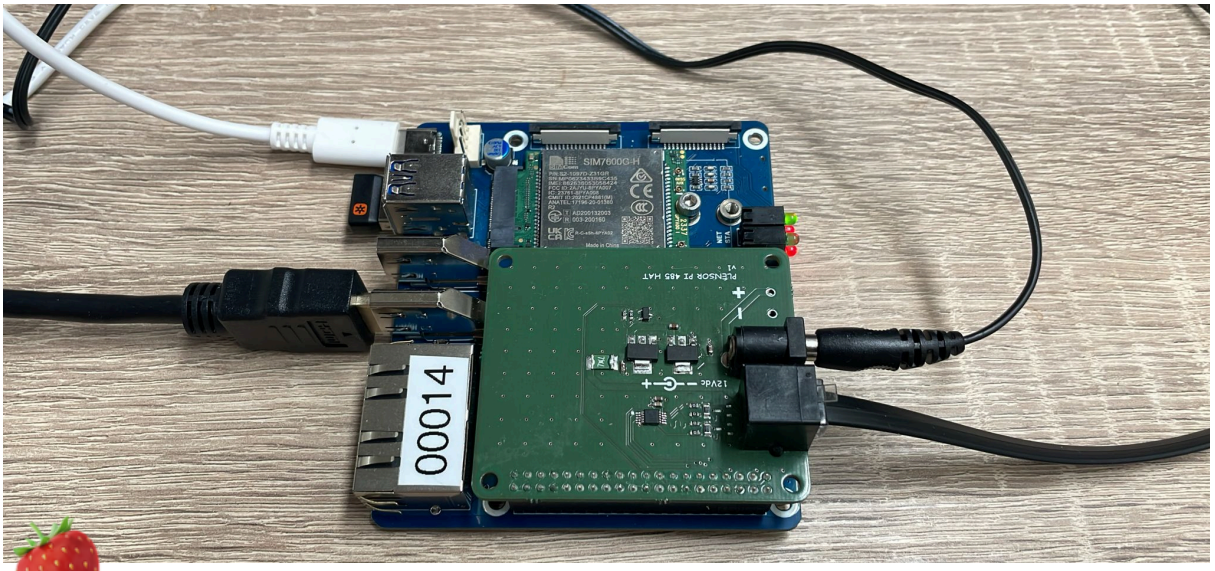


Compute module sensor communication (1)

Owner	Miranda
Data science ideas	
Github link	Commits · plense-tech/edge-code (github.com)
Hardware version	Plensor V4
Last edited	@22 oktober 2024 16:41
Last edited by	Miranda
Meerdere selecteren	
Notebook	
Preparations	7e8aece02ea2457968ad10c7cecb09b0029c1f6f
Status	Active
Tags	Wiki mainpage

Introduction

This wiki page provides a comprehensive overview of the code used on our Compute Module 4 (CM4) for communicating with the sensor. The code is designed to facilitate efficient and accurate data transfer between the CM4 and the sensor, ensuring reliable operation in various applications. This documentation will cover the

architecture, key components, and functionalities of the code, as well as guidelines for implementation and troubleshooting.

The system is built up around a system where we send a command to one specific sensor and then wait for its response, buffer it and then process it and send it to the Cloud. In this page we will not go into the details of any processing or sending it to the Cloud.

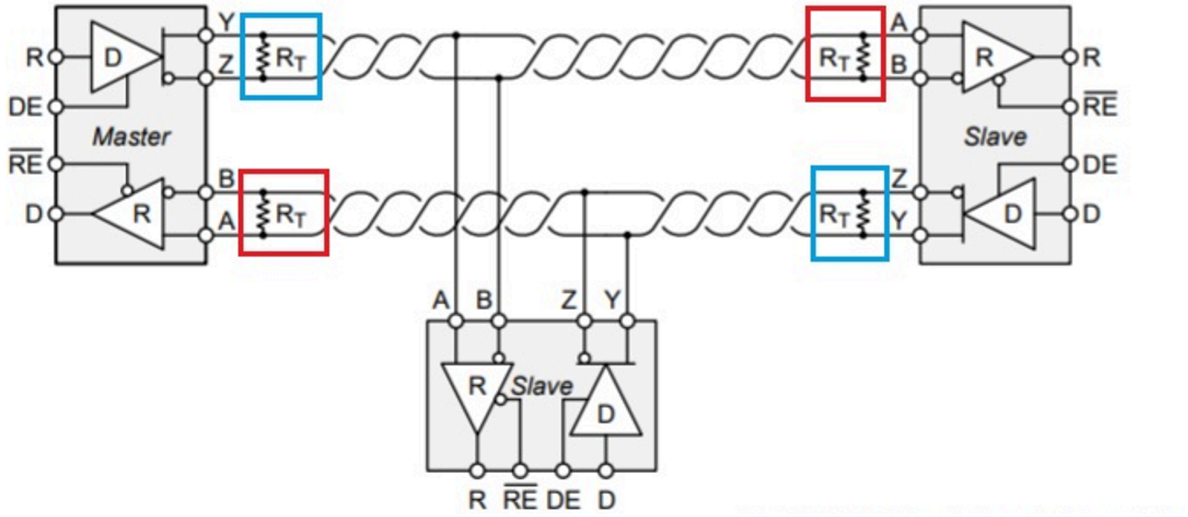
Communication method

Serial communication is a method used for transmitting data between devices. One commonly used protocol for serial communication is RS485, which supports full-duplex communication. Full-duplex RS485 allows for simultaneous two-way data transmission, meaning that data can be sent and received at the same time. This is particularly useful in scenarios where timely and efficient communication is critical.

RS485 is known for its robustness and long-distance communication capabilities. It uses differential signaling, which helps to reduce noise and interference, making it reliable even in electrically noisy environments. The RS485 standard supports up to 32 devices on a single bus, making it suitable for complex systems with multiple nodes.

In our setup, we utilize full-duplex RS485 for communication between the Compute Module and the sensors. The full-duplex configuration provides separate lines for transmitting and receiving data, ensuring that communication is both fast and efficient. This setup is ideal for our application, where we need to send commands to the sensors and receive data back without delay.

To implement RS485 communication, we need to configure the serial ports on the Compute Module and ensure that the RS485 transceivers are properly connected and turned on. Once the hardware setup is complete, the software can handle the sending and receiving of data, ensuring that the Compute Module can effectively communicate with the sensors.



An overview of full-duplex RS485 where slave and master can talk at the same time.

Setting up RS485 communication

To be able to send RS485 data to the sensors we use our own Pi 485 Hat which converts UART data into RS485 full-duplex data. This hat can be controlled using two simple GPIO pins on the Compute Module.

To use this pins we import the relevant library:

```
import RPi.GPIO as GPIO
```

And then setup the following pins:

- GPIO 18: This pin enables and disables the transmitter on the Compute Module side. To be able to send data it needs to be set to HIGH.
- GPIO 4: This pin enables or disables power to flow to the sensor. When it is HIGH, 12V will be sent over the telephone cable powering the sensors.

The setup for the GPIO pins in Python will thus look as follows:

```
def setup_gpio(self):
    """
    Sets up the GPIO pins for communication with the sensor.

    This function configures the GPIO mode and sets the initial states of the GPIO pins
    """
    # Turn on the transceiver
    GPIO.setmode(GPIO.BCM) # Use Broadcom pin-numbering scheme
    GPIO.setup(18, GPIO.OUT)
    GPIO.setup(4, GPIO.OUT)
```

```
    # Set GPIO pins to high
    GPIO.output(18, GPIO.LOW) # Transmit enable starts Low
    GPIO.output(4, GPIO.HIGH) # Power enable starts high


# Setup GPIO
self.setup_gpio()
```

After that we need to enable GPIO 18 every time before transmitting data:

```
# Send the message
GPIO.output(18, GPIO.HIGH)
# Code for sending the message
GPIO.output(18, GPIO.LOW)
```

Setting up serial communication

For the transmission of data from our UART port to the RS485 driver (the hat), we have to set up the serial communication on the Compute Module. To do so we use the `Serial` library in Python and the `time` library for later:

```
import serial.Serial
import time
```

For our serial communication we use the following settings:

- Port: '/dev/ttyAMA0', for communication we use port '/dev/ttyAMA0', this is the default port for UART serial communication.
- Baudrate: 921600, this baudrate was chosen as it is the fastest commonly used baudrate. Going faster will allow us to measure more frequently, however it can also lead to data corruption. There is a trade-off in this aspect which we can further explore once we have a lot of sensors measuring.
- Parity: None, parity in serial communication is a technique used to detect errors in transmitted data. This helps in identifying any single-bit errors that may occur during transmission. Instead of using parity, we check for data integrity using a checksum (https://nl.wikipedia.org/wiki/Controlegetal).
- Stopbits: One, a stopbit is used in serial communication to signal the end of a data packet, allowing the receiving device to recognize the conclusion of one byte before the start of another. It ensures proper synchronization between the transmitting and receiving devices, reducing the chance of data corruption.

- Bytesize: Eight, this is the commonly used size of each package payload, 8 bits.
- Timeout: 2, this setting will be changed throughout the code as different commands require different timeouts. When reading data the action will stop if the Compute module does not read any data for 2 seconds.

Using the Serial library we set up the communication using the parameters as described above:

```
# Set up the serial connection
self.ser = serial.Serial(
    port='/dev/ttyAMA0',
    baudrate=921600,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=2
)
```

Consequently after setup we can use this object to send data to the sensors. We can do so by converting our message (`frame`) into a byte array and combining this with our script for managing the RS485 drivers along with calling the `ser.write` functionality:

```
# Convert to bytearray for transmission
message_bytes = bytearray(frame)

# Setup transmission
GPIO.output(18, GPIO.HIGH)
self.ser.write(message_bytes)
time.sleep(0.05)  # Wait a bit to ensure the message is sent
GPIO.output(18, GPIO.LOW)
```

Similarly, we can use the `ser.read` functionality to read data. For reading data however, it is often wise to include a timeout to prevent the code from blocking when a sensor sends back no data. Furthermore, it can be useful to not read every single byte separately and constantly. For this reason we add `time.sleep` at the end of each while iteration and check if any bytes are in the buffer using `ser.in_waiting:`

```
# Initialize our reading code
start_time = time.time()
response = bytearray()
last_data_time = time.time()
```

```
# Run the reading code
while True:
    # Check for data waiting
    if self.ser.in_waiting > 0:
            response.extend(self.ser.read(self.ser.in_waiting))

    # Check if minimum timeout has been reached and no data is heard for 10ms
    if (time.time() - start_time > self.timeout)  and (time.time() - start_time > self.tim
eout):
            break

  time.sleep(0.001)  # Adjust polling interval as needed
```

With this code we can now send to and read data from our sensors. To fully manage robust communication at high baud rates, however, we need to properly structure the message we send.

Initializing classes

We have built some peripheral classes to manage other important processes in the code without making the code explode. These classes are very straight forward and currently include:

- ErrorLogger: handles all information and error logging related tasks.

- ComponentHandler:
- JSONHandler: handles all .json file related tasks.

From the compute module to the sensor (sensor commands)

The basic message the Compute module sends to a sensor consists of a start of frame byte, a sensor ID, a payload length, a payload and finally a checksum for the sensor to check if the message is correctly received. An overview can be seen below:

An example of a message would look as follows:

Start of frame	**Sensor ID**	**Payload length**	**Payload**	**Checksum**
0x5A	0x00 0x00 0x01	0x00 0x01	0x63	0xBF
1 byte, always the same	3 bytes, variable	2 bytes, variable	x bytes, variable	1 byte, variable

When the sensor recognizes the start of frame and its sensor ID it will use the payload length to read the payload. Subsequently the sensor will check the checksum to see if the data is not corrupted and if not it will extract the parameters and its command from the payload.

Note that the sensor ID 0x000000 will target all sensors, however when this ID is targeted, no sensor will send anything back as that would lead to overfilling of the line and non-usable data. This sensor ID will need to be packed into the correct format. For message sending we will always use Big Endian byte ordering. Using the following code we can pack the sensor ID in this manner:

```
# Pack parameters into to bytes
self.sensor_id_bytes = [(self.sensor_id >> 16) & 0xFF,
                        (self.sensor_id >> 8)
                        & 0xFF, self.sensor_id & 0xFF]
```

Using this packed data we can build the message for a given payload, the code for this can be found in the block below:

```
# Start byte
start_byte = [0x5A]

# Payload length (2 bytes)
payload_length = list(len(payload_bytes).to_bytes(2, 'big'))

# Combine all parts to form the frame without checksum
frame_without_checksum = (start_byte + self.sensor_id_bytes +
                          payload_length + payload_bytes)

# Calculate checksum
checksum = self.calculate_checksum(frame_without_checksum)

# Append checksum to the frame
frame = frame_without_checksum + [checksum]

# Convert to bytearray for transmission
message_bytes = bytearray(frame)
```

This exact frame is the one we send in the **Setting up serial communication** chapter. Here we will however go a bit deeper into the calculation of the checksum. The checksum is calculated by XORing all bytes in the data. For a deeper understanding of XORing see the following subpage:

✅ XOR checksum calculation

The code for XORing can be found below:

```
def calculate_checksum(self, data):
    """
    Calculates the checksum by XORing all bytes in the data.

    Parameters:
        data (list of int): The data over which to compute the checksum.

  Returns:
    int: The computed checksum.
  """
  checksum = 0
  for byte in data:
    checksum ^= byte
  return checksum
```

We will now continue with the way we fill the payload sent to the sensors. These are built as single byte commands telling the sensor what to do, complemented with parameters. The sensor has a multitude of different commands it can receive. An overview of the command bytes can be seen in the table below:

Sensor command bytes:

Name	Type	Hex code	Parameter 1	Parameter 2	Parameter 3	Output
Get sensor ID byte	Problem fixing Setup	0x5B				None
Sine sweep byte	Measurement	0x5C	Start frequency (Hz, 3 bytes)	Stop frequency (Hz, 3 bytes)	Duration (ms, 2 bytes)	signed int16 Audio data
Impulse time of flight byte	Measurement	0x5D	Timeout duration (micros, 2 bytes)			unsigned int32 ToF in ns
Block sweep byte	Measurement	0x5E	Start frequency	Stop frequency	Duration (ms, 2 bytes)	signed int16 Audio data

Name	Type	Hex code	Parameter 1	Parameter 2	Parameter 3	Output
			(Hz, 3 bytes)	(Hz, 3 bytes)		
Temperature humidity byte	Measurement	0x5F				4 times a unsigned int16 measurement in the order: [Temp_inner, Hum_inner, Temp_outer, Hum_outer] devide the data by 100 to get correct value
VCO Calibration byte	Initialization Setup	0x60				None
Set sensor ID byte	Setup	0x61	Sensor ID (3 bytes)			None
Reset sensor byte	Problem fixing	0x62				None
Set voltage level byte	Initialization Problem fixing Setup	0x63	Damping level (1 byte, 0-3, see explanation)			None
Blockwave time of flight byte	Measurement	0x64	Timeout duration (micros, 2 bytes),	Half periods of a blockwave (1 byte)		unsigned int32 ToF in ns

In the coming sections we will go into the functionality of each of these command bytes.

Command bytes

Each command byte has a different functionality, we will go through each starting with the get sensor ID byte.

Get sensor ID byte: 0x5B

This byte asks a specific sensor, for example sensor 1, to present its sensor ID. In case a sensor with the ID 1 is connected to the Compute Module making this request, it will send back ACK. If it is not connected we will receive a timeout from the Compute Module.

The functionality of this byte is that it allows us to scan for the connected sensors and see if all sensors are up and running. This command will respond within a millisecond making easy to scan through all connected sensors within 0.1 seconds.

Sine sweep byte: 0x5C

This measurement asks the sensor to perform a linear sine sweep of the speaker using the input parameters supplied with the command byte.

NOTE: The sensor needs to be calibrated first after powering up for this measurement to work

Impulse time of flight byte: 0x5D

This byte will generate a 1 microsecond impulse on the speaker. This acts like a hammer exciting the speaker and with it all frequencies at once. Due to a lack of power build-up however it is not very convenient for measurements through objects as it will never trigger. For this reason a timeout duration is given along with the command byte. If this timeout value in microseconds is exceeded the sensor will return approximately this timeout, as it can never exactly give back this number. For example if we use a timeout of 200 microseconds, and no response is heard, the sensor will send back something like 200 048 nanoseconds. Since we know the timeout we gave the sensor we know no sound triggered the microphone. Do note that external noise can also trigger the microphone so if we measure it should be done multiple times to filter out noise.

Block sweep byte: 0x5E

This measurement asks the sensor to perform a linear blockwave sweep of the speaker using the input parameters supplied with the command byte.

NOTE: The sensor needs to be calibrated first after powering up for this measurement to work.

Temperature humidity measurement byte: 0x5F

This measurement allows the user to ask a sensor to present its temperature and humidity values towards the underside of the PCB (inside the bellow near the microphone) and towards the outside microclimate (topside of the PCB). The measurement does not require any input parameters.

VCO calibration byte: 0x60

This byte needs to be used upon each time the sensor is powered up for the block and sine sweep measurements to work. By sending this byte the sensor will build the required database for the sweep to work. It can also be used to calibrate the sweep as temperature fluctuation may cause the sweep to start and stop at slightly higher frequencies. The calibration takes between 13 to 15 seconds and ends with the sensor sending an ACK byte.

Set sensor ID byte: 0x61

The set sensor ID byte will mainly be used for the initial setup of the sensor. Using this byte we target the start ID (usually 0xFFFFFF if from the factory) and assign a new sensor ID in the parameter supplied with this byte. The sensor will then send back an acknowledgement with this new sensor ID.

Reset sensor byte: 0x62

This byte will ask the sensor to reboot itself. This can be helpful when a sensor is stuck or unresponsive. Usually this might be used in combination with the 0x000000 sensor ID which targets all sensors at once.

In case this reset byte does not work another method could be to remove power from the GPIO pin powering controlling power to the sensors (GPIO xxx).

Set voltage level byte: 0x63

Using this byte we can regulate the input voltage level to the speaker. As smaller diameter plants will lead to clipping with a 10 Vpp input this command allows for setting three different modes of input voltage. This are sorted by their parameter value:

- 0: 10 Vpp input voltage
- 1: 0.8 Vpp input voltage
- 2: 0.046 Vpp input voltage
- 3: 0.042 Vpp input voltage

Value 2 and 3 lie close to each other due to the hardware design. In general stance 0 is meant for stems between 10-20 mm, stance 1 for 3-10 mm stems, while stances 2 and 3 are for measuring through the air.

For a more in depth read about this byte see the subpage below:

🤫 Damping factors

Blockwave time of flight byte: 0x64

Run a TOF measurement with a specified number of half periods of a 40KHz blockwave. Iterating over these blocks can be used to perform a TOF measurement by building up power in the transducer to reach the TOF trigger level. Running a sequence of bursts gives you the control to isolate the received trigger of separate bursts.

```
x
```

Parameter packing

In order to pack the parameters (all integers), they need to be converted to Big Endian byte objects. We briefly gave an example of this in the form of the sensor ID in a previous chapter:

```
# Pack parameters into to bytes
self.sensor_id_bytes = [(self.sensor_id >> 16) & 0xFF,
                        (self.sensor_id >> 8) & 0xFF,
                        self.sensor_id & 0xFF]
```

In a similar manner we can use the `set_parameters` definition to set all parameters at once. This object will pack the parameters in the relevant number of bytes:

```
def set_parameters(self, start_freq, stop_freq, duration, damping):
"""
Sets the frequency and duration parameters for the communication.

Parameters:
    start_freq (int): The starting frequency.
    stop_freq (int): The stopping frequency.
    duration (int): The duration of the signal.
"""
try:
    self.start_freq = [(start_freq >> 16) & 0xFF, (start_freq >> 8)
                & 0xFF, start_freq & 0xFF]
    self.stop_freq = [(stop_freq >> 16) & 0xFF, (stop_freq >> 8)
                & 0xFF, stop_freq & 0xFF]
    self.duration = [(duration >> 8) & 0xFF, duration & 0xFF]
    self.damping_level = damping & 0xFF
except Exception as e:
  self.logger.log_error(f"Error while setting parameters: {e}")
```

Upon successful packing we can use the parameters and command bytes to build the payload.

Payload construction

Using the parameters and the command byte we can then construct the payload, which is different for the different commands, as shown in the Sensor command bytes table. Putting this table into code leads to the following:

```
# Construct payload based on command byte
# GET, TEMP, CAL, RST BYTE
if self.command_byte in ([0x5B], [0x5F], [0x60], [0x62]):
        payload_bytes = self.command_byte
# Damping byte:
elif self.command_byte == [0x63]:
        payload_bytes = self.command_byte + self.damping_level
# SET BYTE
elif self.command_byte == [0x61]:
            payload_bytes = self.command_byte + self.new_sensor_id
# SINE, BLOCK BYTE
elif self.command_byte in ([0x5C], [0x5E]):
        payload_bytes = (self.command_byte + self.start_freq +
                self.stop_freq + self.duration)
# TOF impulse byte:
elif self.command_byte == [0x5D]:
        payload_bytes = self.command_byte + self.tof_timeout
# TOF blockwave byte:
elif self.command_byte == [0x64]:
        payload_bytes = (self.command_byte + self.tof_timeout +
                self.tof_half_periods)
```

Using this payload we can fill it in our previously explained message structure and send it using our `send_message` definition.

From the sensor to the Compute module (measurement data)

The sensor measurement (measurement type in the command byte table) bytes require specific unpacking to be able to use the data. In the payload a sensor will always send back an ACK or NAK byte to confirm whether the command was properly understood. After this single ACK/NAK byte the payload follows, which is different for

each measurement type. For the commands without any output the payload will thus consist only of the ACK/NAK byte. We will start with the method we use to read data.

Reading data and extracting the payload

As explained in the setup chapter, we can read data using `ser.read` with the following code:

```
# Initialize our reading code
start_time = time.time()
response = bytearray()
last_data_time = time.time()

# Run the reading code
while True:
    # Check for data waiting
    if self.ser.in_waiting > 0:
            response.extend(self.ser.read(self.ser.in_waiting))

    # Check if minimum timeout has been reached and no data is heard for 10ms
    if (time.time() - start_time > self.timeout)  and (time.time() - start_time > self.tim
eout):
            break

  time.sleep(0.001)  # Adjust polling interval as needed
```

Upon successfully reading the full response the 10 millisecond timeout will trigger breaking the reading loop. Next up we need to check if the sensor acknowledged the command we sent. For this we will use the extract_payload definition. For this we first check if the response is not empty and then check if the payload fits the message frame we defined in the previous chapter. This is done in the following manner.

We start by checking if the message is correct. Upon each error we encounter in these steps we will return `None` and use our error logger class to log the relevant error, described in subpage:

⚠️ Error logging

To check if the message is correct we check the message in the following order:

1. First we check if the message is larger than 6 bytes, if not we know it can not be correct.
2. Next up we Verify whether the start byte is indeed 0x5A.

3. Subsequently we check if the sensor ID sending the data back is the same that we asked to send data
4. Next up we extract the payload length and check if the payload length plus the expected 7 other bytes matches the response
5. Then we calculate the checksum of the response, excluding the checksum byte itself
6. Now we extract the ACK/NAK byte to see if the sensor has understood the message, if not we do not log an error but instead just return the NAK/Error result
7. Finally we are sure the response makes sense and we extract it using information about our message structure and the length of the payload we extracted before

To conclude we send back information about the ACK/NAK status and the payload itself. The full code for this definition can be found below:

```
# Check if we have a response and read it
if response:
    ack_nak, payload = self.extract_payload(response)

def extract_payload(self, response):
    """
    Extracts the payload from a sensor response, verifying its
    structure and checksum.

    Parameters:
      response (list of int): The full message from the sensor as a
          list of byte values.

    Returns:
      tuple: (ack_nak, payload) where ack_nak is the status
          ("ACK", "NAK", or "Error")
          and payload is the extracted data as a list of bytes.
          Returns (None, None) if any validation fails.
    """

    if len(response) < 6:
      self.logger.log_error("Invalid response length")
      return None, None

    # Verify the start byte
    start_byte = response[0]
    if start_byte != 0x5A:
```

```
            self.logger.log_error(f"Invalid start byte : {start_byte}")
            return None, None

        # Extract and verify the sensor ID
        sensor_id_received = ((response[1] << 16) | (response[2] << 8) |
                              response[3])
        if sensor_id_received != self.sensor_id:
            self.logger.log_error(f"Invalid sensor ID: {sensor_id_received}")
            return None, None

        # Extract payload length
        payload_length = (response[4] << 8) | response[5]

        # Ensure the response length matches the expected length
        expected_length = 6 + payload_length + 1  # 6 header bytes + pyl + chk
        if len(response) != expected_length:
            self.logger.log_error("Response length does not match expected length")
            return None, None

        # Extract and verify the checksum
        calculated_checksum = self.calculate_checksum(response[:-1])
        if response[-1] != calculated_checksum:
            self.logger.log_error("Invalid checksum")
            return None, None

        # Extract ACK/NAK byte (first byte of the payload)
        ack_nak = response[6]
        if ack_nak == 6:
            ack_nak = "ACK"
        elif ack_nak == 15:
            ack_nak = "NAK"
            return ack_nak, None
        else:
            ack_nak = "Error"
            return ack_nak, None

        # Extract the rest of the payload
        payload = response[7:6 + payload_length]

        return ack_nak, payload
```

Upon successful payload extraction we can start unpacking it.

Unpacking the payload

Since each measurement has a different unpacking method a different definition is used for each. We will start with the block and sine sweep measurement which share the same output format.

Sine and block sweep

Temperature humidity

Blockwave time of flight and impulse time of flight

A full measurement loop

Saving data and pushing it to the Cloud

After a successful measurement loop the data is saved ...