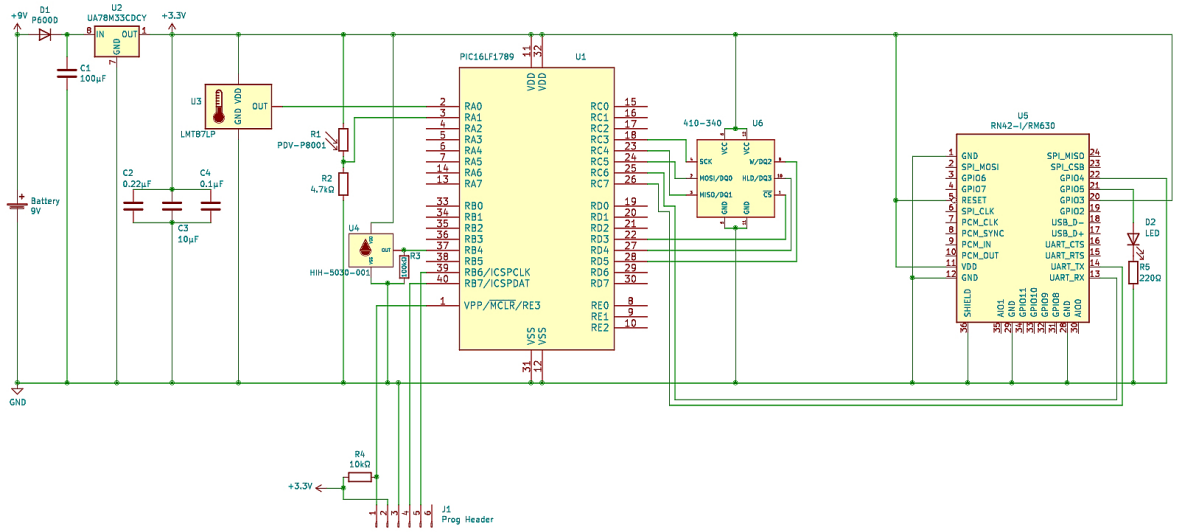


Figure 3: Schematics of the project with the flash memory.

3.1 Final version

The schematic of the hardware in Figure 2 is described from left to right.

- First of all, we have a power supply. The energy source is a 9V battery.
  Then a diode `D1 P600D` is used to protect the circuit in case the polarity of the battery is reversed.
  Finally we use a linear voltage regulator `UA78M33CDCY` to bring to voltage to $3.3V$ which is the commonly accepted voltage by all the components. The choice of this particular component was driven by the fact that it has 3 pins (`IN`, `OUT`, `GND`) which make it easy to connect in the circuit. Two capacitors (`C1=100`μ`F` and `C3=10`μ`F`) are needed to use the component in the circuit. The first one is to stabilise the output of the battery and the second to absorb spikes in the power consumption primarily generated by the PIC and the Bluetooth module. This circuit can deliver an output current up to $500mA$, which is way above the expected consumption of the full system.

- To collect the temperature, a temperature sensor `LMT87LP` is used. Its output pin is connected to a general I/O pin, `RA0`, of the PIC16LF1789 microcontroller. Further details of this sensor are given in section 5.1.

- A photo-resistor is used as a light sensor. The resistance of this component vary with the luminosity, however this is not linear. This make it very difficult to have accurate measurements, but can be used for coarse measurements. As our application doesn't require a great precision this solution is acceptable. We use a voltage divisor with the photo-resistance and a $4.7k\Omega$ resistance to retrieve the photo-resistance value. The output is then connected to a general I/O pin, `RA1`, of the PIC.

- To measure the level of humidity with an increased accuracy in the environment of the mushrooms, the humidity sensor `HIH-5030-001` depicted by a droplet on the schematic, was chosen as explained in section 5.1. In our circuit, its output is simply connected to a general I/O pin, `RB5`, of the PIC16LF1789 microcontroller.

- The PIC16LF1789 microcontroller is the central point of our circuit and will perform all software operation like reading values on sensors, storing them on an external memory module and eventually sending it via Bluetooth. In addition, we have a decoupling capacity C2 of $0.22\mu F$ to stabilize it's voltage input.

- The bluetooth module `RN42/RN42N` is used to send data outside of our system. In particular, the idea is to send data on a smartphone that will be able to send it on an external server. It has a considerable series of pins that are connected appropriately to the PIC16LF1789 microcontroller. In order to communicate with the microcontroller, the `UART RX` and `UART TX` should imperatively be connected to the `RC6` and `RC7` respectively of the microcontroller to receive and to transmit data. One other pin is used to set the bluetooth operation mode to slave mode, which mean that the module wait for a master device to connect to it. Finally, for debugging purpose a led indicate the connection status, it can be removed in the final product to save energy.

On Figure 3, we can see the schematics with the flash module that we weren't able to use. It is however included to help future users implement the flash memory if it's needed. The flash memory `410-340` is used to store collected data, this allow to have a longer operating time. All of the communication pins are connected to the PIC via particular pins dedicated to connection with a SPI bus.

3.2 Changes

During the final steps of the implementation, we had to change a few part of our hardware.

Firstly, after a long time trying, we decided to abandon the use of the flash memory, as we were not able to use it correctly. This is annoying as it reduce the operating time to around 6 days, but is still correct for a prototype.

Secondly, we did not implemented the use of a gas sensor. As stated in our previous reports, this was the last sensor that we intended to add. Unfortunately, we did not had the time to implement it and therefore didn't bought it.

Furthermore, we weren't able to get the photo-diode circuit that we originally planed on using. We replaced it with a photo-resistor. This cause the quality of the measurement to degrade substantially, which mean that the mean use of the luminosity sensor should be to setup an alarm threshold. It is possible by tuning the value of the R2 resistor to have a pretty good measurement around a specific value.

Finally, the pins that are connected to the Bluetooth module have changed. We initially were connecting to much pins, so we just removed those connections.

4 Software architecture

4.1 RR with interrupts

In our application, time constraints are relatively easy to live with because periodic task consisting in collecting data from sensors is not really time-sensitive. Indeed, variables measured by those sensors will not vary quickly over time. For example, the temperature is likely to stay constant in an interval of several seconds or even several minutes. Therefore, measuring it a few seconds before or after will not affect the extracted value.

Even if our periodic task can be delayed for several seconds, there are still actions that need to be performed quickly. For example, we use a timer to know when to make new measurements. As soon as this timer occurs, we must restart another one for the next measurements. Another example is linked to the method used to extract the voltage output from a sensor. We use an Analog to Digital Converter and as soon as its conversion is finished, we must use the computed value.

Following those two observations about time constraint, we decided to use a Round Robin with interrupts architecture. The two next points will describe more precisely what are the tasks and the interrupts used of our implementation.

4.2 Tasks

- `get_temp`: If ADC is available, launch the conversion to extract measure from the temperature sensor.
- `get_humid`: If ADC is available, launch the conversion to extract measure from the humidity sensor.
- `get_luminosity`: If ADC is available, launch the conversion to extract measure from the luminosity sensor.
- `check_space`: Check if there is still space available for future measurements.
- `blue_send_data`: Launch a Bluetooth transaction to send data to an external device. Note that no measurements can be taken during this operation.

4.3 Interrupts

- `timer1_handler`: Triggered by the Timer1 of PIC every 0.524288 seconds. Measurements are not taken for each occurrence of this timer because we want a bigger delay. Therefore, when this interrupt is triggered, we simply increment a counter that allows us to know when to trigger a new set of measurements.
- `adc_completion`: Triggered by the Analog-to-Digital Converter of the PIC when its conversion is over. It is at that moment that we have to extract the converted data and store it into memory.
- `blue_receive_handler`: Triggered by EUSART module when it receives a command from the Bluetooth external module.

- `blue_send_handler`: Triggered when we want to send data using the external Bluetooth module.

4.4 Pseudo-code

The complete pseudo-code can be found in Appendix A.

5 Measurements

5.1 Choice of sensors

The temperature is measured with the temperature sensor `LMT87LP` which is an integrated circuit with an analog output voltage linearly and inversely proportional to the temperature. The choice was turned toward this sensor as it presents interesting advantageous properties for this particular application. Indeed, it can operate up to a 5.5 V power supply with a low power consumption of 5.4 μA and its output is protected against short circuit. Moreover, it has a gain of $-13.6\text{mV}/^{\circ}\text{C}$.

The humidity sensor `HIH-5030-001` was chosen for its high accuracy and also for its resistance to hazards that can be encountered in the environment of the mushrooms such as condensation, dust, dirt, oils and common environmental chemicals. Furthermore, it can be easily connected to the micro-controller thanks to its near linear voltage output and it has a current draw of 200μA which is convenient for battery operated systems.

Instead of photodiode `OPT101P` initially chosen to measure temperature, a photoresistor was used because the photodiode was not received. The photoresistor does not enable to measure accurately exact light levels in Lux but enables instead to notice basic light variations which is suitable for the purpose of the project. According to the enlightment in the box, the resistance of the photoresistor changes from 200 $k\Omega$ in dark rooms to 300 Ω corresponding to 1000 Lux during overcast day. It is sensitive to wavelengths between 400 nm and 600 nm.

5.2 Circuitry

The circuit of the temperature sensor is based on the following schematic of the figure 4. The capacitor is a decoupling capacitor between V_{DD} and the ground.

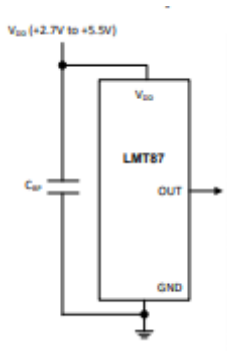


Figure 4: Circuit schematic of the temperature sensor

The circuit of the humidity sensor is based on the schematic provided in the datasheet of the component represented in FIGURE 5 except that 100 kΩ resistor was used instead of the 65 kΩ resistor which we did not have in our possession. This is not an issue as 65 kΩ is the load minimal value.

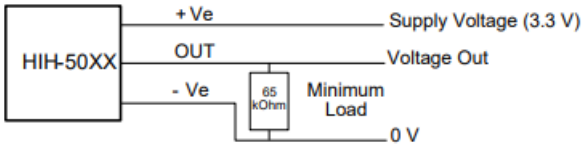


Figure 5: Circuit schematic of the humidity sensor

A resistance of 4.7 $k\Omega$ was placed in series of the photoresistor to obtain coherent values according to the enlightment of a mushroom box. A schematics of the circuitry can be seen later on figure 6.

5.3 ADC

In our project, the ADC module is used to gather the analog voltages outputted by the various sensors (temperature, humidity and luminosity). For this reason, it is necessary to change the channel selected as input for the ADC. Thus, to make a measurement, the first thing to do is to change ADC's input by selecting the channel that corresponds to the right sensor. Finally, after waiting for the acquisition time (i.e the time required by the ADC module to fully charge its charge holding capacitor to the input channel voltage level), the conversion can be started and the value of the analog input voltage can be obtained.

According to the PIC16's datasheet, in the worst scenario (i.e when the impedance of the analog source is equal to $10k\Omega$), the required acquisition time is $4.87\mu s$. In this project, the acquisition time was fixed to approximately $6\mu s$ and should then be sufficient to avoid all issues.

Here are some remarks and explanations about the tasks that use the ADC module to collect data from the sensors :

- When Timer1 overflows and triggers an interrupt it will trigger the first measurement task. Afterwards, tasks are triggered one at a time by the previous one. This mechanism is used so that each measurement data is collected one at a time.
- The job of each task is first to select the correct channel as input of the ADC. Then, it needs to wait for the *acquisition time*. Finally, it will set the GO bit of ADCON0 register to make the ADC start the conversion.
- As it is explained in section 6.2, we decided to store the measurements in the linear data memory of the PIC16. To do so, we used the File Select Registers (FSR). All the details about this part of the implementation are also given in section 6.2. Obviously, we store directly the value that the ADC outputs without translating it into a voltage because we would lose too much accuracy. Indeed, the number got from the ADC needs to be divided by 4096 and multiplied by 3.3, leading to a

number between 2 and 3 so the decimals are very important. The idea is thus to leave this conversion as a job for the entity that will retrieve the measurements in order to use them for interpretation.

5.4 Conversions

The sensors measure environmental variables in terms of voltage values that are collected by the ADC. n is the value between 0 and 4096 that is sent by the ADC, that is stored and sent via Bluetooth to the user's phone. The following formula gives the correspondence between the measured voltage and the value n:

$$V_m = \frac{n}{2^{12}} \times V_{CC}$$

The temperature of the mushroom box can be retrieved with the following formula given in the datasheet:

$$V_{TEMP}(mV) = 2230.8 - [13.582\,(T - 30)] - \left[0.00433\,(T - 30)^2\right]$$

The last term of this formula can be neglected as it is very small and is included in the error margin. The final formula that can be used by the user to transform the n value of the ADC into a temperature in $°C$ is the following:

$$T = \left(\frac{2230.8 - \frac{n}{4096} \times 3300}{13.582}\right) + 30$$

The value of the relative humidity (RH) is given by the following formula typical at $25°C$ provided in the datasheet.

$$V_m = V_{CC} \times (0.00636(\mathrm{RH}) + 0.1515)$$

Possessing the value sent by the the ADC on his smartphone, the user can find the relative humidity (in %) measured by the sensor using this formula :

$$\mathrm{RH} = \frac{1}{0.00636} \times \left(\frac{n}{4096} - 0.1515\right)$$

The photoresistor can be seen as a resistor R_1 which can be computed thanks to the formula of a voltage divider and by knowing that $R_2 = 4.7k\Omega$. The voltage divider is the following:

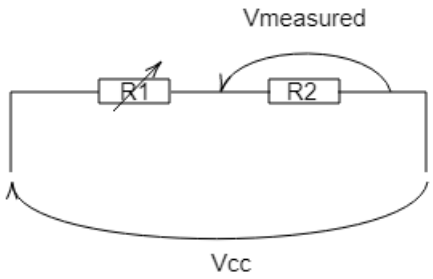


Figure 6: Photoresistor voltage divider

$$V_m = \frac{R_2}{R_1 + R_2} \times V_{CC} \Rightarrow R_1 = R_2 \times \left(\frac{V_{CC}}{V_m} - 1\right)$$

The value n of the ADC being given, the resistance R_1 of the photoresistor can be computed with the following formula:

$$V_m = \frac{n}{2^{12}} \times V_{CC} \Rightarrow R_1 = R_2 \times \left(\frac{2^{12}}{n} - 1\right) \Rightarrow R_1 = R_2 \left(\frac{4096}{n} - 1\right)$$

Once the value of R_1 is computed, the user can determine approximately the enlightment of the mushroom box using the following table:

Ambient light like...	Ambient light (Lux)	Photocell resistance (Ω)
Dim hallway	0.1	600k
Moonlight night	1	70k
Dark room	10	10k
Dark overcast day / Bright room	100	1.5k
Overcast day	1000	300

Table 1: Conversion table of the photoresistor

6 Memory management

One question quickly comes up in our minds. Once our measure are made, what do we do with it ? The most basic response to this question is to store them but where ? Our idea mainly consisted in saying that we will store them temporarily in our system and eventually send them to a remote server where data can be treated and displayed in a fashionable way through a web application. In this section, we will discuss the two solutions we tried to store data temporarily in our system.

6.1 External flash memory

The first solution we thought about is to use an external component and in particular, a flash memory module. The component we choose to do so was the NOR flash Pmod SF3. The actual component we use is a flash and a basic circuitry around allowing us to connect it easily to our system. This element allows us to store up to 32MBytes (or 128 Mbits) of data.

6.1.1 SPI

The interface with the micro-controller is performed through a SPI bus where SPI is a protocol similar to I2C. Like in I2C, we have a master and a slave where those notions are not related to sender and receiver. The master has the responsibility to generate a clock signal SCLK that defines the transmission rate and the periods where data can be sampled and/or modified by the two components. A big difference with I2C is that a transaction is always full-duplex meaning that both components are at the same time senders and receivers. Therefore, there are two lines called MOSI (Master-Out-Slave-In) and MISO (Master-In-Slave-Out) on which data is exchanged. Even if only one of them wants to send data, the other component will have to send some bullshit data on the other line to

be able to receive the message. The other big difference with I2C is that addressing is made by activating the correct slave through a CS (Chip Select) input pin. In general, this means that there is one addressing line per slave making the system less scalable. In our case, we have only one slave, so this is not a problem at all. An example of connections is shown on figure 7

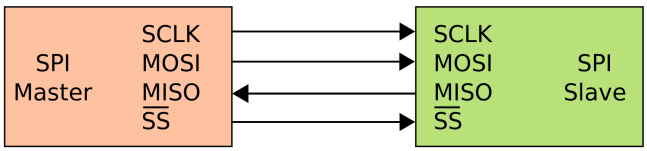


Figure 7: Basic SPI bus example

In our system, the micro-controller is the master and the flash operates in slave mode and we decided never to have any full-duplex transmission. This simplified our system as we could say that the PIC was either receiving or sending data.

6.1.2 Commands

The flash module operates thanks to commands. More specifically, in order to perform an operation (read, write, ...), we first need to send a command identified by a 8-bit number. For example, the READ command corresponds to 0x03. A full list of commands can be found on the component datasheet. In our application, only three of them are used: READ, WRITE ENABLE and PAGE PROGRAM.

The READ command allows, as its name says, to read data stored in memory. To do so, the command should be followed by a 3-byte address (most significant byte first) that describes where to read. Once the flash has received this address, it starts sending what is stored there. The big strength of this command is that it is a continuous read meaning that it reads everything that follows the requested address. For instance, if it is asked to read at address 0x000042, the flash will first responds with value at 0x000042 but will continue to send data after according to what it is stored at the next addresses. The only way to stop this is to stop the entire transaction. Actually, this is quite easy to do. Remember that SPI is full-duplex and needs both actors to communicate. To stop a transaction and to stop reading, the micro-controller simply can stop sending bullshit data and stop generating the clock signal.

The WRITE ENABLE command is there to put the flash in a state where it accepts to modify its internal content. In other words, it enables the next WRITE operation. This command is a 1-byte command to which the flash will not respond (will respond bullshit data or 0x00).

The PAGE PROGRAM command triggers a write operation at a particular address. Like for READ, the command must be followed by a 3-byte address that describes where to write data in memory. After that, the micro-controller can send several bytes that will be stored contiguously at the given address. The notion of page corresponds to the way memory is organised inside the flash memory. Indeed, similarly to several computer

operating systems, the memory is divided into pages of 256 bytes. In our case, this implies that we cannot write into several pages with the same command. Therefore, we need to be careful in our contiguous write operation because if we reach the end of the page, the flash will overwrite the beginning of the page. In our system, this problem is solved by saying that we always store 8 bytes in one operation so that the address is always a multiple of 8. Therefore, when we reach the end of the page, we also reach the end of our write operation.

During our experiments, we capture several transactions to give example of what is exchanged. Those are shown on figure 8. In all of these figures, the yellow signal is always the clock signal generated by the micro-controller. Also note that the oscilloscope was in AC mode and was therefore removing the DC component. This explains why we often obtain negative voltage. If so, it means that by default the signal is high.

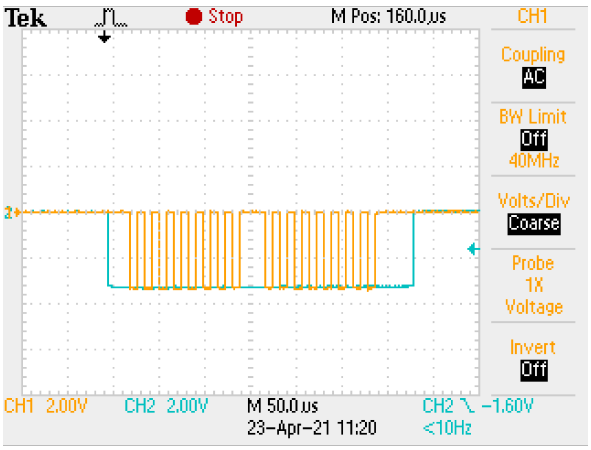
- Figure 8a shows a 2-bytes exchange and it highlights the addressing method in SPI. Indeed, the blue signal corresponds to the Chip Select input pin of the slave that has to be inverted before and after each transaction.
- Figure 8b shows a write operation. The first byte corresponds to the command code, 0x02. Then, the address 0x000000 is sent in 3 bytes. Finally, we send the value 0x42 that we wants to store inside the flash.
- Figure 8c shows the command 0x70 that requests a particular status register in the flash memory. The first byte sent is therefore $0x70$. The next bytes are bullshit and are just there to be able to read the response.
- Figure 8d shows the response of the flash to the previous command. First, the flash reads the command and therefore sends 0x00. Then, it responds with 0x02. Yet, there is only two bytes exchanged in this transaction while there are more in the previous figure. This is simply due to the fact that we slightly modified our code between the two captures but the flash response when there are more clock cycles is the same as if there is only one.

6.1.3 PIC and SPI

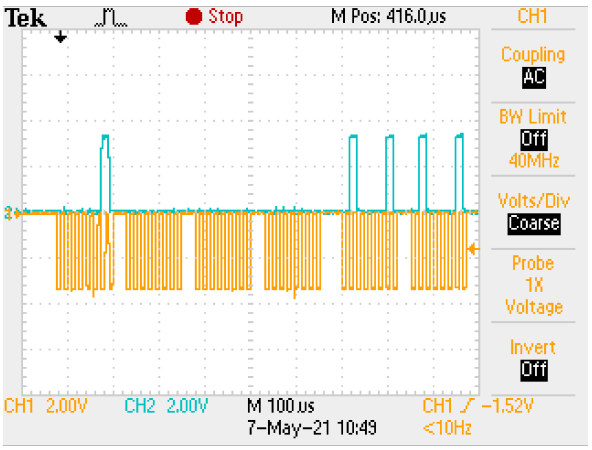
By default, our micro-controller supports SPI and has an internal module allowing to connect another component communicating with a SPI bus. There were a few operations to do in order to enable this feature. First, SPI lines could not be connected anywhere. Looking in the PIC, we choose the appropriate pins. Note that notations in the PIC datasheet were a bit different: MISO line is SDI while MOSI line is SDO. The Chip Select line was connected on a general I/O port.

After having made the correct links, we had to configure the PIC. In practice, it is done by setting up two registers: `SSPCON1` et `SSPSTAT`. In those registers, we define the clock rate at $F_{OSC}/64$ in order to be able to perform $64 \times 8 = 512$ instructions during a single exchange. We also defined the polarity and the phase of the clock. We choose to have a clock that is by default high and that sample variables on rising edge. The flash also works with the opposite polarity and the opposite phase but not if only of one of them is changed.

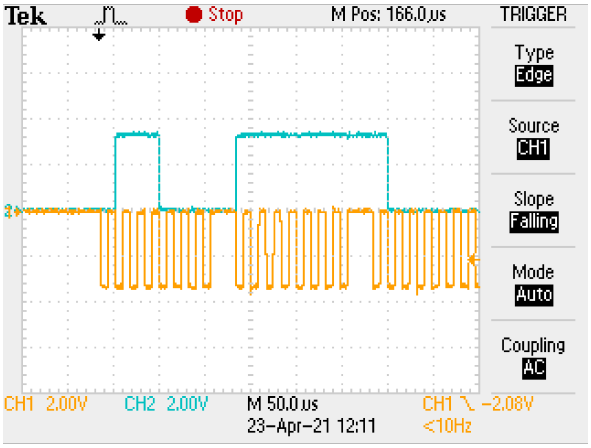
Once all of this is done, using SPI is quite easy. To establish a transaction, we first invert the Chip Select line to enable the slave. Then, to exchange one byte, we set the



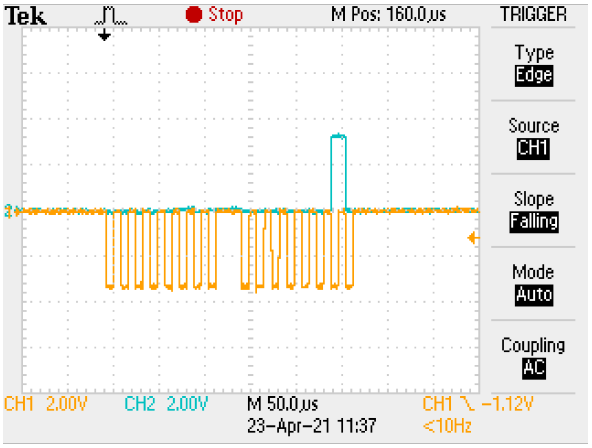
(a) Chip Select during operation



(b) PAGE PROGRAM 0x42 at 0x00



(c) Read flag status register: Master message



(d) Read flag status register: Slave Response

Figure 8: Flash module example

content of the SSPBUF register. When this register is set, the micro-controller automatically starts to generate a clock signal and to send its register over the SDO line. During this transaction time, the slave responds on the SDI line. Once a byte has been fully exchanged, an interrupt is triggered. The byte received can be read in SSPBUF. At that moment, we can either send a new byte or stop the transaction. To send back a new byte, simply setting SSPBUF register again does the trick while inverting the Slave Select pin again allows to stop the transaction. Remember that even if we are not sending data but we only wants to receive something, we still sends some bullshit data. In our case, this bullshit data was simply the previous content of the register.

6.1.4 Pseudo-code

A pseudo-code describing the different tasks performed can be found in appendix B. This pseudo-code omits the way we keep in mind at which address we should write and read because it focuses on how we manage the communication with SPI and how actions are performed globally.

6.1.5 Limitations

We developed all of this solution for a long time but we faced a big problem: we never achieved to read something we previously wrote at a specific address. We are sure we correctly implemented SPI because we achieve to read and/or modify some status registers. The WRITE ENABLE command is also working as we can check in the status register (command $0x05$) that the corresponding bit is correctly modified when sending this command. We could also check that after a PAGE PROGRAM, this bit was correctly reset.

At some point, we thought that it is because we must send a sequence of bytes but we achieve to write into some configuration registers and then to read the written value. Our big problem is that we do not know whether it is the READ operation or the WRITE operation that does not work. We spend a lot of time looking for a solution that we finally could not find. This is why the external flash cannot be found in our final system.

6.2 Internal storage

Because of our problems with the external flash, we chose another solution. It consists in storing the collected measurements directly into the PIC16's linear data memory. Thanks to the File Select Registers (FSR), accessing to this memory area is very easy. Indeed, by loading a value between 0x2000 and 0x29AF into one of the FSRs, the instructions MOVIW and MOVWI allow respectively to load a value from the address pointed to by the corresponding FSR into WREG register and to store the value of WREG register to the address pointed to by the corresponding FSR.

In our case, we simply declared 2 variables base_address and max_address to store the lower and the maximum address at which we begin and stop to save the measurements. As we wanted to maximise the amount of memory used to store the measurements, we decided to keep only the Bank 0 to declare our variables and used the GPRs from Bank 1 to 30, which will be accessed through the FSRs, to save the measurements. By doing so, we are able to store 2400 bytes of data (each bank contains 80 bytes of GPRs). Since

we measure 3 different parameters stored on 2 bytes each (ADC outputs values encoded in 12 bits), 6 bytes of data are produced for each measurement cycle.

Therefore, if measurements are taken every 15 minutes, it means that we can collect some data during 4 days and 4 hours :

$$Total\ Time = \frac{2400}{6} \times 15 = 6000min \approx 4days$$

4 days is sufficient for our application because it means that someone needs to come and collect the measurements made by the mushroom monitoring station at least every 4 days, which is reasonable enough. Indeed, once the station has delivered the data it has collected, it will set its linear data memory's address pointer back to the value of `base_address` variable so that sent measurements are erased little by little by the new ones. In the scenario where there is no empty space left in the PIC16's linear data memory, we simply chose to stop taking some measurements until the older ones are read.

Finally, we could have chosen to reduce the encoding size of our measurements from 12 bits to 8 bits in order to store it on a single byte and therefore save one byte for each measurements. By doing this, we would have been able to collect some data during 8 days and not only 4. This would have been done simply by dividing the value obtained from the ADC by 16. In fact, for further usage of the measurements, their values will anyway be divided by 4096 in order to get back the corresponding voltage. Therefore, if the measurements are stored on a single byte, one simply needs to divide the values by 256 instead of 4096.

In the final version of the station, we decided not to implement this option because we preferred to have a better accuracy than more storage space since 4 days was already a reasonable amount of time for our application.

7 Bluetooth

7.1 Overview

The goal of the bluetooth module is to transmit data to an external device. For ease of testing, we use a mobile phone with this application. This allow us to receive the serial data and visualize them. The bluetooth module is quite easy to use, in the default configuration the bluetooth module is a slave and wait for a master to connect. Once the connection is made, the bluetooth module act as a relay of data, every byte sent by the PIC is directly send over bluetooth and every byte received over bluetooth is transmitted back to the PIC unchanged. This allow to use the bluetooth module in a completelly transparent way.

The communication between the PIC and bluetooth module is done using asynchronous UART with a baud rate of $115200Kbps$, one stop bit and no parity bit. This is a simple transmission protocol where the communicating devices assume that their clock are mostly at the same frequency. There is two communication line, one is used by the PIC to send data and the other to receive data, this is the opposite for the bluetooth module. Each device are in charge of their sending line and keep the voltage high. When a bit is transmitted, the sending device transmission his line to low voltage for one cycle of the

baud rate, sending one bit of zero, this is the start bit. Then the byte is transmitted from the least significant bit to the most significant. Finally, the transmission end with a bit set to 1, this is the end bit. The device can send another byte directly after that, which mean that 10 bit are transmitted for each byte.

7.2 Communication protocol

The communication protocol is very simple. The PIC is waiting to receive a "GET" command, sending any string containing the "GET" sequence in order will trigger the retrieval of data. Once the command is received, the PIC send all data collected since the last communication or the start of the PIC. The data are arranged in section of 6 bytes representing a measure of every sensor. The first 2 byte of each section represent the temperature, the 2 following the humidity and the last 2 bytes are the luminosity.

7.3 Implementation

To implement this, we first connected the PIC adequately, mainly, the GPIO3 is connected to `Vcc` to set the bluetooth module in slave mode and `UART_RX` and `UART_TX` are connected to the `RC6` and `RC7` pins of the PIC respectively to enable UART communication. The bluetooth module doesn't require any further configuration.

The PIC is then configured in software, we need to activate the receiving and sending circuit of the PIC and set it to asynchronous mode. This is done thanks to the `TXSTA` and `RCSTA` special function registers. We also need to set the baud rate, this is done by configuring the source clock to $FOSC/4$ and setting the `BAUDCON` register to $\frac{FOSC/4}{BD_{\text{desired}}}$. In our case this gives $\frac{4000000000/4}{115200000} = 8.68$, we therefore set the `BAUDCON` register to 8. Once all configuration are done we can launch the UART circuit of the PIC, which will start reading on `RC7` and set `RC6` to high until the first transmission.

To receive data from the UART circuit, we need to set the interrupt for EUSART receive. The circuit has two byte of receive buffer, the interrupt fire as long as there is something in the buffer, we cannot clear the interrupt by software. When the interrupt is triggered we can read the first byte in the buffer from the `RCREG` register, this remove the byte from the receive buffer.

In our case, we additionally check in the interrupt if the byte receive correspond to the next expected byte of the "GET" command and if the command is completed set a task flag for sending data.

It is a bit more complicated to send data as it take time to send a byte and there is only a one byte send buffer in the EUSART circuit we also need to use an interrupt. We only enable the EUSART send interrupt when there is something to send, otherwise this interrupt would fire constantly. The interrupt fire as soon as it is enabled and the send buffer is empty. To send something we simply set the position and size of the data to send in a global variable and enable the interrupt. The interrupt handler then send all the data one byte at a time and disable the interrupt at the end.

8 Final Code

The final code for this project as well as other useful documents can be found in the following github repositorty : `https://github.com/jbayaux/embedded`. It contains a `Readme` file that details everything that is available in the repository.

9 Potential improvements

9.1 EEPROM instead of flash

As already mentioned, we did not achieve to make the external flash memory work. However, the idea of having an external memory to save more data. Indeed, this could allow to increase the time before having to come with a Bluetooth device to collect data or it can also increase the measurement rate. Even if variables are not changing so much over time, it may be interesting to analyse those variations on shorter periods of time. This idea could also be used to compute the average of several measurement to extract a more stable value.

During our development, we discovered another type of external memory, which is an EEPROM (Electrically-Erasable Programmable Read-Only Memory). This component is usually easier to connect and to use. Other groups use this components and achieve to do what we could not with our flash memory.

9.2 Additional sensors

Our system currently measures three variables that are the luminosity, the temperature and the humidity. The sensors used for the two last variables are good and therefore outputs interesting results. The luminosity sensor is not the one we were expecting and is not very precise. Yet, it can be used to know whether we are in the dark or under the light but the value in lux is not very accurate. With our current solution, to extract an interesting value, we should first tune the value of the second resistance of the voltage divisor. More precisely, the value of the second resistance should be close to the one of the photo-resistance so that the voltage measured is more sensible to luminosity changes. Moreover, each photo-resistance is different. Then, to have an exact value, we should calibrate the output result. In other words, we should have a conversion table Resistance-Luminosity that is dedicated to that photo-resistance.

Another improvement concerning sensors is basically the idea to add one. Initially, we planned to have a gas sensor in order to measure the CO_2 concentration in the mushroom box. However, the component was not available for a long time and we prefer to focus on making our system work with three basic sensors. Adding one should not be very complicated in the future.

9.3 Timestamping measurements

During our development, we thought about timestamping our measurements. In other words, to match a measure and the time at which it mas made. Nevertheless, we did not plan to have an external clock and we do not have access to the grid power supply to measure time on the PIC itself. The solution we thought about consisted in moving this problem to the software collecting data using Bluetooth. Indeed, we know the time elapsed between two measurements. Then, when collecting data on a phone, we can retrieve the current time. As measurements are time-ordered when stored, we may be able to trace back the history to know when a measure was made.

In our final solution, we do not provide an app or a server to manage data in a fashionable way but the principle can still be made by the person collecting data. He or she simply has to write down what time it is and therefore trace back the history. The little problem of this solution is that data was collected for the last time in the past 15 minutes and this is why they may be a difference between the computed timestamp and the exact timestamp of measurements. This should not be a big problem as variables do not vary quickly over time.

9.4 Server and mobile applications

When thinking about the project, we wanted to allow our client to easily see and manage our measurements. To do so, we planned to make a mobile and/or a web application. This application was supposed to be shared with other groups working with the Pot'Ingé but none of us had time to develop it. yet, we look at solution to use Bluetooth in Flutter but at the end, it took too much time and was not our focus as we prefer to have a working prototype in terms of electronics. This can surely be a great addition in the future.

9.5 Sleep mode

A last improvement that can be very interesting is to put the device in sleep mode when it is not making any measurements. We tried to implement it but we faced several problems. Our basic idea consisted in adding a task that checks whether there is no other task left. If so, it puts the device in sleep mode. The problem is that there might be an interrupt between the check and the sleep operation. A stupid solution could be to disable interrupts but this would prevent the system to wake up. Actually, this idea of sleep mode came back in our mind at the end of the project and we did not had time to make it work perfectly. Therefore, we prefer not to leave it in our final solution.

10 Conclusion

To conclude, although some functionalities are missing for our mushroom monitoring station, this project is a success from our point of view : the final deliverable can still be called a weather station even if it can not store its measurements in a flash memory module or measure the CO_2 concentration. Indeed, it can collect some data about the temperature, the humidity and the luminosity and it is saved in its micro-controller's linear data memory until someone requests them via Bluetooth.

Furthermore, the parts that have not been successful like the flash memory module implementation have still allow us to gain some experience in some fields (e.g working of SPI protocol). In addition, the efforts put in these parts are not useless because all the work we did has been documented and has been made available for the Pot'Ingé.

Finally, doing this project for the Pot'Ingé and not only for the course was really interesting because it added a real value and concrete objective to the project. Moreover, knowing that this project will be used and continued by other people, we did our best to build comprehensive resources and procedures that will be really useful to understand and carry on our work.

A Final pseudo-code

```
; Constants for measurements
; Initiallisation of the counter to make measurements
;       every 15 min 0.202 s
; 1717 * 0.524288 s = 900.202 s
#define MEASURE_COUNTER_INIT 1717
#define MEASURE_MEMORY_SIZE 2400

bool temp_task_flag = false;
bool humidity_task_flag = false;
bool luminosity_task_flag = false;
bool check_full_task_flag = false;
bool measure_stop_writing_flag = false;

int16 measure_counter = MEASURE_COUNTER_INIT;
int16 measure_memory[MEASURE_MEMORY_SIZE];
int16 *measure_current_addr_write = measure_memory;

int8 blue_recv_counter = 0;
bool blue_send_enabled_flag = false;
int8 *blue_send_buffer;
int16 blue_send_size;
int16 blue_send_counter = 0;

char *blue_get_command = "GET\0"
char *blue_no_data = "\0\0\0\0\0\0"

void main() {
    initialisation();
    main_loop();
}

void initialisation() {
    set_analog_input(RA0);
    set_analog_input(RA1);
    set_analog_input(RB4);

    setup_ADC();

    setup_clock(4MHz);
    ; Frequency = clock_frequency/(32*65536) = 1.907348Hz
    setup_timer1(32);

    setup_EUSART_receive();
    setup_EUSART_send();
    setup_EUSART_baudrate(111100);
```

```
    enable_interrupts(timer1|ADC|EUSART_receive);

    return
}

; Trigger every 0.524288s
interrupt void timer1_handler() {
    clear_interrupt(timer1);
    if(--measure_counter != 0)
        return;
    measure_counter = MEASURE_COUNTER_INIT;
    if(measure_stop_writing_flag)
        return;
    temp_task_flag = true;
    return;
}

interrupt void adc_completion_handler() {
    clear_interrupt(ADC);
    *measure_current_addr_write = adc_get_result();
    measure_current_addr_write += 2;
}

interrupt void blue_receive_handler() {
    char recv_char = bluetooth_read();
    if(recv_char == blue_get_command[blue_recv_counter]) {
        if(blue_get_command[blue_recv_counter + 1] != '\0') {
            blue_recv_counter++;
        } else {
            blue_send_enabled_flag = true;
            blue_recv_counter = 0;
        }
    } else {
        blue_recv_counter++;
    }
    return;
}

; Trigger when bluetooth is ready to send
interrupt void blue_send_handler() {
    bluetooth_send(blue_send_buffer[blue_send_counter]);
    if(++blue_send_counter != blue_send_size)
        return;
    blue_send_counter = 0;
    clear_data();
    disable_interrupt(EUSART_send);
    return;
}
```

```
void clear_data() {
    measure_current_addr_write = measure_memory;
    measure_stop_writing_flag = false;
}

void main_loop() {
    while(true) {
        if(temp_task_flag)
            get_temp();
        if(humidity_task_flag)
            get_humidity();
        if(luminosity_task_flag)
            get_luminosity();
        if(measure_check_full_task_flag)
            check_empty_space();
        if(blue_send_enabled_flag)
            blue_send_data();
    }
}

void get_temp() {
    if(!adc_available())
        return;
    ; Select ADC input of temperature sensor
    ; Wait for acquisition time
    ; Set GO bit of ADCON0 register
    temp_task_flag = false;
    humidity_task_flag = true;
    return;
}

void get_humidity() {
    if(!adc_available())
        return;
    ; Select ADC input of humidity sensor
    ; Wait for acquisition time
    ; Set GO bit of ADCON0 register
    humidity_task_flag = false;
    luminosity_task_flag = true;
    return;
}

void get_luminosity() {
    if(!adc_available())
        return;
    ; Select ADC input of luminosity sensor
    ; Wait for acquisition time
```

```
    ; Set GO bit of ADCON0 register
    luminosity_task_flag = false;
    measure_check_full_task_flag = true;
    return;
}

void check_empty_space() {
    measure_check_full_task_flag = false;
    if(measure_current_addr_write - mesure_memory != MEASURE_MEMORY_SYZE)
        return;
    measure_stop_writing_flag = true;
    return;
}

void blue_send_data() {
    measure_stop_writing_flag = true;
    blue_send_size = measure_current_addr_write - measure_memory;
    if(blue_send_size == 0)     {
        blue_send_buffer = blue_no_data;
        blue_send_size = 6;
    } else {
        blue_send_buffer = measure_memory;
    }
    blue_send_enabled_flag = false;
    activate_interrupt(EUSART_send);
    return;
}
```

B Flash pseudo-code

```
bool enable_write_task = false;
bool store_data_task = false;
bool read_data_task = false;
bool clear_flash_task = false;
bool compute_next_data_task = false;
bool use_data_task = false;

int8 next_data;
bool is_sending = false;
bool is_reading = false;
bool will_write = false;
bool will_read = false;

!! set up pins used by flash
!! set up SPI configuration registers
CS <- 1
```

```
void interrupt spi_completion() {
        // Triggered when one byte has been fully exchanged
        if (is_sending) {
                compute_next_data_task = true;
                SSPBUF <- next_data;
                return;
        }
        else if (is_reading) {
                use_data_task = true;
                SSPBUF <- SSPBUF;
                return;
        }
        else {
                clear_flash_task = true;
                return;
        }
}

void enable_write() {
        // Send the ENABLE WRITE command
        enable_write_task = false;
        will_write = true;
        CS <- 0;
        SSPBUF <- 0x06;
        return;
}

void store_data() {
        // Send the PAGE PROGRAM command
        // and initiate a write operation
        store_data_task = false;
        compute_next_data_task = true;
        is_sending = true;
        CS <- 0;
        SSPBUF <- 0x02;
        return;
}

void read_data() {
        // Send the READ command and initiate
        // a read operation
        read_data_task = false;
        compute_next_data_task = true;
        is_sending = true; // First we need to send data and not read
        something_left_to_send = true;
        will_read = true;
        CS <- 0;
        SSPBUF <- 0x04;
```

```
        return;
}

void clear_flash() {
        // Ends up a SPI transaction with flash
        clear_flash_task = false;
        CS <- 1;
        if (will_write) {
                will_write = false;
                store_data_task = true;
        }
        return;
}

void compute_next_data() {
        // Compute what should be sent next
        compute_next_data_task = false;
        if (something_left_to_send) {
                !! put in next_data what has to be sent next
                !! possibly update something_left_to_send
        } else {
                is_sending = false;
                if (will_read) {
                        will_read = false;
                        is_reading = true;
                }
        }
        return;
}

void use_data() {
        // Use data that has been received
        use_data_task = false;
        !! use data (send it to Bluetooth)
        if (not keeps_reading()) {
                is_reading = false;
        }
        return;
}
```