Overview

The Panther Logger is a powerful data logger with multiple options for reading sensors and sending data to the internet using WiFi, cellular or long range radio. It can be programmed with the Arduino IDE software and is pre-loaded with the Adafruit Feather M0 bootloader. It also contains integrated lithium ion battery charging capabilities from either solar or other DC power sources (e.g. USB) for long term field deployments.

This is the first tutorial on the Panther Logger in our learning center providing an overview of the architecture of the Panther Logger board. Further tutorials give step by step instructions for using the Panther Logger. You may want to try out the other tutorials and frequently come back to consult this first tutorial.

Below we give a basic description of the board layout and the pins available for reading sensors as well as the board specifications. See the video to the right for an introduction.

Pinouts

The processor on the Panther Logger is a SAM D21 Cortex M0+. The WiFi module is a WINC1500 and the LoRa modem is the RAK11720. Most of our cellular tutorials currently use the Nimbelink BG96 Quectel modem (CatM and NB-IoT).

The Panther Logger pinouts are arranged in eight, 4-pin screw terminal blocks plus the two-pin battery screw terminal and two-pin solar input screw terminal blocks (see image on the right). These are pluggable screw terminals that for convenience can be unplugged to loosen and tighten wires and then plugged back in. We recommend unplugging the screw terminal blocks to attach any wires.

The function of each screw terminal pin is listed below. The tutorials in the learning center show detailed examples of using these pins to hookup and read various kinds of sensors for monitoring applications and code descriptions for how to send that data to the internet using the available communication technologies embedded on the board.

Navigating the Terminal Blocks:
The connector blocks are arranged in two rows of 4-pin screw terminals. The pin definitions below start with the top row (row 1), listed left to right. See image to the right for where these blocks are located on the Panther Logger board. You may wish to label the blocks with a permanent marker. The abbreviations below are also marked on the board.

Panther Logger Pinout Descriptions (per block, left to right):
Row 1, Block 1

- VCC = 3.3V power supply (always on)
- GP6 = General input/output

- GP5 = General input/output
- 5V = 5V power supply (always on)

Row1, Block 2

- A3 = Analog input
- A2 = Analog input
- A1 = Analog input
- A1 = Analog input

Row1, Block 3

- TX3 = Transmit UART
- RX3 = Receive UART
- GND = Battery GND
- 12VS = 12V power supply, switched on/off (described here)

Row1, Block 4

- TX2 = Transmit RS232
- RX2 = Receive RS232
- TX1 = Transmit RS232
- RX1 = Receive RS232

Row2, Block 5

- 3VS = 3.3V power supply, switched on/off (described here)
- GND = Battery Ground
- SCL = I2C clock pin
- SDA = I2C data pin

Row2, Block 6

- 3VS = 3.3V power supply, switched on/off (described here)
- D6 = Digital pin 6
- GND = Battery Ground
- 5V = 5V power supply (always on)

Row2, Block 7

- 5V = 5V power supply (always on)
- D11 = Digital pin 11
- GND = Battery Ground
- 12VS = 12V power supply, switched on/off (described here)

Row2, Block 8

- GP3 = General input/output
- GP2 = General input/output
- GP1 = General input/output
- GP0 = General input/output

Battery Block

- Battery Ground (left)
- Battery Positive (right)

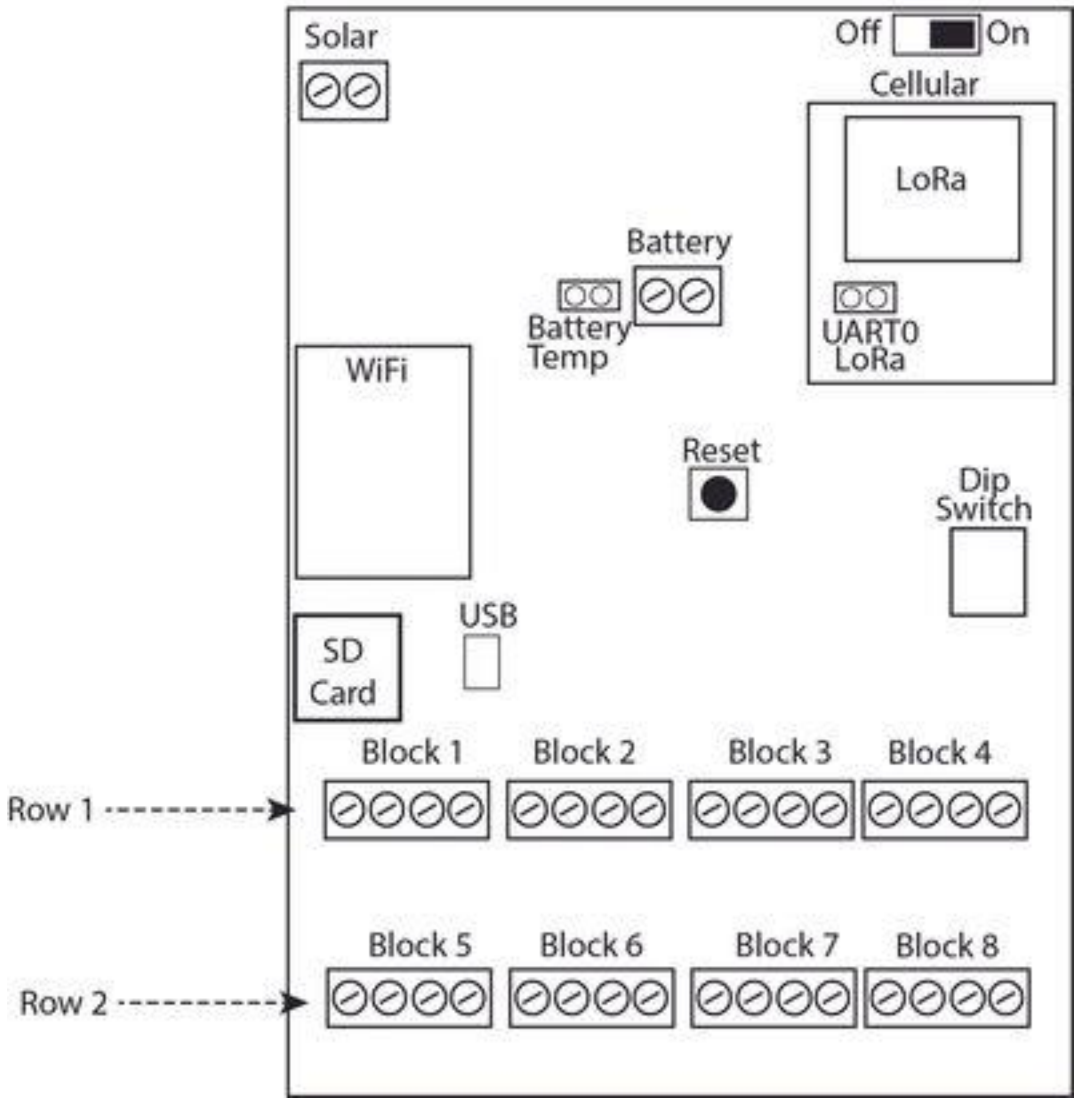
Solar Block

- Solar Negative (left)
- Solar Positive (right)

Note, the board also contains a two-pin header on the UART0 pins of the LoRa module for uploading new firmware and an unpopulated (bare) two-pin header for a battery temperature monitor with ground connection (not currently used, but can be).

Keep in mind that for safety and convenience there is an on/off switch. We will generally ship the board with the switch off. Make sure you turn it on when you are ready to use the board.

Telelogger
Location of Screw Terminals and Major Components



How does the Panther Logger board work?

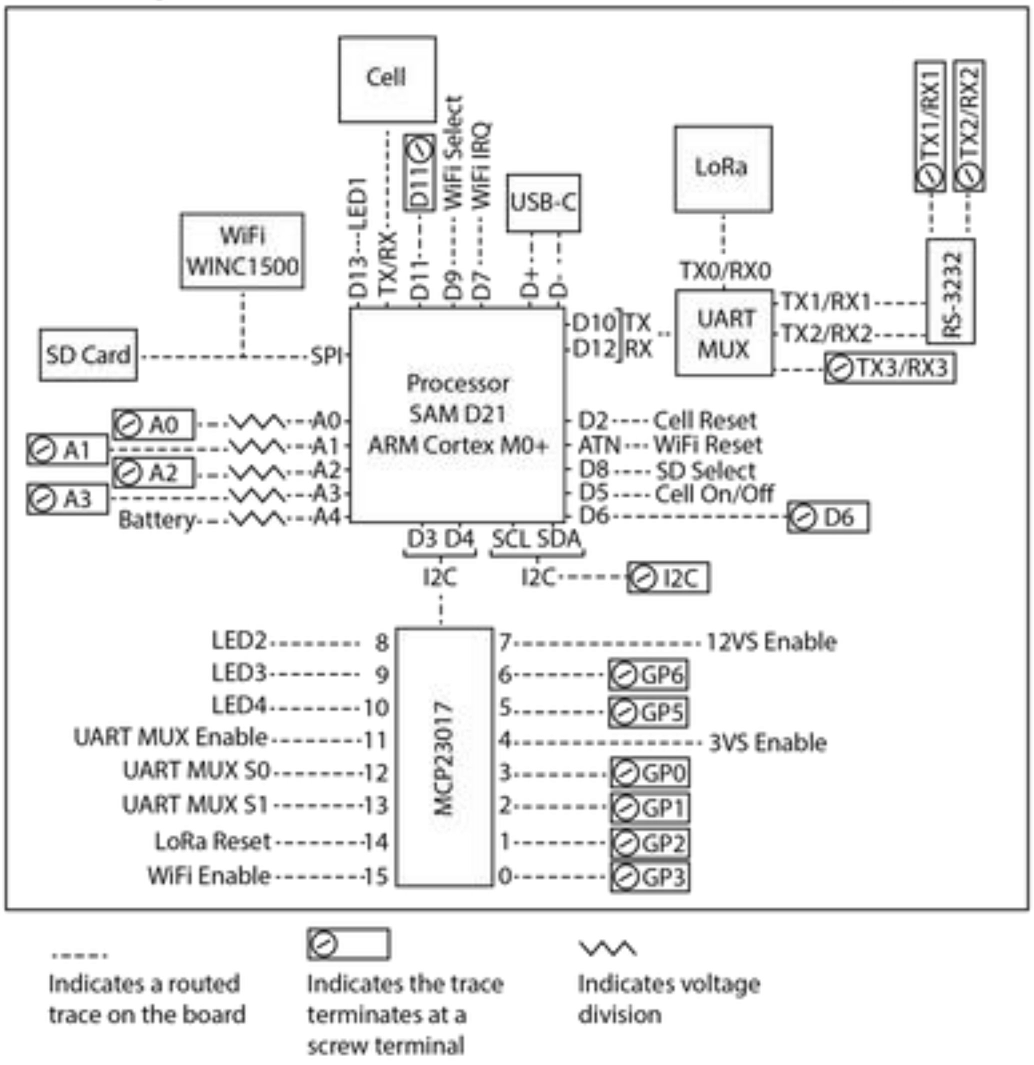
At the core of the Panther Logger board is the ARM Cortex M0+ processor. It communicates with peripheral components to achieve various tasks such as turning on a voltage rail, reading an analog or digital sensor or sending data to the internet using WiFi, cellular or LoRa.

The Panther Logger board has an expanded set of serial interfaces that allow it to communicate with hundreds of serial sensors as well as multiple communication methods. The main UART line (RX/TX) of the processor is dedicated to the cellular modem. An additional UART line is created from pins D10 and D12 and this line is expanded to four with use of the UART MUX chip. Of these four channels two are for reading RS232 sensors, one for communicating with the LoRa modem and one that is left open to further expand this port even further with our UART

MUX expansion board or to (for example) add our RS485 communications board if you have to read RS485 sensors.

The MCP23017 GPIO expansion chip is used to turn on/off LEDs, turn on/off 12V and 3.3V power rails, the WiFi module and to set the channel of the UART MUX that is in use. Other GPIO expansion pins are brought out to screw terminals for other uses (examples given in Learning Center tutorials). Four analog pins (A0 - A4) and two general usage digital pins (D6 and D11) on the processor are also brought out to screw terminals. The analog pins are put through a voltage divider allowing use of sensors with up to 5V output to be used on this 3.3 volt logic system. The two digital pins D6 and D11 may be used for communicating with SDI12 or OneWire (for example). The processor communicates with the WiFi module (WINC1500) and the SD card via the processor's SPI pins. The processor's I2C pins (SDA and SDL) are brought out to a screw terminal and are pulled up with 10K pull-up resistors. You can use this for reading multiple I2C sensors. Pins D3 and D4 are used to created a second I2C port that is dedicated to operating the GPIO expander.

Block Diagram of Communications Between Board Components

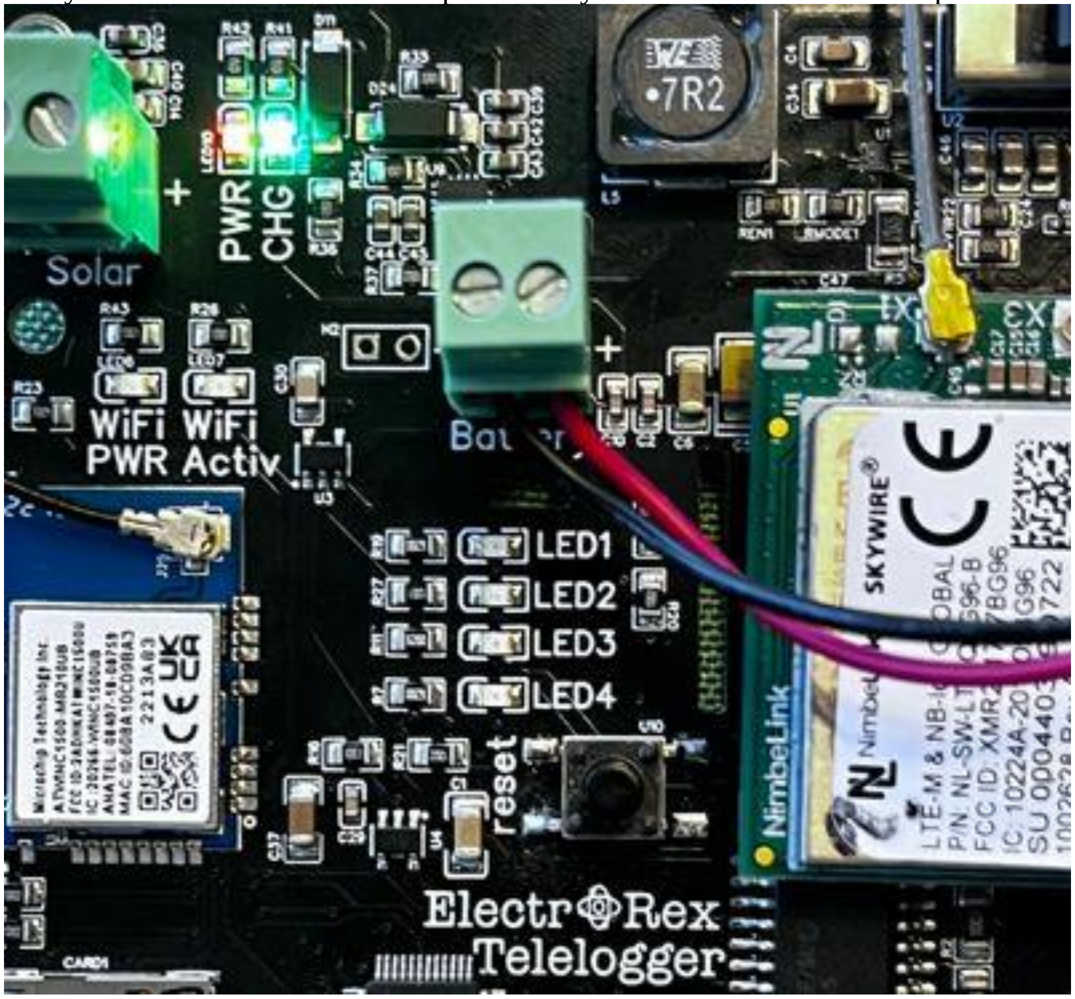


Power

Power is supplied to the Panther Logger from a 3.7 volt lithium ion polymer battery (purchased separately). See battery recommendations here. The battery can be charged from either USB or solar panels delivering less than 10 volts (typically USB provides 5V). USB charging occurs when a micro USB cable is plugged in from either a computer or wall outlet. When USB or solar power is applied the green charge LED at the top of the board will light up. When a battery is plugged in then the red power LED next to the charge LED will also light up (as shown in the image to the right). When the battery is fully charged the red LED will turn off. It is normal for the charging chip at the top of the board to become warm during charging.

The cellular modem is only powered by the battery positive pin, so a battery must be plugged in to use a cellular modem. The WiFi and LoRa modems are powered by the 3.3VS switched power rail. This is turned on in code. Additionally, to deliver power to WiFi or LoRa modems the appropriate dip switch for each modem must be turned on.

The 12VS and 3.3VS power rails are turned on in code while VCC (3.3V) and the 5V rails are always on. The maximum current provided by each rail is shown in the specifications below.



Panther Logger Specifications

For reference, basic specifications of the Panther Logger are provided below. We encourage our users to checkout the detailed explanation of power management on the Panther Logger in our learning center instructions here.

Dimensions:
4.7 x 3.1 inches (11.9 x 7.9 cm)

Maximum Supply Currents:

VCC	800 mAmps
3.3VS	800 mAmps
5V	2.4 Amps
12VS.	1.5 Amps

Typical Current Draw:

*Quiescent	20 - 30 mAmps
**Sleep	1 - 5 mAmp
Wifi (Transmitting)	230 mAmps
Wifi (Receiving)	68 mAmps
Wifi (Powered Down)	<0.5 uAmps
LoRa (Transmitting)	87 mAmps
LoRa (Quiescent)	3 uAmps
***Cell (Transmitting)	100 - 300 mAmps
Cell (Receiving)	20 - 60 mAmps
Cell (Powered Down)	<100 uAmps

*This quiescent current draw is when no sensors are drawing power and all modems have been powered down in code or physically turned off via the dip switch and cell modem detached.

**The SAMD21 processor on the Panther Logger has two sleep modes, standby and idle. We report here current draw in standby sleep mode. Idle sleep mode has not been tested, but it may be possible to sleep the Panther Logger at much lower current draw in idle mode.

***Cell modem current draw varies with models. Reported here for BG96 Quectel modem. See Nimberlink modem datasheets (here) for more exact specifications for other modems.

Note: the LoRa and Wifi modems can be turned off with the onboard dip switch if not used in a deployment. Similarly, the cell modem can be left unattached to the board if it is not to be used in a given deployment.

Battery Charging Specifications:

Input Voltage	5 - 10V
Max Charge Rate	1 Amp

Power up and connect the Panther Logger to Your Computer

Plug in the screw terminal blocks and the micro USB cable into the USB connector on the Panther Logger (see video on right). Plug the other end into your computer's USB port.

Use of a battery at this point is optional, but you will need one to use a cellular modem. See our battery recommendations page here. You will need to remove any connector from the battery wires, strip 1 to 2 centimeters of insulation off the wires and plug them into the battery screw terminal block on the Panther Logger with the positive wire on the right side (as indicated by plus sign on the board). **Do not cut both wires at the same time.** When a battery is plugged in, the power indicator LED will turn on if USB or solar power is applied.

Install Arduino IDE Software

Download and install the Arduino IDE here for Windows, macOS, or Linux. The current latest versions are Arduino 2.X, which came with some big changes from 1.X. Some of our tutorials use the older version so keep in mind things might look a bit different.

After installing, open the Arduino IDE and go to Arduino>Settings on macOS or File>Preferences on Windows and Linux. Follow the instructions at Adafruit.com here to setup the Arduino IDE for use with Feather M0 boards . You will need to add Adafruit boards support in the Arduino preferences and then install Adafruit and Arduino SAMD boards from the Arduino boards manager (see video on right). When finished, close and re-open the Arduino IDE. In the video we show how to arrange windows to upload a sketch to the Panther Logger and monitor serial output.

In the menu at the top of the Arduino IDE go to Tools>Board>Adafruit SAMD Boards and select Adafruit SAMD21.

Then go to Tools>Port and select the port where the Panther Logger is connected which will be labeled as "Adafruit Feather M0."

See here for other notes on connecting a board to the Arduino IDE.

Upload your first code to blink the onboard indicator LED

Go to our Github page for our Blink script here. Click the copy button in the upper right to copy the code then go to Arduino IDE and click file>new to make a new sketch. Highlight the default code that is there and erase it or just paste over with the code copied from Github. Save this new sketch as a new file name.

If you are not familiar with the basic structure of an Arduino C/C++ script read here.

As you can see from this sketch, we try to provide lots of notes in all of our sketches to indicate what is happening. These notes or comments follow two forward slashes "//" or are written in between slash-star and star-slash... like this:

```
/*
this is a note, it is not code
*/
```

The blink script will blink three of the four indicator LEDs in sequence and then blink them all together.

We are now ready to upload this code to the Panther Logger. Ensure that the board is selected under the Tools>port menu and then click the upload button in the top menu area which looks like an arrow pointing to the right.

NOTE: In the port menu the board will show up as "Adafruit Feather M0" because we are using that bootloader. If you do not see this then ensure that the board's power switch is on and that the USB cable is connected.

If upload fails then click the reset button on the Panther Logger board once and then double click it. The top red LED, LED1, turns on and begins to slowly fade or pulse in and out. The processor is now in bootloader mode and the code should upload at this point. The Arduino IDE will normally reset the microprocessor before sending code updates, but sometimes this fails to happen. By clicking and double clicking the reset button we are manually stopping any program that is running and then putting the processor into a state where it is ready to accept new code.

After uploading the script, the LEDs on the board will blink in sequence and then blink together. Open the serial monitor with the magnifying glass button on the top right of the Arduino IDE window. The script will print messages to the monitor indicating which LED is blinking.

If all goes well then you have successfully uploaded code to the Panther Logger and are on your way to using the full functionality of this device.

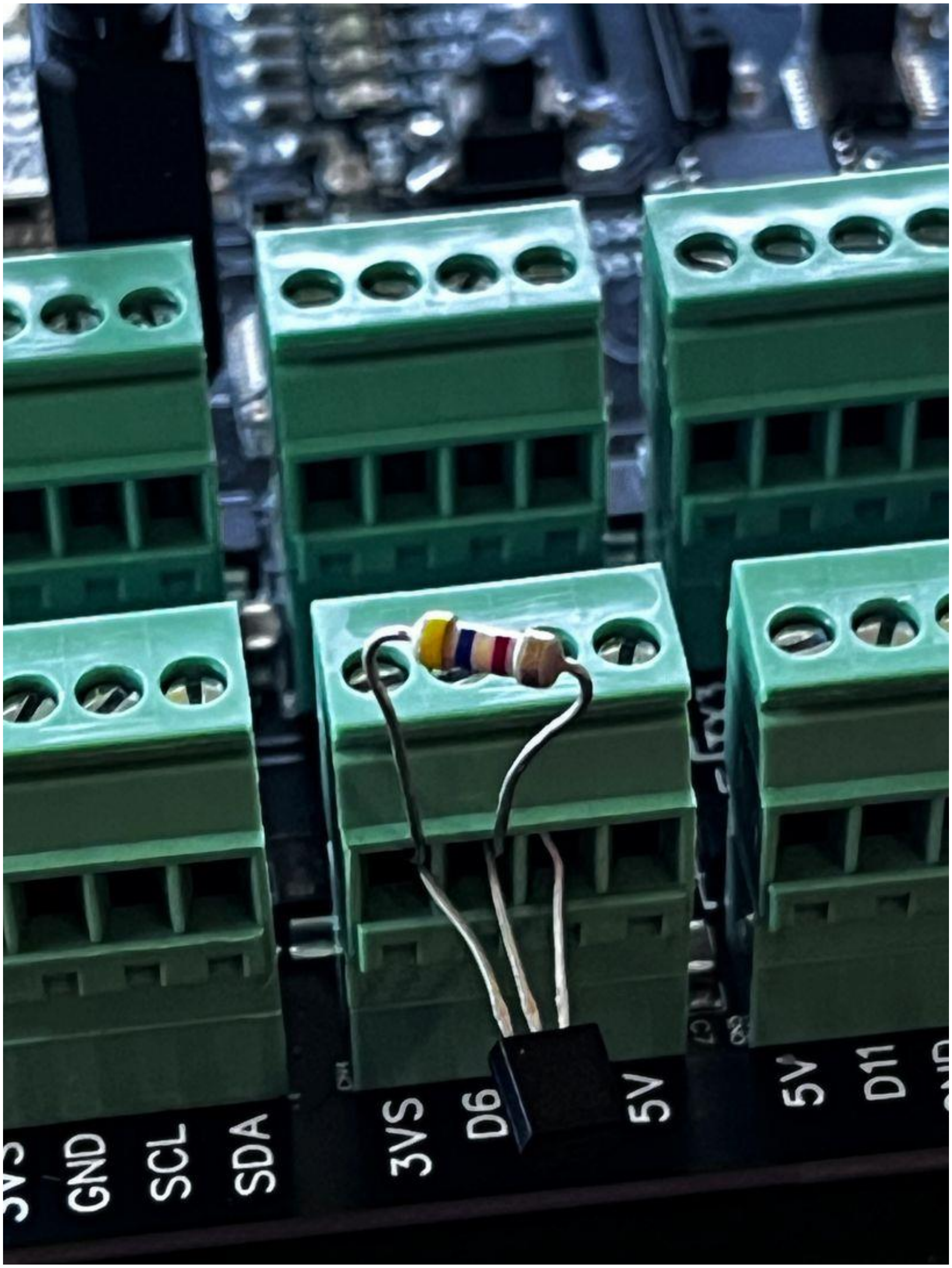
Attach the Temperature Sensor

To get started you will need the following:

1. Panther Logger board
2. DS18B20 temperature sensor (Digi Key Part# DS18B20+PAR-ND)

3. 4.7K resistor

Attach the DS18B20 sensor to block 6 screw terminal plug on the Panther Logger. To do this, hold the flat side of the DS18B20 sensor up and the sensor turned upside down (sensor wires up). See video to the right and picture below. In this position, the left most sensor wire is the power in (VIN) pin and aligns with the left most screw terminal port on the block 6 plug, which is one of the terminals for the switched 3.3V power rail (3VS). The two next to it are the other two we need for this sensor, the D6 digital pin for the sensor's signal wire and ground. Screw in the three wires from the sensor into these screw terminals for 3VS, D6 (signal), and ground. Next install a 4.7K resistor between 3VS pin and D6. When finished be sure the sensor wires are not bent and touching each other.



The DS18B20 temperature sensor correctly attached to the Panther Logger for this tutorial

The OneWire Protocol

The DS18B20 sensor communicates over the 1-wire protocol. Accordingly, in this tutorial we will use the OneWire library to read this sensor written by Paul Stoffregen. This library can be installed through the Arduino IDE. In the Arduino IDE menu go to Sketch>Include Library>Manage Libraries. In the search box that appears type OneWire. Find and click on the OneWire library from Paul Stoffregen. Select the latest version and click install. Close the dialog box.

The OneWire library comes with example scripts, but those will not work as is with the Panther Logger. Go to the DS18B20_Scan script in our Github library here. You can simply copy this code into a new file in Arduino. Click on the copy symbol in the upper right next to the "raw" button, then in the Arduino IDE go to file new. Erase everything in this new file and paste the code copied from Github. Save the new file with a name of your choice. Make sure the Panther Logger board is selected at Tools>Port and then upload the script to the Panther Logger. Open the serial monitor to view the data.

This script will scan for DS18B20 devices on the one wire bus by their address, read their temperature data and print the results to the serial monitor. If there is more than one DS18B20 device on the bus then it will go to the next address and report the data. If there is no more devices then it will report "no more addresses." Similarly, if no devices can be found then it will only print "no more addresses" to the serial monitor.

Touch the sensor and you should see the temperature as reported to the serial monitor begin to increase and then decrease when you remove your finger.

In this tutorial we have one device connected to the board, so the serial monitor should report data from one device and then print "no more addresses" and repeat about every few seconds. If no data is presented then check to be sure the sensor is on D6 and the pull up 4.7K resistor is in place between 3.3VS and D6. Be sure the sensor and resistor wires are firmly seated in the screw terminal and the screw terminal is tightened down.

Reading Multiple Sensors by Address

In most circumstances we want to read multiple DS18B20 sensors on the same bus by their known, pre - determined address in a specific order by distance from the processor. For example, in the past we have used the Panther Logger and multiple DS18B20 devices potted onto a cable to measure water temperature by depth in lakes onboard our Panther Buoy (here). It was necessary in this application, and indeed in most applications to know the position of the sensor in the array.

In the DS18B20_Scan script we can see the address of the sensor reported to the serial monitor window thanks to Paul Stoffregen's code. It is the 64-bit hexadecimal numbers printed after the word "ROM = " in the serial monitor output. Now is a good time to mention that its always a good idea to checkout the datasheet for the device you are using. The datasheet for the DS18B20 can be found here. This sensor was invented by Dallas Semiconductor, now Analog Devices.

With the OneWire library we can search for specifically a device with this address on the OneWire bus (which in this case is digital pin 6) and read its temperature data. If we have mutliple DS18B20 devices then we can store their addresses in an array and then have the processor iteratively search for each of these devices by their address and store the data in a float array.

Navigate back over to our Github library and pull up the DS18B20_Multiple script here. As previous, copy and paste this into a new Arduino script and save it with a new file name.

In this script we moved Paul Stoffregen's code in the loop of the last script into its own function called readTemps. We also define the number of sensors we have with:

#define NSENSORS 5

We are going to pretend we have 5 even though we only have one. We have also created a float object, Temp, that contains enough placeholders for as many temperature values as we have temperature sensors.

float Temp[NSENSORS];

Similarly, we create a byte array that is 8 columns of data corresponding to the 8 parts of the address and as many rows as we have sensors. Again, we only have one, but for demonstration purposes we are going to pretend we have five and just repeat the same address five times. Here's my address repeated five times:

```
byte Address[NSENSORS][8] = {
  {0x28, 0x12, 0x98, 0xAC, 0x0D, 0x00, 0x00, 0x09},
  {0x28, 0x12, 0x98, 0xAC, 0x0D, 0x00, 0x00, 0x09},
  {0x28, 0x12, 0x98, 0xAC, 0x0D, 0x00, 0x00, 0x09},
  {0x28, 0x12, 0x98, 0xAC, 0x0D, 0x00, 0x00, 0x09},
  {0x28, 0x12, 0x98, 0xAC, 0x0D, 0x00, 0x00, 0x09}
};
```

Now notice the "for loop" that was added in the readTemps function. Since we have more than one sensor we need to take each address one-by-one and read temperature data from the corresponding sensor, then that data is put into the same position in the float Temp array until the loop reaches the number of sensors as defined in NSENSORS. For more information on how the very useful for loop function works see the Arduino reference here.

The setup function is the same as previous. We start communications with the MCP and turn on the 3.3V switched rail using pin 4 on the MCP.

In the loop we now just need to run the readTemps function and print the data to the serial monitor. In order to make the code agnostic with respect to the number of sensors we again use a for loop to cycle through each position of data in the Temp float array to print the results from each sensor.

Run the previous DS18B20_Scan script and find the address of your sensor. Copy the address and enter it five times into the byte Address array in the DS18B20_Multiple sketch. See the video to the right to see how this is done. The addresses are entered as hexadecimal numbers. Make sure there is a comma after each of the first four addresses. Then upload this sketch to the Telleloger and open the serial monitor.

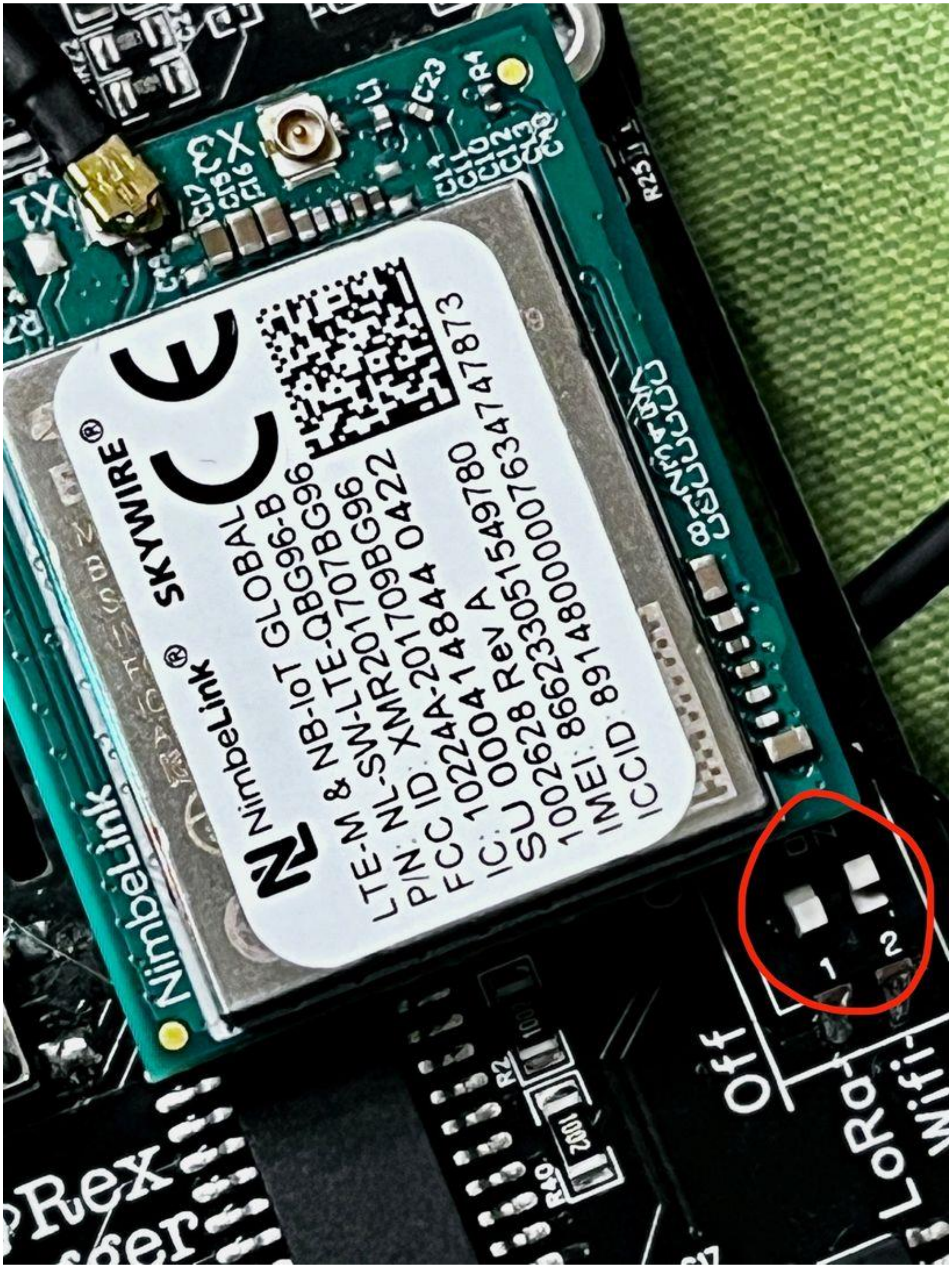
Notice that although we are reading the same sensor, we can have different temperatures reported to the serial monitor because the sensor is being read at different times. Natural variation and sensor noise cause this to happen.

If you actually have multiple sensors attached and want to read them in a specific order then you would just put their addresses in the correct order in the Address byte array. The temperature data is saved in a global float array so that it can be used later to send to an IoT database using one of the Panther Logger's communication methods.

Install WiFi101 Library and Power Up WiFi Module

To get started we need the WiFi101 library installed in Arduino IDE. Go to Sketch>Include Library>Manage Libraries. In the search box type wifi101. Click on WiFi101 and if not installed then install the latest version. If it is already installed check to be sure the latest version is installed which is 16.1. When finished close the dialog box and restart the Arduino IDE.

The Panther Logger board has a dip switch located on the right side of the board that allows one to keep the WiFi modem and LoRa modems off if not used. Since we will be using the WiFi modem in this tutorial, flip the right dip switch labeled "WiFi" up. This dip switch allows current from the 3.3V switched rail to flow to these modules. The 3.3V switched rail needs to be turned on in code for these modules to work (see below).



The dip switch is just below the LoRa and cell modem on the right side of the board

Scan for Nets

Let's look for available networks in your area. Go to our Github page here for the WiFiScanNets code. Copy and paste it into a new Arduino script. Save the script with a new file name of your choice.

In the setup function of this script the WiFi module needs to be turned on first. We set the dip switch on the Panther Logger board to use the WiFi module, but this only *enables* power to the module from the 3.3V switched rail. The 3.3V switched rail needs to be *switched* on in code in order to power the module. To do so, pin 4 on the MCP expander chip on the Panther Logger board needs to be set high. Near the beginning of the setup function you will see how this is programmed:

mcp.pinMode(4,OUTPUT); //Set mcp pin 4 as output
mcp.digitalWrite(4,HIGH); //Set mcp pin 4 high

In addition, we need to set the chip enable pin high on the WiFi module to enable it. We can do that with pin 15 on the MCP by setting it high with the following code:

mcp.pinMode(15, OUTPUT);
mcp.digitalWrite(15,HIGH);

However, the datasheet for the WINC1500 states that this enable pin should be *low* at power up (Table 2-1). As such, the sequence of events we program in the setup function to get the WiFi module up and running correctly should be the following:

- Set MCP pin 4 low (turns off 3.3V switched rail and turns off the WiFi module)

  mcp.digitalWrite(4,LOW);

- Set MCP pin 15 low (sets chip enable pin low on WiFi module)

  mcp.digitalWrite(15,LOW);

- Set MCP pin 4 high (turns on the 3.3V switched rail and turns on WiFi while chip enable pin is low as requested by the datasheet)

  mcp.digitalWrite(4,HIGH);

- Set pin 15 high, chip enable is high and the module is both powered and fully enabled

mcp.digitalWrite(4,HIGH);

So, with the above sequence of events we turn off the WiFi module, then set enable pin low and then power up the WiFi module with the enable pin low and then set enable pin high. This is all done in the WiFiScanNets setup function and in all other scripts using the WiFi module.

If the WiFi module were to be powered **on** at the same time as the processor was turned on, this sequence above might not be necessary, but the Panther Logger provides full control to turn on and off modules within code. Note that if you set the enable pin low after this power up sequence, this will put the module into Power-Down mode, which could be useful for saving battery power.

With the module powered up correctly, we then need to tell the WiFi101 library what pins are to be used for CS, IRQ, RST and Enable.

The CS pin is the chip select pin for SPI communications. This pin is used to turn on SPI communications between this module and the processor as opposed to other SPI devices like the SD card on the Panther Logger.

The IRQ pin is the Interrupt Request pin which allows the WiFi module to occasionally ask the processor to stop its activity momentarily so the WiFI module can finish communicating.

The RST is the reset pin, which if pulled low will reset the WiFi module.

The Chip Enable pin on the Panther Logger is pin 15 on the MCP we discussed above and we are managing this manually.

The Panther Logger does not use the default pin numbers that the WiFi101 library is expecting for these pins. So we run the function WiFi.setPins in setup like below to tell the library what pins we are using for these functions:

WiFi.setPins(9,7,3,-1);

The arguments for this function are (in order):

- ChipSelect (CS): this is digital pin **9** on the Panther Logger
- Interrupt Request (IRQ): this is digital pin **7** on the Panther Logger
- Reset (RST): this is digital pin **3** on the Panther Logger
- Chip Enable : we set this to **-1** so the WiFi library does not use it. We are already taking care of it with pin 15 on the MCP (as explained above)

There is code to turn on the WiFi power LED, the green LED above the WiFi module labeled "WiFi PWR." That code looks like this:

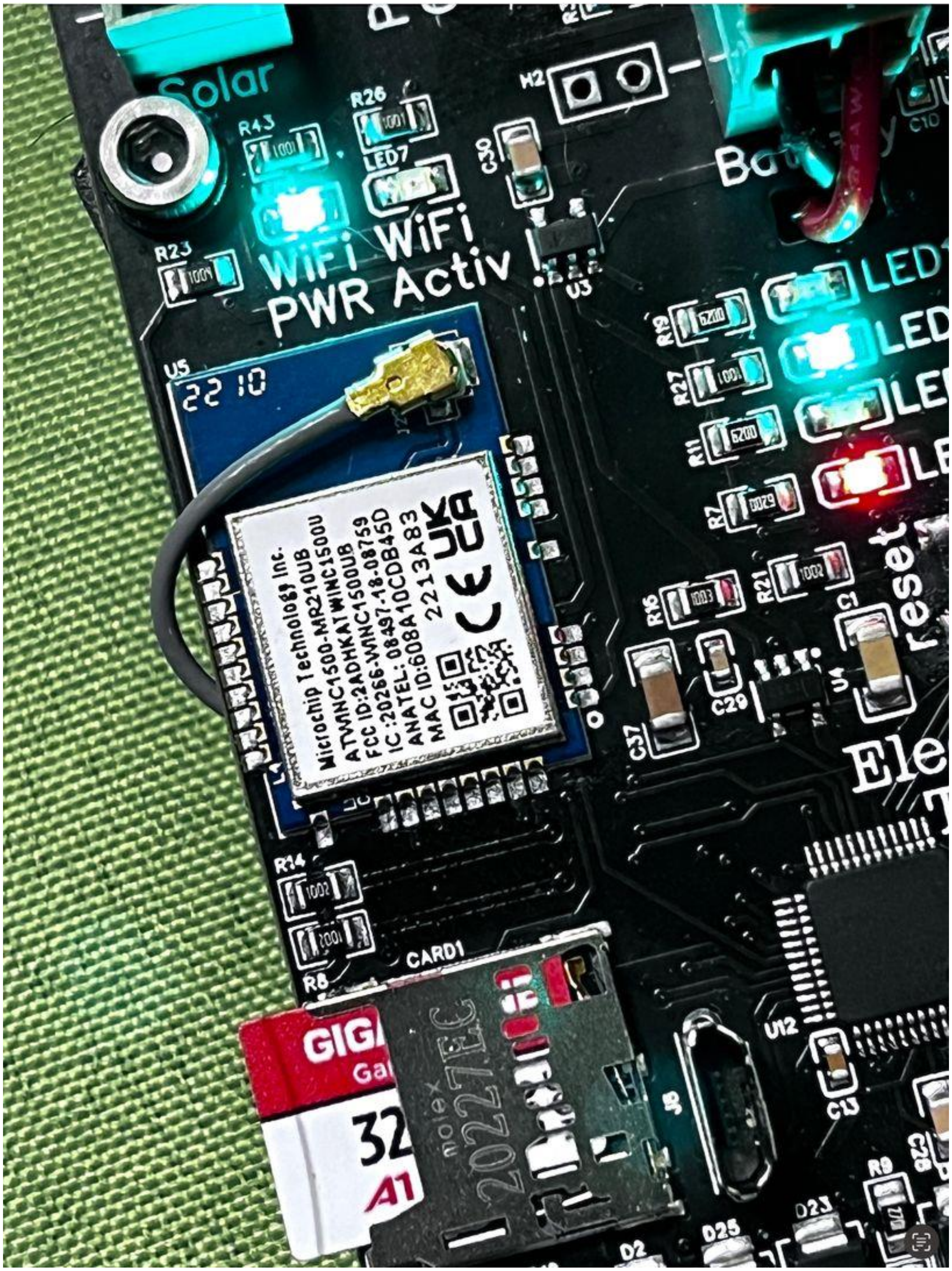
m2m_periph_gpio_set_dir(M2M_PERIPH_GPIO6,1);
delay(100);
m2m_periph_gpio_set_val(M2M_PERIPH_GPIO6,0);

The first line above sets the GPIO pin #6 on the WiFi module to output, then we give a short delay and the final line sets the GPIO pin #6 to ground, which turns on the green LED (see image below).

The other LED labeled "WiFi Activ" is the WiFi activity indicator LED and will flash on its own when sending or receiving data.

The rest of the code in this script is not specific to the Panther Logger and contains the basic commands needed to scan for available networks that is in the example scripts provided with the WiFi101 library. Go ahead and upload the code and then open the serial monitor. You will see a printout of available WiFi networks in your area. It might take a little while for networks to be detected.

To continue, you will need the WiFi credentials to login to one of these networks. Note that the WINC1500 WiFi module on the Panther Logger should be able to connect to enterprise networks, but we have not had experience with this. Others have made modifications to get this working (see here). For now, use a non-enterprise WiFi network on a router such as used on a home network. You can also connect to most smart phone hotspots.



The WiFi green power LED can be turned on in code

Connect to a WiFi Network and Send Data to ThingSpeak

We will show here how to send data from the Panther Logger to a few of the many IoT Databases that have free starter accounts or at least free trials. Most of them have an HTTP API service allowing us to send data using what is known as a REST (Representational State Transfer) protocol using GET or POST methods. These are hardware and language agnostic methods. We can send data to many different databases without relying on any platform specific libraries.

We will start out with using GET to send data to ThingSpeak and then show how to use POST methods to send data to Ubidots, Thinger, and ThingBoard (and possibly others as we update this tutorial.)

To get started, let's send data to ThingSpeak with GET. You will need to go to https://thingspeak.com/ and setup a free account. Once you have an account, login and click "New Channel" (see video to the right). Give the channel a simple name and description. The name only needs to be unique to the channels in your account. I'll call mine "loadstar" but you can choose whatever you want.

You have up to 8 fields in your channel that can receive sensor data from your device (if you need more then you can make an additional channel later to hold more fields). Let's say you have a weather station. Those 8 channels could be named:

Air Temperature
Humidity
Barometric Pressure
Windspeed
Wind Direction
Dew Point
Wind Chill
Battery Voltage

So for now go ahead and name the fields with those names to get started sending some mock weather data. The other options in the channel setup form are not necessary right now so just click save channel after naming the fields. Note that you do not need to remember the exact spelling of these fields for ThingSpeak, but you **do** need to remember the order.

After clicking save, click on the channel name and then click on the private view tab, which should be displayed by default. Here you can setup some graphs to display new data that is coming in. To do so click on add visualizations and select a field to be displayed in a graph. Do this for all fields. You can also setup a gauge for battery level if desired. These graphs and widgets will display the data once we start sending it.

Now click on the API_Keys tab. Copy the "write API key" or write it down as it will be needed later.

Go to our Github page for the "SendData_WiFi_ThingSpeak" code here. Copy and paste it into a new Arduino script, save as a new filename of your choice.
This script will connect to your local WiFi network, make up some weather data and send it to ThingSpeak.

You will need to enter your login credentials for your local WiFi network in the code starting with:

char ssid[] = "XXXXX";
char pass[] = "XXXXX";

The SSID is the name of your WiFi network and pass is the password. These should be in quotes and keep the semi colons at the end of each line.

Then where you see:

char ThingAPIKey[] = XXXXXXXXX

Replace the X's with your write api key you copied from ThingSpeak.

Save and then upload this code to the Panther Logger and open the serial monitor to watch it send data.

The script will send some mock data to ThingSpeak using the GET method and then read the server response. The response is a json formatted message containing the channel ID number, the time data was sent, the "entry_id" which corresponds to the number of data points sent and the data that was sent. Look at your ThingSpeak channel and you will see the channel ID number at the top of the page under the channel name. This number should match the channel ID number in the server response.

Send Data with POST to Ubidots using the WiFi Modem

We will now show how to send data to Ubidots.com using the WiFi modem on the Panther Logger and a POST method. Go to Ubidots.com and sign up for a new account with free trial or if you are a student sign up for a free student STEM account at https://ubidots.com/stem. After setting up a new account go to "Devices" and click the plus button in the upper right to setup a new device (see video on the right). Choose to setup a "blank" device. In the next dialog box that appears add a name for your device. You will enter this name later in the Arduino script to send data. Click the check mark at the bottom of the page. Now, click on the device just created in your list of devices. The device dashboard page appears. On the left you will see that there is a "token" which is hidden. You will need to copy and paste this into code in Arduino. You can copy it with the copy button next to the token (see video on right). You can add a description of your device if you want, but otherwise nothing else is needed right now.

Go to our Github page for the "SendData_WiFi_Ubidots" code here. Copy and paste into a new Arduino script and save as a new file name. Find the section of the code that starts with:

char ssid[] = "XXXXXXXXXXX";
char pass[] = "XXXXXXXXXXX";
char server[] = "industrial.api.ubidots.com";
char UbiToken[] = "xxxxxxxxxxxxx";
char UbiDeviceName[] = "xxxxxxxxx";

Here, you need to replace the X's with the name of the WiFi network to connect to and the password. You also need to replace the x's with the token from ubidots associated with your new device and the device name (exactly as entered in Ubidots).

Save the sketch and upload to the Panther Logger.

After uploading code, open the serial monitor. You will see the WiFi modem connect to Ubidots (see video on right). A successful transmission of data will return "HTTP/1.1 200 OK." Other useful information is also returned like the server date and time and json formatted data with HTTP code 201 for each variable sent indicating that new data was created.

When data is sent for the first time to a Ubidots device something kinda cool happens. Without us even setting up new variables, Ubidots automatically creates them based on the data that was sent. Keep in mind that if you change the spelling or misspell one of the variables (temp vs temperature) a new variable will be created.

In contrast to how we sent data to ThingSpeak by GET, here data was sent with POST. Using the POST method requires header information. In the Arduino script this begins with the line:

client.print("POST /api/v1.6/devices/");

The syntax and type of information sent in the header is critical to a successful data transmission and varies somewhat between APIs. Using an API tester like postman can be very useful for troubleshooting problems with this header information.

Send Data with POST to ThingsBoard using the WiFi Modem

We will now send data to two more IoT data hosting sites using a POST method. This will be very similar to Ubidots, but with all three cases the header information required is different. In addition, each service handles data and device setup a little differently.

Lets start with ThingsBoard. Go to the ThingsBoard login page (here) and click on "signup" at the bottom. Setup a new free trial account. Login to your account and in the left panel click on entities > devices. In the upper right side of the page click on the "+" button to add a new device. Give your device a name and label and then click on the "Add" button at the bottom. You will see your new device now in the list of devices. Click on this new device and a device detail window opens (see video on right). Click on the "Copy access token" button to copy the token. You will need to paste this token into the Arduino script to send data to ThingsBoard.

Go to our Github page for the SendData_WiFi_ThingsBoard script here. Copy and paste into a new Arduino script and save it with a new file name. Go to the section of code here:

```
char ssid[] = "XXXXXXXX";
char pass[] = "XXXXXXXX";
char server[] = "thingsboard.cloud";
char Token[] = "XXXXXXXX";
```

Replace the X's with the information required including the token you copied from ThingsBoard device details page.

Upload the script to the Panther Logger and open the serial monitor. Once data is successfully sent the monitor will report "HTTP/1.1 200." ThingsBoard doesn't return any information except the HTTP code and the time.

Go back to your ThingsBoard account and click on dashboard in the left panel then click on the "+" symbol off to the upper right. Give your dashboard a title and description and click "Add" at the bottom. Your new dashboard appears in the list of dashboards. Click on it and click on the "Add Widget" button. Choose a time series chart. For the data source choose your new device name. Under series delete the default "temperature" listed for the series set and click in the box to display the variables associated with your device. Since we have already sent temperature and battery voltage data those two variables should be listed. You can choose either one and just click add. A live time series chart now is displayed in the dashboard. Every time you see data sent successfully in the Arduino serial monitor you will see this chart update.

There many other chart widgets available to explore visualizing the data. For example, you could display battery voltage in a gauge.

Send Data with POST to Thinger using the WiFi Modem

Now let's send data to Thinger.io using the WiFi modem and POST method. Go to the Thinger.io login page here and click on the "Create an account" button. After your account is setup click on Devices on the left panel and then click "create a device" (see video to the right). For device type select HTTP device from the drop down list. Give the device a name in the device ID box then click Add Device at the bottom. A device dashboard will then appear. In the upper right click on the button with three horizontal lines next to the properties button. Select "callback" in the drop down list. Copy the Authorization key at the bottom of the page that is displayed.

Go to our Github page for the SendData_WiFi_Thinger code here. Copy it and paste it into a new Arduino script then save it as a new file. Find the part of the code here:

char ssid[] = "XXXXXXX";
char pass[] = "XXXXXXXX";

char ThingerToken[] = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"

Replace the X's with your WiFi network name and password as well as the authorization key (token) from Thinger.

Go back to Thinger and in the callback details page click on the overview tab. Look at the target URL. In the Arduino script, find the code:

char server[] = "backend.thinger.io";

Make sure the server given in the target url is the same as listed here. If not then change it to match.

Upload this code to the Panther Logger device and open the serial monitor. The Thinger server returns "HTTP/1.1 200 OK" indicating successful data transfer.
Let's go back to our Thinger account and setup some data handling and visualizations (See video to the right). On the left panel of the Thinger account page click on "Data Buckets" and then click on create a data bucket. Give the data bucket a name and ID like "PantherBucket" for both. Set the data source to be from device resource and then select the device we set up previously. For resource name just type anything for now. We will put Panther Logger. Select "Stream by Device" for refresh mode. Click on the "Add Bucket" button at the bottom to create the bucket.

Click on devices and then on the device name we set up above. Go to the call back details page and click the "Write Bucket" check box and then in the pull down list select the data bucket we just created. Click save button at the bottom of the page.

Go back to Data Buckets and click on the data bucket we just made. The bucket will now start filling with rows of the temperature and battery data we are sending from our device.

Click on Dashboards on the left panel and then click on create a dashboard. Add a dashboard ID, name and description and then click on the "Add Dashboard" button. You will see a page now that says "empty dashboard." Ok, lets make it not empty. Click on the write button in the upper right corner and then click on "Add Widget." Name the widget temperature and then check the "link to" box and select the dashboard name we just created. In the "type" drop down box select a time series chart. Select as the data source the data bucket we just created and temperature as the variable. Choose "configurable" as the time frame option. Then add widget with the save button. You will now see temperature time series plot and it will be updated with new data coming in from our device. You can also setup other types of widgets like a battery gauge.

Disabling the Modem and Sleeping the Board to Save Power

The WiFi modem on the Panther Logger will draw a relatively large amount of current (~150 mAmps) constantly if it remains connected to the network. Connected to a wall outlet that's not so bad, but in a mobile battery operated application it would be best to disable the modem and possibly even sleep the board to reduce this power consumption.

To try this out go to our github repository and copy/paste the "SendData_WiFI_ThingSpeak_PowerSave1" script into a new Arduino file and save it with a name of your choice. This is a modification of our "SendData_WiFI_ThingSpeak" script. As previously, replace the SSID name and password for your network and the API key for ThingSpeak. Note that this is just a modifcation of the "SendData_WiFI_ThingSpeak" script above. It could be modified to send data to any online database.

The specific modifications in this script are to 1) disable the WiFi modem after data is sent, 2) turn off the 3.3VS power rail removing power to the modem, and 3) implement a function to sleep the SAMD21 microprocessor. To do these tasks, at the end of the loop we send the command, WiFi.end() to disconnect the modem from the WiFi network, then set pin 15 on the Panther Logger's mcp GPIO expander low to disable the modem and then the 3.3VS rail is turned off by dropping pin 4 on the mcp GPIO expander low. Then the Panther Logger's processor is put to sleep for 10 x 6 second sleep intervals or for 1 minute. The board will wake up after 1 minute and enter the start of the loop again. At the beginning of the loop we now need to turn on the 3.3VS rail by setting pin 4 on the mcp GPIO expander high. We also need to re-enable the WiFi modem by setting pin 15 on the mcp expander high and re-initiate a connection to the network with a WiFi.begin statement.

With the modem disabled and no sensors attached to the Panther Logger the board will pull about 45 mAmps. If the sleep function is also used then the board will pull about 10 to 15 mAmps. This assumes a cellular modem is not plugged into the board, no sensors are being powered and the LoRa modem is off. With the sleep functions enabled the Panther Logger's processor will likely stop reporting messages to the serial monitor. To know that it is working you should check for updated data at ThingSpeak.

How to add SSL certificates to the WiFi modem

Sending data to HTTPS addresses might require that the WiFi modem contains the correct SSL certificate, which enables an encrypted connection to the server's domain. To do upload a certificate we will use the older Arduino 1.x versions of the IDE. See here to get it. Connect the Panther Logger to your computer using a USB cable and open the Arduino IDE. Head over to our Github repo and download or copy/paste the WiFiFirmwareUpdate script into a new script and save it. Upload the script to the board and be sure the dip switch for the WiFi modem is on. Do not open the serial monitor. Instead click on the tools menu and then on Firmware updater. A new window opens up. Select the port connected to the Panther Logger. You can test the connection. In the bottom window click add a domain and type in the domain name of the HTTPS server that you want to connect to via WiFi. Click on "Upload certificates to the modem."

To send data securely via SSL connections, change the client class to WiFiSSLClient instead of WiFiClient and use the function connectSSL instead of connect. In the connect function instead of port 80 use port 443.

Overview

You will need **four** things to send data over cellular communications with the Panther Logger:

1. Cell modem
2. SIM card
3. A lithium ion polymer 3.3V battery
4. Antenna connected to the cell modem

1. Cell Modem
A variety of modems from Nimbelink fit the Panther Logger's 2 x 10 pin header format and pin outs. For this tutorial we will use Nimbelink's Quectel BG96 modem (specifically NL-SW-LTE-QBG96-B or revision C) which can be purchased from Digi Key (here) and other distributors. See the Nimbelink site for this modem here. Please note that model revision "B" of this modem or later should be used as this comes with the correct firmware. The BG96 modem is a narrow band LTE-M and NB-IoT modem that operates at lower power. It is used in many different low power IoT applications. Nimbelink also provides a variety of other interchangeable modems that work with the Panther Logger including wide band modems (see here).

2. SIM Card
For cell service we will use a Hologram.io SIM card that works globally or just within the United States. See Hologram's website for more details.

Do not leave the board running code that sends data needlessly using up your purchased data or running up a bill with your service provider! If you load code to the Panther Logger that sends data using the cellular and SIM card then be sure to unplug the cell modem when not using.

So keep in mind how often you send data and how much you send with each transmission. There are ways to decrease data usage and costs significantly. First, Hologram (and probably other service providers) allows you to limit the amount of data for any SIM. For example, any SIM that supports machine to machine transmissions and there are many competing companies.

Hologram provides roaming service mainly on ATT, US Cellular and T-Mobile. Soracom is another option for SIM cards. Whatever the SIM card you choose, it needs to be activated according to the instructions provided by the service provider. Go to the providers website, setup an account and enter the SIM card number to activate it (for Hologram see here). The dashboard provided by Hologram and Soracom provide useful monitoring information about your device's connectivity.

The SIM card should be inserted as a microSIM (see here) into the back of the Nimbelink cell modem and then the modem should be inserted into the 2x10 pin header on the Panther Logger board. Please insert the modem into the headers with the power switch on the Panther Logger **off** and **no battery** plugged in. ***The Nimbelink modem should be orientated on the 2x10 pin headers such that the antenna connections on the top of the cell modem are closest to the***

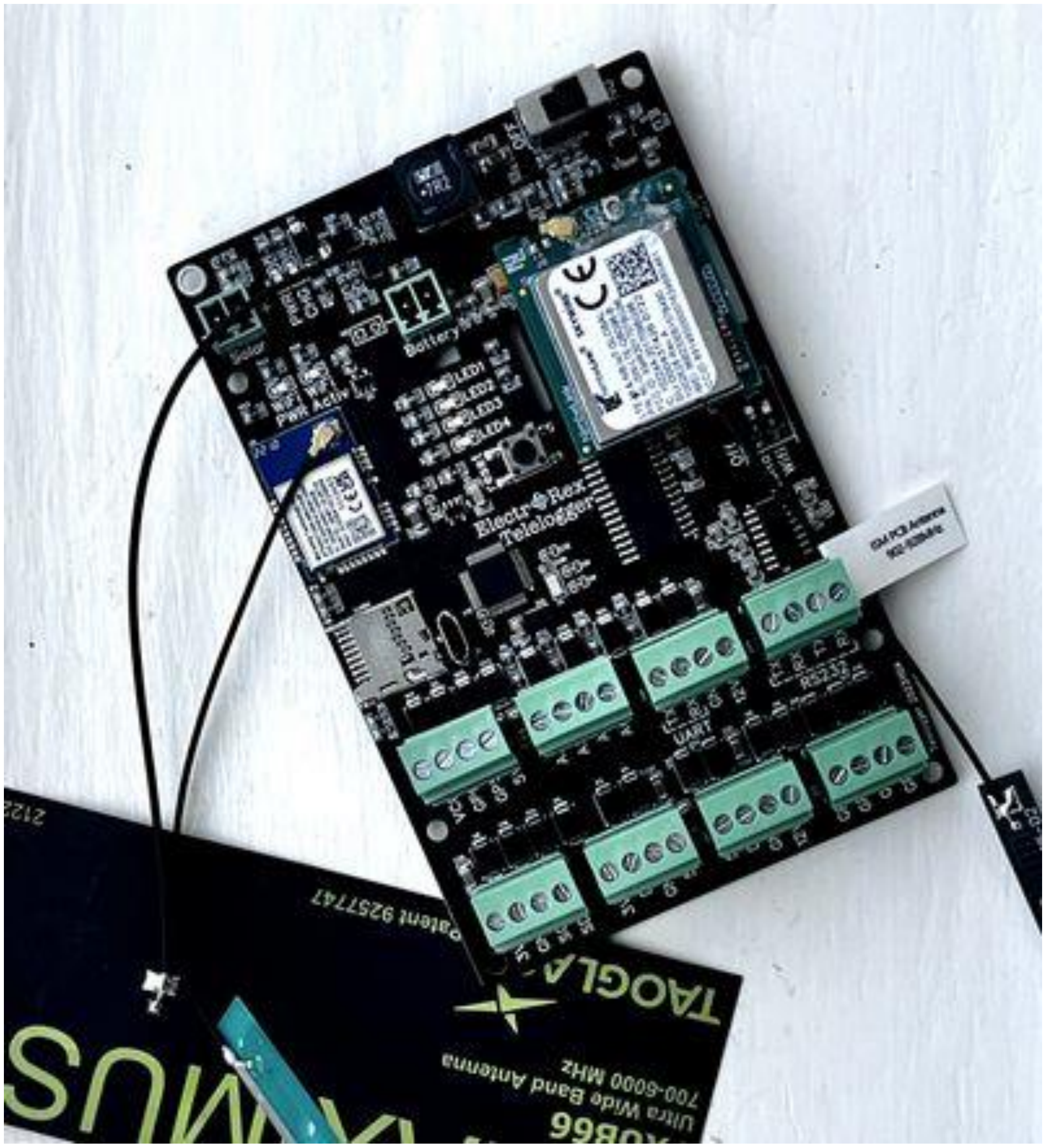
on/off switch at the top of the Panther Logger board (see photo to the right). Powering the Panther Logger with the cell modem inserted into the header pins in the opposite direction will damage the modem, short the battery, and could possibly damage the Panther Logger.

3. Battery

You will need a battery plugged into the Panther Logger board in order to use a cell modem as the cell modem is only powered by the battery, not by any of the power rails on the board. This is because current draw by cellular modems often needs to spike, briefly, to a high level during data transmissions. In order to provide this to the cell modem, its power pin is directly connected to the positive battery terminal with wide traces over a short distance and with a large amount of capacitance. See our battery recommendations here and usage notes.

4. Antenna

We provide specific recommendations on which antennas to use with Panther Logger modules. A cellular antenna is needed to run this tutorial. It should be plugged into the u.fl connector on the Nimbelink modem. **The antenna should be greater than 20 cm away from any human body during use to comply with FCC rules.**



Sending Data With Cellular: Creating the Arduino Sketch

With the cell modem installed and battery connected, go to our Github page for the SendData_Cellular_GET code here. You will see multiple files. That's because this Arduino sketch has multiple tabs to organize the code better. As such it has multiple files that correspond to those tabs.

In the Arduino software, make a new sketch, then copy/paste the code from the SendData_Cellular_GET.ino file in our Github directory linked above into this new sketch and then "save as" with the same name. (see video on right)

Now, click on the down arrow in the upper right part of your new sketch. Choose new tab and give it the name of one of the other files in this Github folder (e.g. _01_Sensor_Functions).

There are four files so you will need a total of three tabs in addition to the main first tab so go ahead and create all of the tabs needed. You can give the tabs any name you like, HOWEVER, the beginning of each tab name is critical. It gives the order and this order is important as it is the order that the code should be run. We used numbers so they are automatically ordered correctly. You will need to copy/paste the contents of each Github file into each respective tab and then save the entire script. See the video to the right as an example.

Setting Up to Send Data & Introduction to the Cellular Code

We will send data first to ThingSpeak using the GET method. Your ThingSpeak account and channels should be setup just as we did in tutorials for WiFi communications in the Panther Logger 3 tutorial. See that tutorial to setup your account and the channels we will use.

We need to change some things in the SendData_Cellular_GET code to send data to a specific ThingSpeak channel just like we did for WiFi. Find this section in the code near the beginning on the first tab:

char Token[] = "XXXXXXXXXXXXXXXXXXX";

Replace the X's with your api_key from ThingSpeak.

Then find this code:

char Endpoint[] = "https://api.thingspeak.com/update.json?api_key";

This shows the ThingSpeak endpoint for making a GET call. After the "?" is the name of the credential we need to include. ThingSpeak calls this the "api_key" whereas other sites might call it just "key" or "token." See the API documentation for a REST HTTP GET method for the site you are communicating with. ThingSpeak's API documentation is here. Since we are sending data to ThingSpeak with this tutorial nothing needs to be changed, but if sending to a different database then the name of this "Token" might need to be changed based on the API documentation.

The code is going to build up this url with the api_key and the data we want to send. It will use the sprintf function to do this. See this function below:

```
sprintf(Payload,
"%s=%s&field1=%.2f&field2=%.2f&field3=%.2f&field4=%.2f&field5=%d&field6=%.2f
&field7=%.2f&field8=%.2f",
        Endpoint,
        Token,
        AirTemp,
        Humidity,
        BaroPressure,
        WindSpeed,
        WindDirection,
        DewPoint,
        WindChill,
        Batv
       );
```

This function takes three arguments. The first is the character string, in this case we called it "Payload." This is the character string we will fill with the url. The second argument is the url text or characters. Any text with a percent in front of it refers to an injected variable from the third argument and in the order listed with a comma in between. So the first text with a percent is %s which means fill this with a string variable. The first variable is "Endpoint" define in the sketch as a C string or char variable and it is the first part of the url we need to send (i.e. https://api.thingspeak.com/update.json?) . Others are floats and ints or integers containing the data we want to send so we use %f or %d, respectively for those variables.

sprintf is a very important and useful function when the concatenate function strcat cannot be used because you need to inject various number and text formats into a string variable.

Here is an example of a GET url generated by this code. We replaced our api_key with X's.

https://api.thingspeak.com/update.json?api_key=XXXXXXXXXXXXXX&field1=16.00&field2=60.00&field3=995.00&field4=39.00&field5=158&field6=7.00&field7=21.00&field8=4.13

ThingSpeak expects data in "fields" so data is specified by field numbers instead of variable names. Sensor data is inserted into this url with the sprintf function in the GET Functions tab. It is worth some time studying how this sprintf function works as it is very useful for getting sensor data into a string. Most modems expect or at least can use strings to send data. They usually do not or cannot expect float or int data.

We are going to send random numbers for weather data and actual measured battery voltage.

Sensors are measured on the Sensor Functions tab. We have a function to measure battery voltage that also makes use of a function to read millivolts on any analog pin (battery voltage is measured on analog pin 4 on the Panther Logger).

We have a function called readSensors() that puts random numbers into mock weather sensor variables and implements the battery voltage measurement. In other tutorials in our Learning Center we show how to measure various kinds of sensors using the Panther Logger. We are not showing that here in this tutorial so as not to complicate the task of sending data to the internet. In an actual application one would add additional functions here to read actual sensors on this tab, like weather and water quality sensors, soil moisture etc and then that data would get sent using the cellular modem provided that the sprintf function is updated to include them and that the server database is setup to receive them.

On the GET Functions tab there is a function to make the url (i.e. the payload) and a function to send the payload using the HTTP GET method called "GetData." The GetData command sends multiple AT commands to the modem to setup communications and finally sends the data. Note that many of these AT commands are universal to all modems whereas some are specific to this Quectel modem. We put comments in the Arduino sketch indicating what each command is doing.

Finally, the Cellular Functions tab contains functions to interact with the cell modem. This includes a function "sendAT" to send AT commands to the cell modem and read its response. It takes three arguments:

sendAT("AT+Command", "Response1", "Response2", Wait Time);

1. AT+Command = The AT command to send to the modem
2. Response1 = One of the expected responses from the modem, usually positive
3. Response2 = A second response expected from the modem, usually expected error code
4. Wait Time = Amount of time to wait for either response above

It is very important that code to send AT commands waits for an expected response from the modem whether it is a positive response or error before your program continues and sends another AT command. Sending multiple AT commands to the modem without waiting for each response from each AT command can cause the modem to freeze. This is why the code has multiple delays between sending AT commands along with this "sendAT" function to wait for expected responses. The AT command documentation for the Quectel BG96 modem provides the expected responses for each AT command. Generally a one second delay between sending AT commands is recommended.

This Cellular Functions tab in the sketch also contains functions to read network registration status, signal strength and quality. In other tutorials we will show how to read the date and time from the modem, get geolocation information and basic modem information like modem model number, SIM number, and network tower ID to which the modem is connected as well as the service provider.

Send Data With Cellular Using HTTP GET to ThingSpeak

Upload the SendData_Cellular_GET sketch to the Panther Logger and then open the serial monitor. You will see the processor sending AT commands to the modem in order to setup the modem to first get a signal and register with the network and then to setup the URL to send data with "GET." After the data is sent you should get:

+QHTTPGET
Sending:
Received: : 0,200

The "200" indicates a successful transmission. After that you should also get the header response information because the code sent the response header command:

AT+QHTTPCFG="responseheader",1"

Where the 1 indicates we wish to receive a response header (0 if we do not).

And we send the read AT command:

AT+QHTTPREAD=60

That command indicates that we want to read the server response and we give it 60 seconds to do so.

Here is a typical response from ThingSpeak:

```
CONNECT
HTTP/1.1 200 OK
Date: Wed, 06 Dec 2023 04:06:24 GMT
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Status: 200 OK
Cache-Control: max-age=0, private, must-revalidate
Access-Control-Allow-Origin: *
Access-Control-Max-Age: 1800
X-Request-Id: 6b0566e5-9a04-4aca-a25a-b601810d05bf
Access-Control-Allow-Headers: origin, content-type, X-Requested-With
Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS, DELETE, PATCH
ETag: W/"313f8984cbbc2c3c315c8d0790cb6b85"
X-Frame-Options: SAMEORIGIN

{"channel_id":2362505,"created_at":"2023-12-
06T04:06:24Z","entry_id":2251,"field1":"6.00","field2":"49.00","field3":"1192.00","fiel
d4":"38.00","field5":"1"e610,e710"e:."au:llgu:llann"a"u
OK
```

If you do not get an HTTP/200 response or errors then check the following:

1. Is antenna connected?
2. Is battery connected and the board turned on
3. Is the token or api_key entered correctly?

Be sure you copied the code correctly and if you changed anything from our Github code then go back to the original first. An example of serial monitor output is included in our Github repository here.

Sending Data Using HTTP POST with Cellular

Now we will send data using HTTP POST. We will also send the data over a secure sockets layer connection (HTTPS). We are going to send data to Ubidots so make sure you have setup an account as described in Panther Logger 3

Go to our Github respository page for the SendData_Cellular_POST code here. As with the GET code above there are multiple files, each one corresponding to a tab in the Arduino sketch. Copy the contents of each file and put into tabs in a new Arduino sketch as done for the GET code. Name the tabs according to the file names. The main (first) tab should be the code in the SendData_Cellular_POST.ino file.

Go to this code in the main tab:

```
char url[] = "https://industrial.api.ubidots.com";
char Host[] = "industrial.api.ubidots.com:443";
char Token[] = "X-Auth-Token:
XXXXXXXXXXXXXXXXXXXX";
char Destination[] = "/api/v1.6/devices/XXXXXXXXXX";
```

The **char url** holds the endpoint address for Ubidots API and this is obtained from the Ubidots API documentation (here). The Host char will be inserted into the header information sent with the data and it should just be the domain name with a port of 443 since we are sending data over HTTPS. If sending over HTTP then the default is port 80. Again, check the API documentation for port settings. The token holds the API key or token or in this case "X-Auth-Token." The name of the token and the token itself will need to be changed according to the API documentation and of course your device's given token. The token from Ubidots is on your device information page. Copy the token from Ubidots and replace the X's in the Token char. In the Destination char ubidots used the device name that you gave your device when creating a new device at ubidots. In our case we called our device "Panther Logger" Replace the X's in the Destination char with your device name at Ubidots (see video on the right).

That's all that needs to be edited in the code. Go ahead and upload to your board and open the serial monitor.

As with the GET code above you will see AT commands sent to the modem and the modem responses. These AT commands register the modem on the network, get it setup for a POST request and then sends the data with a post command. Here is an example of a successful post request to Ubidots.

About the Cellular POST Code

A critical part of the POST command code is the syntax of the header information. In the SendData_Cellular_POST code above see this part of the code on the _02_Post_Functions tab:

```
void MakePayload(){
  String DataStringS = DataString;
  sprintf(ContentLength,"Content-Length: %d",DataStringS.length());
  strcpy(Payload,"POST "); strcat(Payload, Destination); strcat(Payload," HTTP/1.1");
strcat(Payload,"\r\n");
  strcat(Payload, "Host: "); strcat(Payload, Host); strcat(Payload, "\r\n");
  strcat(Payload, "Accept: */*"); strcat(Payload, "\r\n");
  strcat(Payload, ContentLength); strcat(Payload, "\r\n");
  strcat(Payload, "Content-Type: "); strcat(Payload, ContentType); strcat(Payload,
"\r\n");
  strcat(Payload, "User-Agent: "); strcat(Payload, UserAgent); strcat(Payload, "\r\n");
  strcat(Payload, "Cache-Control: no-cache"); strcat(Payload, "\r\n");
  strcat(Payload, "Connection: keep-alive"); strcat(Payload, "\r\n");
  strcat(Payload, Token); strcat(Payload, "\r\n");
  strcat(Payload, "\r\n");
  strcat(Payload, DataString); strcat(Payload, "\r\n");
}
```

This is a function to make the payload to send and the first part of it is creating the request header. The request header looks like this:

```
POST /api/v1.6/devices/panther2 HTTP/1.1
Host: industrial.api.ubidots.com:443
Accept: */*
Content-Length: 42
Content-Type: application/json
User-Agent: panther2/1.0
Cache-Control: no-cache
Connection: keep-alive
X-Auth-Token: XXXXXXXXXXXXXXXXX

{"temperature": 16.00,"battery": 4.12}
```

All of the spelling (syntax), characters, and even spaces are important in this header information. The same was true in tutorial #3 when sending data via POST using WiFi. For example, the header must begin with "POST", then a space, then the destination address, then a space and then "HTTP/1.1" Each line of the header must end with \r\n indicating new line and line feed. A blank line must be in between the last line of the header and start of the data string to send. If these are not entered correctly then will result in an error.

There is a function on this tab to create the data string to send in json format. Right now this just sends actual battery voltage and fake temperature data. There are two lines of code to measure the length of this data string, which is inserted into the header for "Content-Length."

There is a function on this tab called "getPayloadCommand" to calculate the total length of the payload including the header information plus the data string and then make the AT command for a POST.

The length of the messages sent must be exact or the transmission will fail.

Talking to the Modem with a Terminal

If you want to practice sending AT commands to the modem then you can upload our Cellular_Terminal sketch here. This will create a terminal from the serial monitor to talk to the modem. Upload the sketch and then open the serial window. The modem will be reset and then wait for "APP RDY."
At the bottom of the window set the line ending to "Both NL & CR." Then in the "send" box at the top of the serial window enter the simple AT command, AT, and then hit enter. The modem will respond with "OK."

See video on the right for a demonstration and be sure to turn on captions.

You can now send any AT command here and see how the modem responds. This can be a good way to try out AT commands, see how the modem works, and troubleshoot problems with sending data or getting data from the modem.

The AT command manual for the BG96 modem is here. Below are some useful AT commands. Try entering them into the serial monitor window and see the modem response.

The AT command below is in bold and the response in italics.
//Get modem model and revision number
ATI

Quectel
BG96
Revision: BG96MAR02A07M1G

//Get SIM card number. I replaced some numbers with X's
AT+GSN

8662330515XXXX

//Get the modem temperature. Returns temperature of three different parts of the modem (in Celsius)
AT+QTEMP

+QTEMP: 40,35,36

//Get the current date and time. Note this requires that the modem is registered on the network
AT+CCLK?

+CCLK: "23/12/11,08:13:15-24"

//Get signal strength and quality as well as current network.
//As you will see right now we are on a Cat-M1 network
//There are four numbers. The second and last numbers are the RSRP and RSRQ

//RSRP is strength and RSRQ is quality. See here for nice description.

AT+QCSQ

+QCSQ: "CAT-M1",-95,-125,108,-14

//Get network registration status
//We can send three commands at once
//This gets registration status for the three network types GPRS, ESD and CSD
//Look at the AT command manual for explanation of modem responses
//A response of 5 means registered and roaming
AT+CREG?;+CGREG?;+CEREG?

+CREG: 2,5,"4301","4A00F09",8

+CGREG: 2,4

+CEREG: 2,5,"4301","4A00F09",8

//Set to choose network automatically
AT+COPS=0

//Search for available networks
//This can take up to 30 minutes
//In my experience usually takes about 2 minutes
//This can be useful to do to get the modem registered initially
//Set AT+COPS=0 first and then:
AT+COPS=?

+COPS: (1,"311 589","311 589","311589",8),(1,"Verizon","Verizon","311480",8),(1,"311 588","311 588","311588",8),(1,"U.S.Cellular","USCC","311580",8),(2,"AT&T","AT&T","310410",8),(1,"313 100","313 100","313100",8),(3,"T-Mobile","T-Mobile","310260",8),(1,"311 490","311 490","311490",8),,(0,1,2,3,4),(0,1,2)

Get GPS Information
First we need to setup how we want to receive GPS information and then turn it on.
//Turn on the DC power to the GNSS antenna, and save it in this state across power off

AT+QCFG="gpio",1,64,1,0,0,1

OK

AT+QCFG="gpio",3,64,1,1

OK

//Query status of the antenna pin
AT+QCFG="gpio",2,64

+QCFG: "gpio",1

//Turn on GPS
AT+QGPS=1

OK

//Turn on NMEA output
AT+QGPSCFG="nmeasrc",1

OK

//Request NMEA sentences containing GNSS information. See here.
//The NMEA sentences "GGA" and "RMC" provides most information available.
AT+QGPSGNMEA="GGA"

+QGPSGNMEA: $GPGGA,,,,,,0,,,,,,,,*66

//The above response indicates we do not have GPS fix. Move device to clear view of the sky

We can enter the AT commands in the SendData_Cellular_GET sketch manually to send data to ThingSpeak. Copy and paste the AT commands below into the serial monitor line by line hitting enter to send each one. Replace the X's in the URL with your API key from ThingSpeak. This assumes you are using a Hologram.io SIM card. If not then change the APN in the AT+QICSGP command to your provider's APN. In the video to the right I am using a SIM card from Soracom, so the APN is set to "soracom.io" instead of "hologram"

```
AT+CFUN=1,1
ATE
AT+QCFG="nwscanseq"
AT+COPS=0
AT+QICSGP=1,1,"hologram"
AT+CREG=2;+CGREG=2;+CEREG=2
AT+QHTTPCFG="contextid",1
AT+QHTTPCFG="responseheader",1
AT+QHTTPCFG="requestheader",0
AT+QHTTPCFG="sslctxid",1
AT+QSSLCFG="seclevel",1,0
AT+QIACT?
AT+QIACT=1
AT+QIACT?
```

//After entering the AT command below the modem will respond with "CONNECT" and then you can enter the URL.
//The first number is the length of the URL (number of characters), which you will need to determine manually

AT+QHTTPURL=165,80

https://api.thingspeak.com/update.json?api_key=XXXXXXXXXX&field1=23.00&field2=43.00&field3=962.00&field4=29.00&field5=300&field6=18.00&field7=22.00&field8=4.15

AT+QHTTPGET=80

//You will receive "ok" from the modem, but now wait for "+QHTTPGET: 0,200" (successful transmission) or at least some other response before continuing

//Issue read command to get response from ThingSpeak server
AT+QHTTPREAD=80

How to use the Forbidden Network List

The forbidden network list or FPLMN (Forbidden Public Land Mobile Network) is a list of networks that the modem will maintain and not attempt to make a connection. In some cases it might be advantageous to add a carrier to this list. For example, we recently learned that one carrier in our country did not have great support for CatM/NB-IoT devices. Our modem (Quectel BG96) would connect to this carrier's network and send a few data packets and then completely stop. To fix the problem, we added this carrier to the forbidden network list in our area, the modem connected to a different carrier and was able to send data continuously.

To add a carrier to the FPLMN you need to know its MCC and MNC codes in your area. To find this, upload the Cellular_PassThrough sketch and send the following AT commands:

ATE
AT+COPS = 0
AT+COPS = ?

Wait up to 30 minutes (usually about 5 minutes). You will get a comma separated list of carriers in your area which will include their Home Network Identity (HNI) codes. This is simply a concatenation of their MCC and MNC numbers where the first three numbers of the HNI are the MCC and the last three numbers of the HNI are the MNC. You will need to use the MCC and MNC codes to create the PLMN code used to add the carrier to the forbidden network.

The PLMN code for a carrier is created from the MCC and MNC as follows:
```
Byte 1: MCC Digit 2 + MCC Digit 1
Byte 2: MNC Digit 3 + MCC Digit 3
Byte 3: MNC Digit 2 + MNC Digit 1
```

For example:

```
HNI ID = 310410 (home network ID for AT&T in the US)
MCC = 310
MNC = 410
PLMN ID = 130014
Byte 1 = 13
MCC Digit 2 = 1
MCC Digit 1 = 3
Byte 2 = 00
MNC Digit 3 = 0
MCC Digit 3 = 0
Byte 3 = 14
MNC Digit 2 = 1
MNC Digit 1 = 4
```

Once the PLMN has been created we need to issue it in the AT+CRSM command as follows:

AT+CRSM=214,28539,0,0,12,"XXXXXXFFFFFFFFFFFFFFFFFF"

Replace the X's in the above with the PLMN number of the carrier you want to add to the FPLMN list. Issue this AT command in the setup sequence for the Panther Logger cell modem.

Overview

The LoRa modem on the Panther Logger is the RAK Wireless 11720 (datasheet is here). It is located on the upper right side of the board between the headers for the cellular modem. Sending data with a LoRa modem requires that a gateway is nearby, much like WiFi requires a router nearby. However, unlike WiFi, LoRa can send data much farther, several miles (LoRa = long range radio communication) and with lower power usage, while WiFi is limited to several hundred feet and will use more power. The tradeoff is that not as much data can be sent with LoRa and it must be sent in smaller packets. Considering that many monitoring applications only need to send a couple to a couple dozen data points every sampling interval, LoRa can then be a perfect solution.

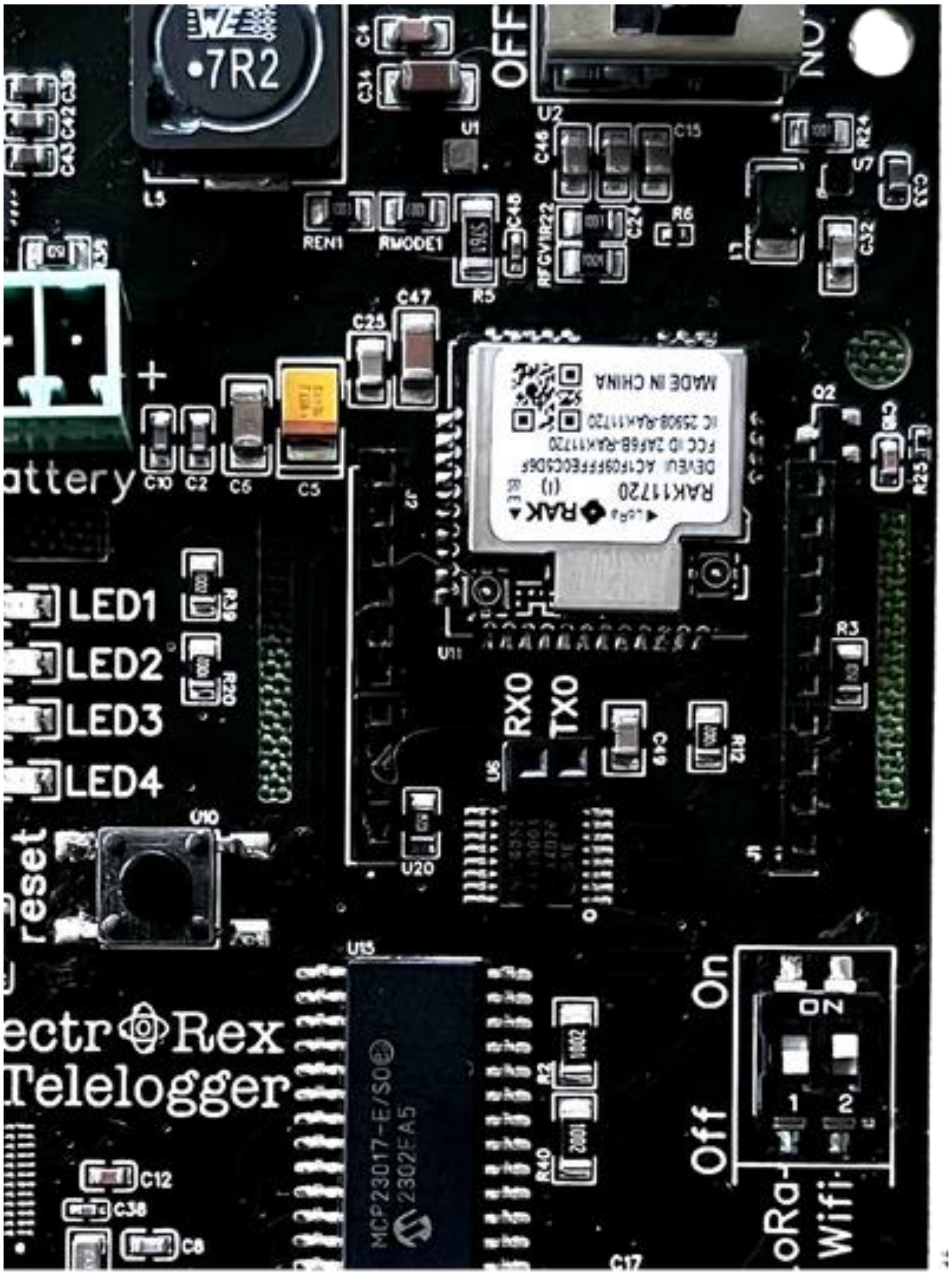
Data is sent by the gateway to a network that routes it to a server of your choice. In this tutorial we are going to use The Things Network (TTN) to route the data to one of the IoT databases we setup for sending data with the WiFi modem (Ubidots).

Requirements for LoRa in this tutorial:

1. LoRa antenna attached to the LoRa modem. **Do not power the LoRa modem without an antenna attached as this could damage the modem.** We provide a small flexible LoRa antenna glued to the modem antenna ipex connector. This is for two reasons. First, the reason stated above, do not power the modem without an antenna. Second, these connectors (ipex4) can become disconnected easily. Third, frequent connecting and disconnecting of the antenna on these small ipex4 connectors can damage it. They are popular with manufacturers like RAK Wireless because they do not take up a lot of

space. We suggest to keep the current antenna sold with the board in place until it is absolutely necessary to change it to a larger antenna and then carefully replace it with an ipex4-to-SMA adapter cable, to which is then attached to an antenna with SMA connection. You could also attach an SMA-to-N-type adapter to an antenna with an N-type connector. See our antenna recommendations page here.

2. A LoRaWAN gateway You can check the the Things Network coverage map here for available gateways near your location. Better yet, you can help grow the network and purchase your own gateway and set it up on The Things Network. The folks at The Things Network have released a very affordable indoor home gateway (such as here). Instructions are very easy to follow for setting it up on the Things Stack (see here).



The LoRa modem by RAK Wireless on the Panther Logger board

LoRa Terminal Communication

We are going to start by uploading the LoRa_Terminal sketch to the Panther Logger so we can start sending AT commands manually (one-by-one) to the LoRa module for testing purposes. Before doing that make sure a LoRa antenna is connected to the LoRa module. **Powering and sending or receiving data without an antenna attached to the LoRa can damage it.** The LoRa antenna goes on the right ipex4 connector. After the antenna is connected, turn on the dip switch for the LoRa modem.

Go to our Github page for the LoRa_Terminal sketch here. Copy/paste into a new Arduino sketch and upload it. After uploading, open the serial monitor window. When you see setup finished we can now start sending AT commands in the upper send box. Make sure the ending string is set to both "NL & CR". Start by sending:

AT

You should receive:

OK

If you sent it too early you could get an error. Just send it a couple more times.

Now we need to get some information from the module including the APPEUI, the DEVEUI and the APPKEY. We can get them with the following AT commands:

```
AT+APPEUI=?
AT+DEVEUI=?
AT+APPKEY=?
```

For my module these return the following:

```
AT+APPEUI=?

AT+APPEUI=AC1F09FFF9151720
OK

AT+DEVEUI=?

AT+DEVEUI=AC1F09FFFE0C5D79
OK

AT+APPKEY=?

AT+APPKEY=AC1F09FFFE0C5D79AC1F09FFF9151720
OK
```

Your APPEUI (also called join EUI), DEVEUI, and APPKEY will be different.

Leave the serial monitor up or copy these numbers because we will need these numbers later to setup our device on the Things Network.

Setting Up an Application and New End Device at TTN

Go to The Things Network (TTN) here, and click on the sign up button in the upper right. Then click on Join The Things Network for a free account (community edition). Follow instructions to setup a free account. After logging into your account go to the console (click on user name, then on console). Click on applications and then create an application (see video to the right). Click on register end device, then enter device specifics manually.

Give your device an ID, all lower case
Give your device a name and description
For the frequency plan select FSB2, 902 - 928 MHz United States
Choose specification 1.03
Enter the APPEUI and then the DEVEUI and APPKEY from the serial window

Click register end device and now you are finished.

Send Your First Join Command

Your LoRa modem is now ready to join the TTN and send data to the application. We will need to send a series of AT commands to setup the modem to join TTN and then one AT command to send some mock data. Enter the series of AT commands below in the send box of the Arduino serial monitor, line by line, hitting enter after each one. This assumes you still have the LoRa_Terminal sketch loaded onto the Panther Logger. If it has been awhile since you last sent AT commands to the LoRa modem you might need to reset the board, close the serial monitor and re-open it.

Replace the X's in AT commands below with your device info (APPKEY, DEVEUI, and APPEUI). If you want to learn what each AT command is doing, the AT command manual for the modem is here. Enter the following line-by-line in the serial monitor send window and hit enter after each command is written. Be sure the line ending is set to "Both NL & CR"

AT+NJM=1

OK

AT+NWM=1

OK

//Wait for the following....

```
RAKwireless RAK11720 BLE Example
------------------------------------------------------
Current Work Mode: LoRaWAN.

AT+PNM=1

OK

AT+CFM=0

OK

AT+CLASS=A

OK

AT+BAND=5

OK

AT+MASK=0002

OK

AT+ADR=1

OK

AT+DR=1

OK

AT+RETY=2

OK

AT+DEVEUI=XXXXXXXXXXXXXXXX

OK

AT+APPEUI=XXXXXXXXXXXXXXXX

OK

AT+APPKEY=XXXXXXXXXXXXXXXXXXXX
```

```
OK

AT+JOIN=1:1:8:10

+EVT:JOINED              //Wait for the join response. Can take several minutes

AT+SEND=1:00170038       //Send the data

OK
```

This sends the data string 00170038. This is our hexadecimal representation of the number 23.56 (for example, as in temperature data, 23.56 degrees Celsius). The hexadecimal version of 23 is 17 in hex and 56 is 38 in hex. The leading zeros indicate that each is a positive number. The LoRa modem does not accept decimal numbers so we must convert to hexadecimal and then it gets decoded back to decimal at TTN by our payload decoder (discussed below).

Go to TTN and click on applications and name of the application you created and then on end devices and then on your device. In the live data tab you should now see a new entry for "Forward Join Accept Message" and "Forward Uplink Data Message" that contains the hexadecimal numbers we sent.

Setup Payload Decoder at TTN

We sent data above to TTN, but if we want to collect the data in a database and display it in a dashboard online then we will need to put it somewhere. Below, we will setup a payload decoder so that we can send decimal data to a database . Go to your TTN account and then to Applications, then End Devices and click on your device listed by EUI number (see video to the right). Then click on the payload formatters tab. We are going to setup the uplink decoder. Choose custom javascript formatter in the drop down box. Copy/paste the following in the formatter code box:

```
function Decoder(payload, port) {
   if(port === 1) {
      return [
         {
            "temperature": ((payload[0] & 0x80 ? 0xFFFF<<16 : 0) | payload[0]<<8 |
payload[1])+(((payload[2] & 0x80 ? 0xFFFF<<16 : 0) | payload[2]<<8 | payload[3]))/100
         }
   ];
   }
}
```

This is a decoder function to decode a hexadecimal string of numbers that represent sensor float decimal data that can be positive or negative. It expects 8 hexadecimal digits like our temperature string that was sent above, 00 17 00 38. See an example payload for a weather station here and example function to decode that payload here. That example provides a variety of use cases that should work for most environmental monitoring data.

On the TTN javascript payloader page you can test the decoder on the right by entering a hexadecimel string. The formatter is expecting 8 hex digits so enter the data string we will send in hexadecimal format representing temperature data, 00170038 in the byte payload box and click test decoder. You should see the following in the decoded payload test window.

```
{
  "0": {
    "temperature": 23.56
  }
}
```

The javascript code converts the hexadecimal payload to decimal.

Click on save changes to save the payload decoder.

Go back to Arduino and assuming the terminal program is still loaded on the Panther Logger then in the serial window run the series of AT commands above ending with the AT+SEND command again.

Now go back to TTN and look at the live data for your application for this device. You should see another message with the payload sent. Click on it and a window opens on the right with information about the message in JSON format. Scroll down and you will see a similar JSON result as we saw in the test decoder window. It will look like the following:

```
"uplink_message": {
      "session_key_id": "AYyJJIeA3C2yQqrw3qEZqQ==",
      "f_port": 1,
      "f_cnt": 1,
      "frm_payload": "ABcAOA==",
      "decoded_payload": {
        "0": {
          "temperature": 23.56
        }
      },
```

Notice that the temperature data is now decoded and formatted how we would expect, as decimal format. It is also in the format expected by Ubidots. Note that other databases might be expecting a slightly different format so the payload decoder might need to be modified. There is a lot of other information in this message including antenna signal strength ("rssi") and signal to noise ratio ("snr") as well as the timestamp and geolocation of the gateway that sent the data. This entire message will now be sent by TTN to Ubidots with a webhook.

Sending Data to Ubidots

To send data from TTN we need to use a webhook. In TTN applications click on your application, end devices and then click on your device. On the left click on integrations and webhook. Click on Add Webhook. You will see pre-configured webhooks for a number of different IoT databases including those used in the WiFi tutorials. Click on the one for Ubidots. For the webhook ID choose anything, all lower case. I'll use "ubidots."

For the plugin ID and token we will need to setup the plugin first in our Ubidots account. Login to your Ubidots account (see here to get a free stem account), click on devices and then on plugins. Add a plugin with the plus "+" button in the upper right corner and then scroll down to the Things Stack plugin. Click on it and then scroll down and click on the right arrow and then on the next screen choose the default token from the drop down list. Click on the right arrow and then the green check mark to complete. The plugin is now listed in your list of plugins. Click on it to configure it.

On the decoder tab you will see the endpoint URL listed. Copy that and then go back to TTN in the webhook setup window and paste it into the plugin ID box. Then erase this part:

https://dataplugin.ubidots.com/api/web-hook/

We only want the last alphanumeric code listed. This is the unique endpoint for your device at Ubidots.

You do not need to create the device at Ubidots. The first time you send data from the device will be created automatically named with your device eui.

Go back to Ubidots and in the upper right click on your profile picture and then on API credentials. Copy the default token and then paste this into the Ubidots Token box in the TTN webhook setup page. Click create webhook.

Back at Ubidots on the decoder tab of your plugin look at the default decoder function listed. Uncomment this line below by removing the two slash marks in front of it:

var decoded_payload = args['uplink_message']['decoded_payload']

Then add ['0'] at the end so it looks like this:

var decoded_payload = args['uplink_message']['decoded_payload']['0']

Why did we do this? Look at the uplink message again we saw above at TTN. The path to our temperature data is **uplink_message>decoded_payload>0**

You will also see that this default decoder is grabbing some other useful information like the signal strength and signal to noise ratio.

One more thing that needs to be done. We are not actually decoding data here. That is being done with the payload formatter at TTN. So scroll down and comment out the two lines below by putting two slashes in front of each line.

let bytes = Buffer.from(args['uplink_message']['frm_payload'], 'base64');
var decoded_payload = decodeUplink(bytes)['data'];

These lines send the data received to the function called decodeUplink to decode the message. We don't need that so you can just delete that function, or you can leave it as it, does not matter.

Click "Save and Make Live" button and then click on devices tab and Devices. Go back to Arduino and run the series of AT commands we ran above again ending in the AT+SEND command. You should see a new message at TTN under live data for your device and a new device show up with data at Ubidots. You might need to refresh the page at Ubidots to see this.

If you see new data show up at Ubidots then you have now sent data with the Panther Logger over LoRa and that process can now be easily automated in code (below).

Sending Data to Ubidots Automated

We can now send sensor data over LoRa with our Panther Logger through a gateway on TTN. However, we have only done it manually with the terminal sketch. Let's automate this to happen on a set time schedule. It is important to keep in mind here that if you have a free account with TTN that you are bound by the fair use policy, which stipulates that you can have:

- An average of 30 seconds uplink time on air, per 24 hours, per device.
- At most 10 downlink messages per 24 hours, including the ACKs for confirmed uplinks.

In other words... "A good goal is to keep the application payload under 12 bytes, and the interval between messages at least several minutes." This is explained here.

We will automate a script to send mock temperature and battery data to Ubidots over LoRa about every 5 minutes.

Go to our Github page for the LoRa_Send_Data sketch here. Just like code for sending data via cellular, this sketch has multiple tabs. Copy and paste each file into a new sketch with multiple tabs as was explained here for cellular sketches and save with a name of your choice.

This sketch expects that the DS18B20 temperature sensor is attached and that you know its address. See previous tutorial on how to attach this sensor to the Panther Logger board and on reading these temperature sensors by their address in our tutorial on these sensors. The

Send_Data_Ubidots_LoRa sketch will read these temperature sensors and battery voltage, convert the data to hexadecimal values and create the payload, then send the payload with the LoRa modem every five minutes.

Since this sketch will send more than one data variable we need to edit the payload formatter at TTN. Go to the formatter and replace it with the below code (it is also in our Github repository here).

```
function Decoder(payload, port) {
if(port === 1) {
return [
         {
"BATV": payload[0]+(payload[1]/100),
"Temp0": ((payload[2] & 0x80 ? 0xFFFF<<16 : 2) | payload[2]<<8 |
payload[3])+(((payload[4] & 0x80 ? 0xFFFF<<16 : 0) | payload[4]<<8 | payload[5]))/100,
"Temp1": ((payload[6] & 0x80 ? 0xFFFF<<16 : 0) | payload[6]<<8 |
payload[7])+(((payload[8] & 0x80 ? 0xFFFF<<16 : 0) | payload[8]<<8 | payload[9]))/100,
"Temp2": ((payload[10] & 0x80 ? 0xFFFF<<16 : 0) | payload[10]<<8 |
payload[11])+(((payload[12] & 0x80 ? 0xFFFF<<16 : 0) | payload[12]<<8 |
payload[13]))/100
         }
      ];
   }
}
```

This just takes the existing formatter and replicates the line of code to decode temperature data two more times and adds one more line first to decode battery voltage.

Save the changes to the formatter and go back to the Arduino sketch.

To get started, find the code below at the beginning and replace the X's with your device's DEVEUI, APPEUI, and APPKEY:

```
char DEVEUI[] = "AT+DEVEUI=XXXXXXXXXXXXXXX";
char APPEUI[] = "AT+APPEUI=XXXXXXXXXXXXXXX";
char APPKEY[] = "AT+APPKEY=XXXXXXXXXXXXXXXXXXXXXXXXX";
```

Go to the code listed below and replace the address with your DS18B20 temperature sensor address repeated three times (or you can wire three different ones together!).

```
byte Address[NSENSORS][8] = {
{0x28, 0xD7, 0xA8, 0xE2, 0x08, 0x00, 0x00, 0x39},
{0x28, 0xD7, 0xA8, 0xE2, 0x08, 0x00, 0x00, 0x39},
{0x28, 0xD7, 0xA8, 0xE2, 0x08, 0x00, 0x00, 0x39}
};
```

Note that you can change the interval between data transmissions, but we do not suggest a shorter interval than 5 minutes as it could violate The Things Network fair use policy.

Upload the sketch to the Panther Logger board and watch for new uplink data messages at TTN and new data in your device at Ubidots.

There is a function, hexConvert1, in this sketch on the "Payload" tab to convert a floating point decimal number to hexadecimal where we expect no more than three numbers before the decimal and want two after the decimal. This also handles negative data. So temperature data would be a good example.

```
char HexOutThreeNeg[10];
char* hexConvert1(float var){
char HexVar[10];
char FloatChar[10];
dtostrf(var,3,2,FloatChar);
float VarConvert = atof(FloatChar);
int16_t Var1 = VarConvert;
int16_t Var2 = (VarConvert - Var1)*100;

sprintf(HexVar, "%04hX%04hX",
        Var1,
        Var2
);
strcpy(HexOutThreeNeg,HexVar);
return(HexOutThreeNeg);
}
```

Decimal data needs to be converted to hexadecimal. To do this, we first need to make sure we know how many numbers to expect before and after the decimal. We can set this a priori with the dtostrf() function. This function will convert a floating point value to a character string and in the process allow us to set the number of digits.

```
dtostrf(var,3,2,FloatChar);
```

var = the float value we want to convert
3 = number of digits before decimal
2 = number of digits to limit after decimal
FloatChar = Where to put the result, created with char FloatChar[10];

The result is a string representation of the float decimal with a set number of digits before and after the decimal.

Then we can convert back to a float with the atof function and continue with the conversion.

```
float VarConvert = atof(FloatChar);
```

We now have a float number called VarConvert with a defined number of digits before and after the decimal. Now we need to separate the numbers before the decimal from those after the decimal. We can get just the numbers before the decimal by converting to an integer:

```
int16_t Var1 = VarConvert;
```

So if VarConvert was 23.56 then Var1 would now be 23.

Well if we subtract Var1 from VarConvert we get 23.56 - 23 = 0.56. And if we multiply by 100 we get 56, which we can store as an integer. We do this with the following:

```
int16_t Var2 = (VarConvert - Var1)*100;
```

Now we have the digits before and after decimal separated. We can now convert to hexadecimal and a character string simultaneously using the sprintf function.

```
sprintf(HexVar, "%04hX%04hX",
        Var1,
        Var2
);
```

The %04hX tells the sprintf function we want to output hex (this is the X) and use four digits. We use four because two will be used to represent a decimal number up to 255 (good enough for ambient temperature data) and another two to represent the sign as a negative or positive number. We copy the result to a global character string that can be used outside of this function.

There are other functions to convert data of other types to hexadecimal such as data that is expected to have four digits before the decimal, but not negative data.

The sketch uses these functions to convert sensor data to hexadecimal and then uses the AT commands we showed above to send the data to TTN.

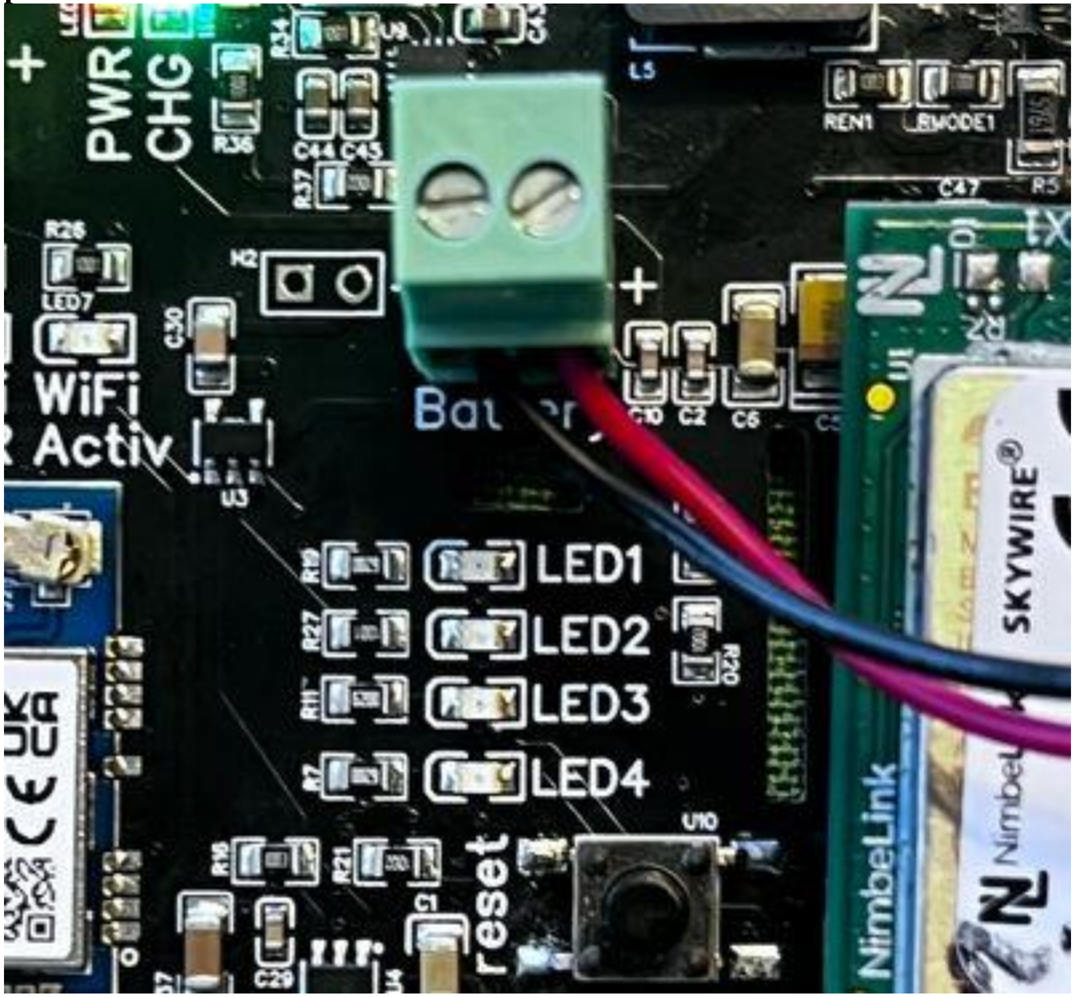
This sketch could be improved prior to deployment to do things like use millis() for more exact timing, possibly use sleep functions, include a software watchdog and other functions to ensure the program continues to send data during long deployments.

Overview

The Panther Logger board is designed for long term deployments powered by a battery with or without solar charging or other charging source. During deployments the entire board can be powered from a battery and attached sensors powered from one of the available 3.3V, 5V or 12V regulated power supplies at the convenient screw terminal connections.

When you first start programming the board for your application you will power it from USB cable attached to your computer and if using a cell modem then you will also need a battery

(battery recommendations here) attached to the two pin screw terminal block marked "Battery." This will provide power to the cell modem and charge the battery. The battery can be charged when either USB power is connected to the micro USB connector or a DC power source (e.g. solar panel) is plugged into the two pin screw terminal block marked "solar." It can also be powered from a USB wall outlet if one is available near your deployment location. **All input power must be less than 10V.**

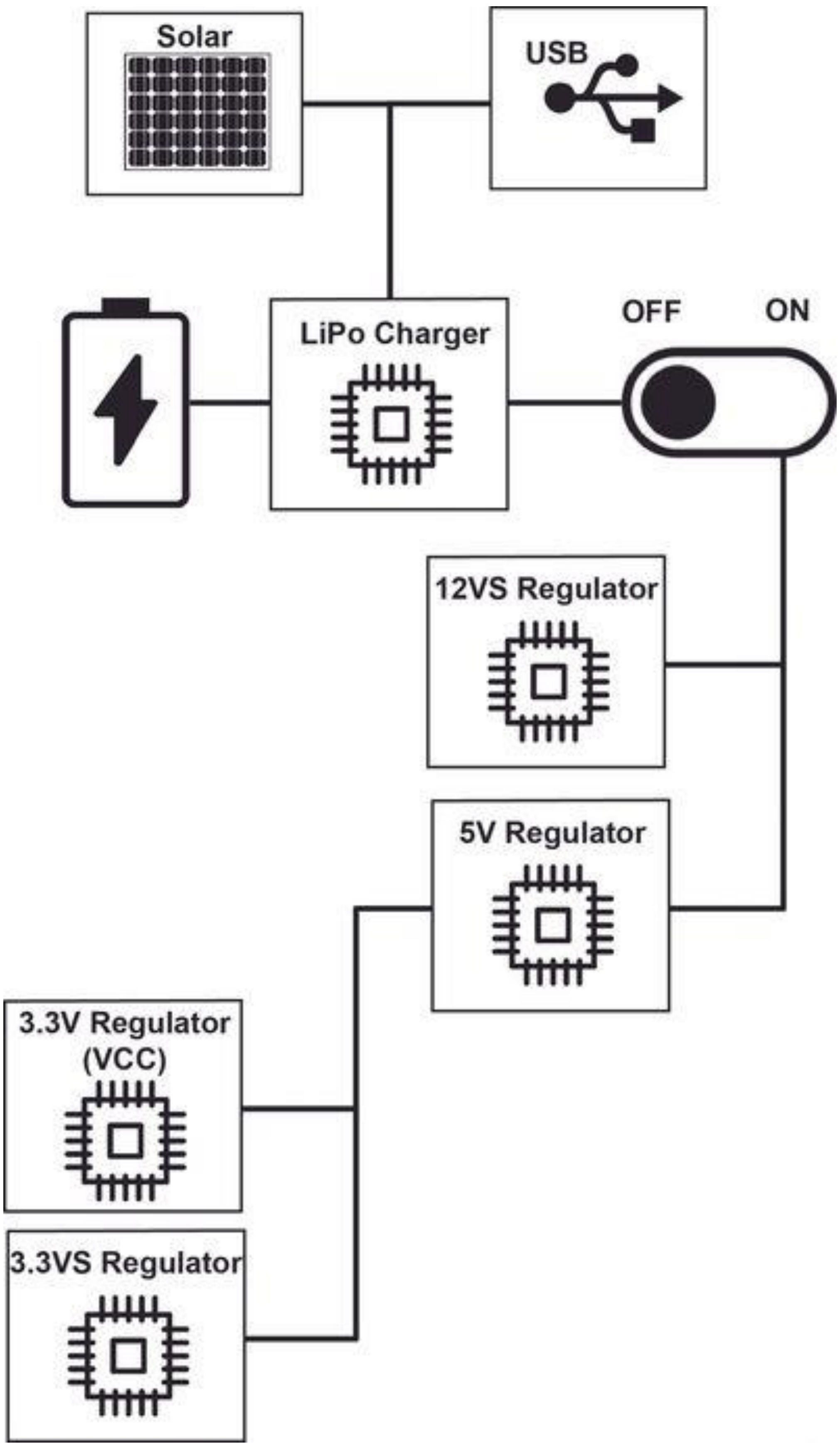


Power and Charging LEDs on the Panther Logger Board

The Panther Logger Power System

Solar or USB power is regulated to charge the battery by the "LiPo Charger" chip which in turn provides 3.3V output to the 12V boost/switched and 5V boost regulators. These two boost regulators make the 12V and 5V power supplies. The 5V power supply is used to create VCC (3.3V) and a switched, regulated 3.3V power supplies using low drop out regulators. The on/off switch in the upper right part of the board will turn power on or off to the entire board except for the battery charging circuit and the cell modem. **As such, even if the power switch is off, the cell modem will still be using battery power if the battery is attached and the battery will still be charging if USB is attached. This is so that the board can act as a battery charger**

while not running code. <u>Do not charge the battery unattended</u>. <u>*The board is completely off when USB, battery and any power source connected to the solar screw terminal plug are disconnected.*</u>


Solar
USB
LiPo Charger
OFF
ON
12VS Regulator
5V Regulator
3.3V Regulator (VCC)
3.3VS Regulator

The Panther Logger Power System

Current Limits and Sleep

Current limits (mAmps) are limited by each power supply as stated in board specifications (here). The WiFi and LoRa modems are powered from the switched 3.3VS rails which has a max current rating of 800 mAmps. The WiFi and LoRa modems could use up to 230 and 87 mAmps , respectively, while sending/receiving data. Ensure that attached sensors and peripherals do not exceed the current limits including current draw from these modems if they are in use.

The microprocessor can go into sleep and deep sleep modes and all power rails except the 5V and VCC rails can be turned off in code to save power. So, it is very possible to build up an environmental monitoring observatory platform with the Panther Logger that sleeps at low microamps range in between loop iterations. We give examples of this in the Telelogger tutorials.

Panther Logger 7: Battery Recommendations

The Panther Buoy requires a 3.7V LiPo battery. We will list some batteries here that we have tested with the Panther Buoy.

1. Adafruit 3.7V 6600mAh LiPo battery, Product ID: 353, https://www.adafruit.com/product/353
2. Adafruit 3.7V 4400mAh LiPo battery, Product ID: 354, https://www.adafruit.com/product/354
3. Maker Focus 3.7V 10,000 mAh LiPo battery, https://www.amazon.com/dp/B093WS6C66
4. Adafruit 3.7V 10,050mAh LiPo battery, Product ID: 5035, https://www.adafruit.com/product/5035 (NOTE: this is a newer more dense version of #1 above. Same size, more capacity!)

The Panther Buoy battery terminal accepts bare wires as it is a screw terminal. You will have to cut off the JST connector that comes with the battery and strip back the insulation 1 to 2 cm to expose bare wire and then insert into the 2-plug screw terminal for the battery on the Panther Buoy (positive+ and negative- are marked on the board). **When cutting off the connector cut one wire at a time. Do not let the bare negative and positive wires (red and black) touch each other or you will short out the battery and could damage it.**

Panther Logger 8: Read an In Situ RDO Pro Using SDI12

<-- Back to Panther Logger Tutorials
In this tutorial we will wire an In-Situ RDO Pro dissolved oxygen sensor to the Panther Logger and program the logger to read the sensor.

Sections in this tutorial

- Introduction
- Wiring
- Programming

Introduction

The In-Situ RDO Pro sensor is a standalone sensor for measuring dissolved oxygen in aquatic monitoring applications. The Panther Logger can read the RDO Pro using SDI12 and power it with the 12VS rail. In this tutorial we test sensor communications on the bench top outside of an environmental enclosure. However, in a real field deployment one would need to place the Panther Logger and battery in an enclosure that prevents water and dirt intrusion with the RDO pro cable and wires securely passed through the enclosure.

For this tutorial you will need:

1. Panther Logger Board and USB cable.
2. In-Situ RDO Pro Sensor with flying lead wires stripped to bare wire about 5 mm.

The RDO Pro should have its DO cap installed according to the manual. Go to In-Situ's website to find the manual. You should also configure and test the sensor as indicated by the manufacturer using their software first.

This sensor reports water temperature, dissolved oxygen as mass per volume (mg/L) and percent dissolved oxygen saturation.

The In-Situ RDO Pro dissolved oxygen sensor

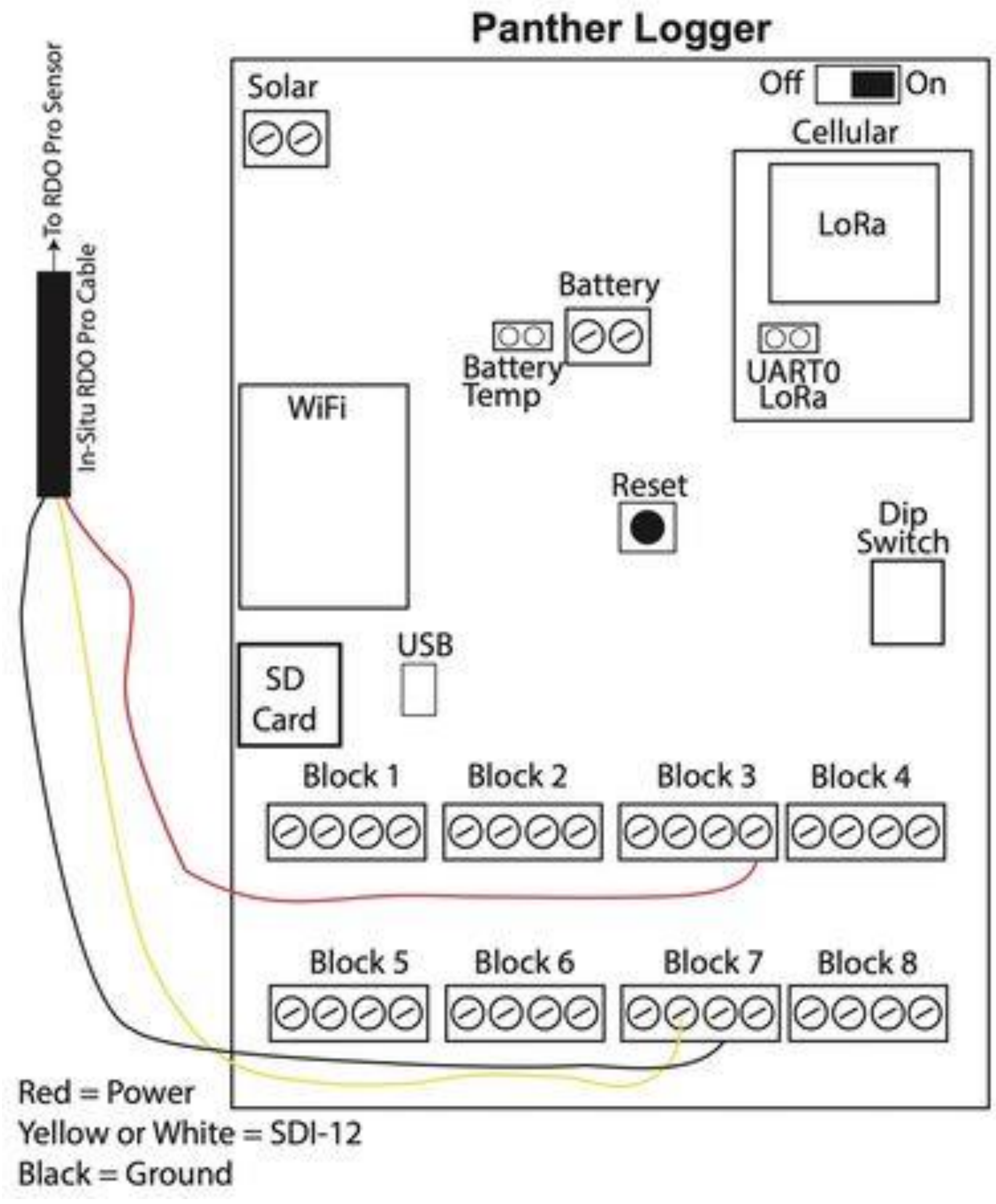
Wiring

There are only three wires needed for SDI12 communication and to power the sensor. These are wires for power, ground, and signal. The Panther Logger D6 or D11 pins (marked on the board) can be used for signal. The RDO Pro sensor requires at least 9V to run, so either of the two switched 12VS power rail pins on the Panther Logger should be used to power this sensor. The ground wire goes to any ground pin on the Panther Logger. Note that terminal block 7 on the Panther Logger has D11, GND, and 12VS altogether in a row so it is the ideal location to attach this sensor's wires to the Panther Logger. Note that SDI12 communication is addressable so multiple SDI12 sensors could be connected here.

Attach the correct wires from the RDO Pro sensor to the corresponding Panther Logger screw terminals (use image at right as a guide). Be sure to check the In-Situ RDO Pro manual for wire color definitions, but last time we looked it was:

- Red = Power (12VS)
- Black = Ground (GND)
- White or Yellow = SDI-12 Signal (D11 or D6)

At this point the RDO Pro should be wired correctly and ready for programming.



Programming

Go to our Github repository here and grab the "RDO_SDI12" example code to program the Panther Logger to read the RDO Pro sensor. Just copy/paste the code to a new Arduino script, save it and upload to the Panther board. This sensor outputs three variables including dissolved oxygen in mg/L, dissolved oxygen percent saturation and water temperature in Celsius. As such the code creates three float variables to hold these data.

The meat of the code is in the function called readRDO(). That function is pasted below:

```
void readRDO() {
mcp.pinMode(7, OUTPUT);
mcp.digitalWrite(7, HIGH); //Turn on 12VS rail

  DO = -9999;
  DOS = -9999;
  RDOTemp = -9999;
unsigned int RDONowTime = millis();
unsigned int RDOInterval = 10000; //Give sensor 10 seconds to send good data
while(RDONowTime + RDOInterval > millis()){
if(DO == -9999 || DOS == -9999 || RDOTemp == -9999){
sdi12.begin();
sdi12.clearBuffer();
float RDOData[4] = {0};
sdi12.clearBuffer();
     String command1 = "4M!"; //Tell sensor to make a measurement
sdi12.sendCommand(command1);
delay(100);
     String sdiResponse = sdi12.readStringUntil('\n');
Serial.println(sdiResponse);
delay(3000); //RDO pro needs two seconds to make measurement
sdi12.clearBuffer();
     String command2 = "4D0!"; //Tell sensor to send us the data
sdi12.sendCommand(command2);
delay(100);

for (uint8_t i = 0; i <= 3; i++){
RDOData[i] = sdi12.parseFloat();
}

     DO = RDOData[1];
     DOS = RDOData[2];
     RDOTemp = RDOData[3];

sdi12.clearBuffer();
sdi12.end();
}
}
mcp.pinMode(7, OUTPUT);
mcp.digitalWrite(7, LOW); //Turn off 12VS rail
}
```

The first thing this code does is to fill the float variables with -9999. This way we know that if at the end of the loop those floats are still -9999 then data was most likely not received from the sensor.

The code then enters a while loop that will repeat reading of the sensor for a predetermined time period. This will provide a mechanism for the board to poll the sensor for data again if prior attempts did not yield data (i.e. replace -9999 with new data). Then there is an IF statement such that if sensor data is -9999 then the sensor will be polled for new data, otherwise measurements will not repeat.

To make SDI-12 measurements within the while loop we use functions within the handy Arduino-SDI-12 library from the folks at EnviroDIY, here.

SDI12 data is read from the sensor first by requesting that the sensor take a measurement with the sendCommand() function. Look at the code where the sendCommand() function is used. The function sends commands to the sensor in SDI-12 format. The SDI-12 command consists of the sensor address (in this case it is 4) followed by the M command and all commands ending in exclamation point (!). We read back the sensor response with the sdi12.readStringUntil() function and in this case read until the end of the line or until a new line is given ("\n"). The sensor will return the time it will take before the measurement is complete and data available. For this sensor it is about two seconds.

We give it three seconds with a delay to ensure we give it enough time. Then we send the command to get the data. This command again begins with the sensor address (4) followed by D0 (for data) and exclamation point. We read the data back into a float array (RDOData) with the parseFloat() function. We can then put each array value into the individual float variables for the two dissolved oxygen variables and water temperature.

The three variables are within this "0" segment of data, but if the sensor returned more data then we would need to subsequently issue a "D1" command and then read the data into another float array. See the tutorial on reading the In-Situ Aqua Troll sonde for an example of reading a larger number of variables from an SDI-12 sensor.

Panther Logger 9: Read a YSI EXO Sonde with RS232

Introduction

The YSI EXO instrument is a multi-parameter data sonde that is an industry standard in aquatic monitoring programs. The Panther Logger can power and read this instrument in long term deployments, such as on a data buoy or a fixed monitoring station over a river. In this tutorial we show how to wire up the EXO2 sonde with the Panther Logger and program the Panther Logger to request, receive and parse serial data from the EXO2 using the Panther Logger's RS232 serial interface.

In order to read the EXO2 sonde with the Panther Logger you will need the following:

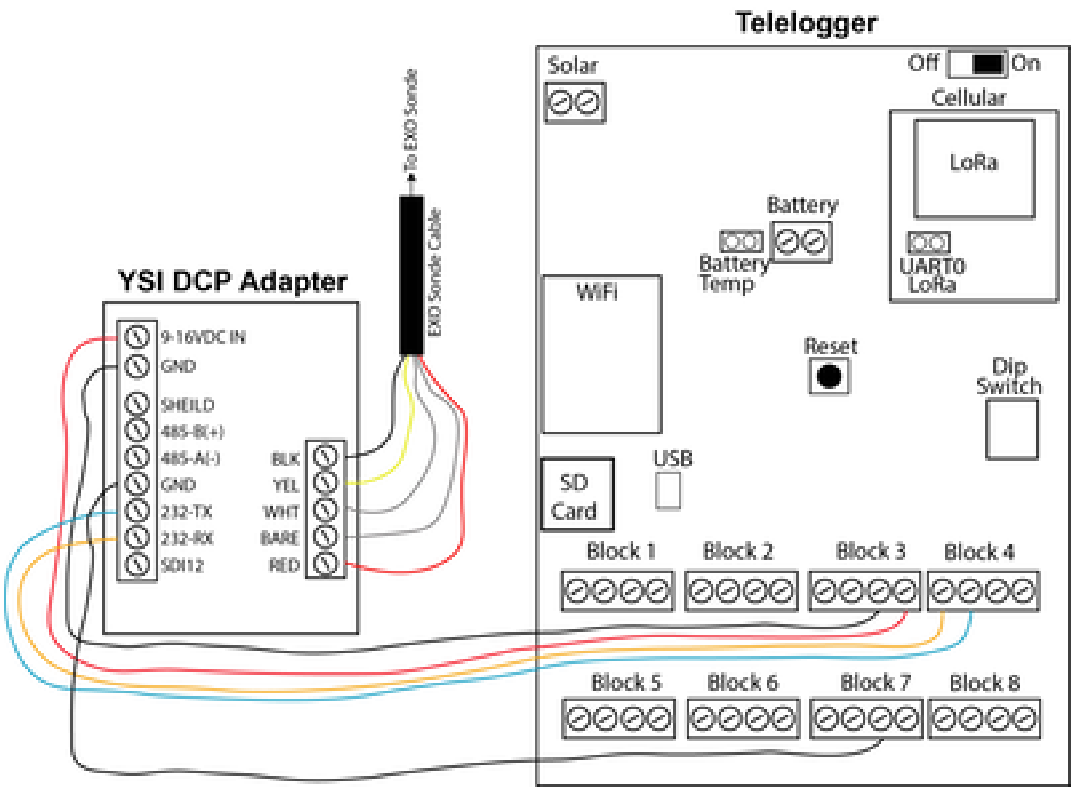
1. EXO2 Sonde with cable
2. DCP Adapter
3. 24 - 26 AWG hookup wire of various colors (stranded)
4. Panther Logger Board
5. Battery or power supply set to 3.7 to 4.2V

The EXO2 sonde does not output RS232 signals by itself. To do this, YSI sells their customers a device called a DCP Adapter which translates YSI's proprietary signal into RS232 and SDI12. Use of this adapter is discussed in the EXO2 manual (link here). Note that the EXO3 sonde has integrated SDI12 output. As such it would make more sense to read an EXO3 sonde with the Panther Logger using SDI12 since the DCP adapter would not be needed.

Wiring the EXO to the Panther Logger Board

Before doing any wiring make sure the battery on the Panther Logger is unplugged and that no USB cable or solar charging source is plugged into the Panther Logger. In this tutorial we are wiring the EXO sonde and DCP adapter on the bench top indoors just for testing purposes. However, keep in mind that for outdoor deployment the Panther Logger board, battery and in this case also the DCP adapter must be in some type of environmental enclosure that prevents water and dirt intrusion. Therefore, wires from the EXO sonde cable must go through the enclosure using a connector or cable gland and this cable should have tension release.

The EXO sonde cable should terminate in flying lead wires with the insulation stripped back about 5 mm. These are plugged into the five screw terminal block of the DCP adapter according to wire colors marked, as explained in the EXO sonde user manual (page 35). On the other side of the DCP adapter run a red hookup wire from the screw terminal marked "9-16 VDC IN" to one of the 12VS screw terminals of the Panther Logger. Run two black wires from the two screw terminals on the DCP adapter marked "GND" to ground (GND) pins of the Panther Logger. Run another wire color from the terminal marked "RS232 - RX" on the DCP adapter to the TX2 pin on the Panther Logger and another wire color from the terminal marked "RS232-TX" on the DCP adapter to the RX2 pin on the Panther Logger. See the image to the right as a guide.



Programming the Panther Logger to Read the EXO Sonde

The EXO sonde requires that we send it RS232 commands (page 38 of the EXO manual). These commands can be used to activate the sonde's wiper, wait for the dissolved oxygen sensor to warm up, change baud rates and request data among a few other tasks. Commands must end with a carriage return (i.e. "/r").

Go to our Github page for the "ReadEXO_RS232" script here. Copy and paste into a new script in the Arduino IDE and save it. This script powers on the EXO with the switched 12VS rail of the Panther Logger, then waits for it to startup and complete one wipe of the sensors, then requests data with the command "data/r" , reads it back and then parses that data into individual float values. The script then turns off the 12VS switched rail and the EXO sonde powers down. After a delay of 10 seconds the loop repeats.

The data/r command returns a comma separated string of values. This string of values is defined in the sonde. To identify this string of values it is necessary to connect the sonde to a PC using the YSI USB adapter for EXO sondes and the KOR EXO software. In the deployment section of the software the order of sensor output variables can be chosen.

With the DCP adaptor and EXO sonde hooked up to the Panther Logger, attach the USB cable to the Panther Logger board to power it and to your computer. Plug in the battery if available. Upload the ReadEXO_RS232 script to the Panther Logger board and open the serial monitor. You will see the serial monitor report when it is turning on th EXO and waiting for it to complete a wipe of its sensors. At the same time you should see the blue and red LEDs on the sonde turn on. The LED on the DCP adaptor will turn on solid for several seconds and then turn off. After about 20 seconds the DCP adaptor LED will start to blink and then data will be requested by the Panther Logger, which will show up in the serial monitor.

Read an In-Situ Aqua Troll Sonde

<--Back to Panther Logger Tutorials

For now, see the example code on our Github Panther Logger repository here

Introduction

The Aqua Troll from In-Situ is a multi parameter sonde from in-situ.com. Available sensors include water temperature, dissolved oxygen, conductivity, algal pigments and others. See in-situ.com for more details. The Panther Logger can read the Aqua Troll using SDI12 or RS232 communications. For this tutorial we will use SDI12. Note that this procedure is very similar to reading the In-Situ RDO Pro dissolved oxygen sensor as explained in our previous tutorial. One difference is that the Aqua Troll reports more variables. To get all of them we need to do additional data request commands.

You should setup the sonde's SDI12 communications including the order of variables to be sent via SDI12 either using VuSitu software on a PC or by using the smart phone app over blue tooth. See the manual for your AquaTroll sonde to set this up. At the very least we need to know the order of variables that it will send via SDI12 and the SDI12 address it was assigned.

In-Situ Aqua Troll sonde. See in-situ.com for more details.

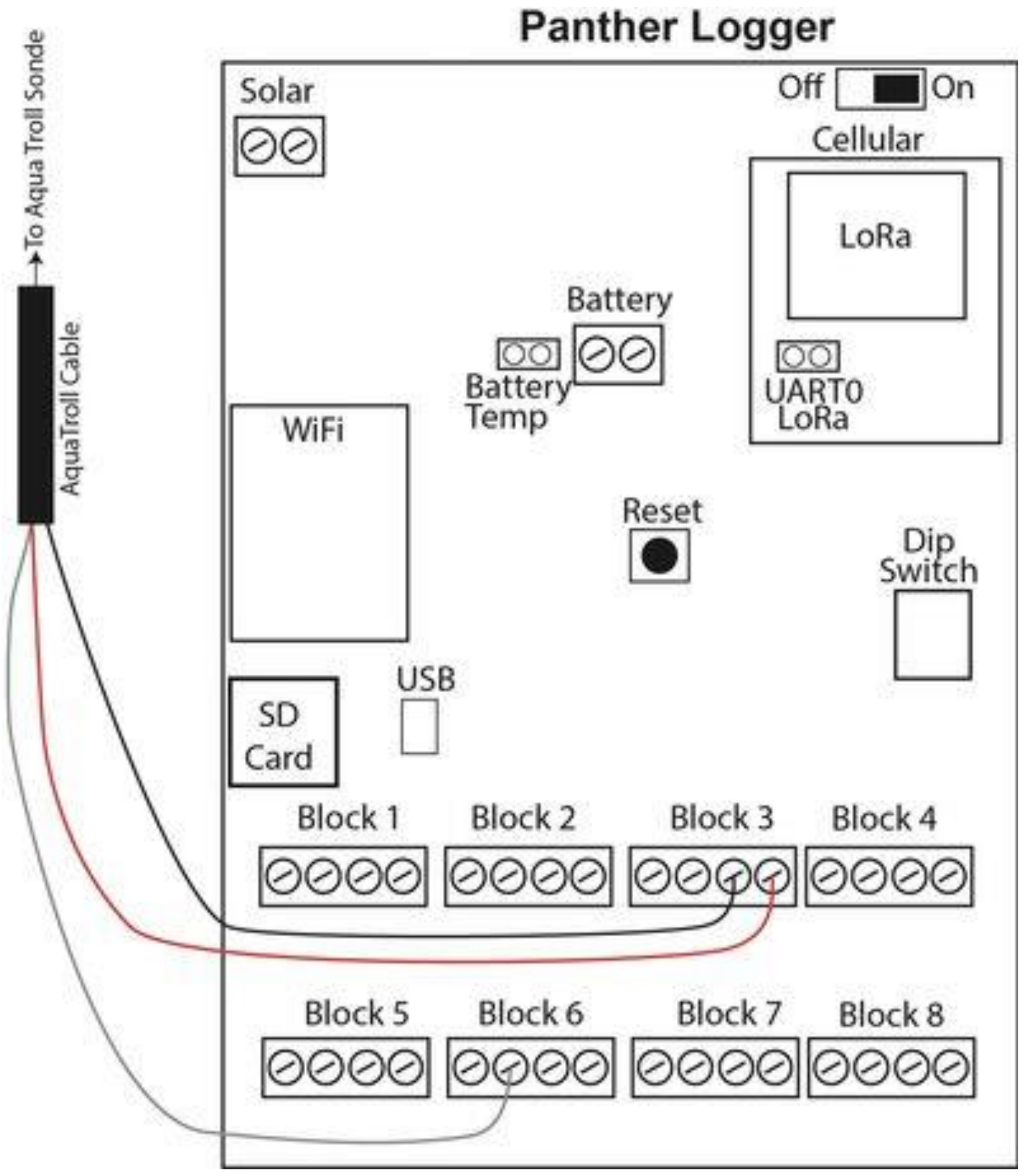
Wiring

The cable that comes with the Aqua Troll should terminate at a twist lock connector on one end that connects to the sonde and the other end should be flying leads (bare wires). We will be using the red, black and grey wires, but check the manual for your AquaTroll to ensure colors match the function needed. Wires should be connected to the Panther Logger according to the list below. Wiring is also shown in the image to the right.

Wire Color = Function --> Screw Terminal on Panther Logger

- Red = power (12V) --> 12VS
- Black = ground --> GND
- Grey = SDI12 signal -- > D6 or D11

Note that the example code we provide expects SDI12 to be on D11, but this can be easily changed in code to D6. The sonde requires at least 8V and will pull about 50 mA while sending data, a little more while wiping.



Programming

To program the Panther Logger to read the Aqua Troll sonde go to our Github site for the AquaTroll SDI12 code here. Copy and paste into a new Arduino sketch, save under a new name of your choice and upload to the Panther Logger.

Note that this code first requests the sonde to take a measurement with the "M" command. After ~15 seconds we can request the first set of data that is associated with this measurement with the "D0" command. This returns four variables, but the first is just the sonde address so we ignore it. We then request the next set of data with the "D1" command, parse the data and then the last part of data with "D2" command, and parse that data. That's not all the data though! We then issue another measurement command of "M1" and we can get three more pieces of data associated with that measurement by issuing "D0" command. It will take about 30 to 45 seconds to get all of the data.