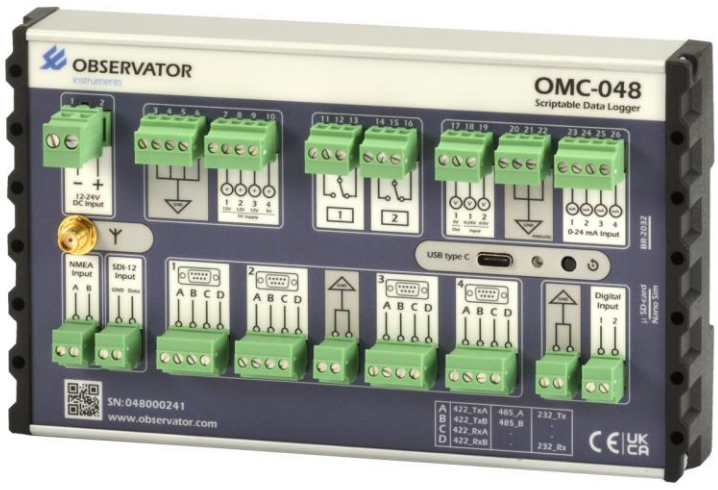


Manual

OMC-048 data logger

Configuration manual

Version: 20250930

Status: Released

Confidentiality: Not Confidential

Date: 30 September 2025

Author: Rob Beun

www.observator.com

Preface

You may wonder what this ‘OMC-048 data logger’ is all about and what makes it so special. I will try to tell you in this document. The focus is on *configuring* the logger for actual use. For more information, please have a look:

- On the product page of our website: OMC-048. Here you will find:
  - the *Data sheet* (pdf);
  - the *Hardware manual* (pdf);
  - this *Configuration manual* (pdf);
  - the ‘Config template’ (Word document);
  - the *Software manual* (an online page).
- On our firmware download page. Here you will find:
  - A zip file containing the latest software and firmware.

This document relates to firmware version FW_1.03B2803

Note:
This document contains many internal references like ‘see section 6.1’ Or ‘as Figure 2 shows’. These are all clickable links. Just click to go there!

Document history

The Observator product range is in continuous development and so specifications may be subject to change without prior notice.

Revision history

Date	Amendments
2022-05-04	Concept version
2022-05-23	For internal review.
2022-06-20	Approved for distribution. Belongs to FW_1.03B2531 and loggers with serial number below 048000201
2022-10-25	Modified for new hardware (SN>048000200) and BL: 1.03B0. Added new drivers and more on remote control. For internal review.
2022-10-25	Released
2025-09-23	Added: Modbus and several related drivers; relay_port; D-record over serial output; control of various wipers; digital input drivers; pulse_driver. For internal review.
2025-09-30	Released

Distribution list

Date	Distribution
2022-05-23	Internal
2022-06-20	External
2022-10-25	Internal
2022-10-25	External
2025-09-23	Internal
2025-09-30	External

Table of contents

1 The logger hardware at a glimpse

The first thing you may notice about the logger (if you don't have one yet, take a look at the picture on the front page): It is beautiful! Next, you may notice that is has a lot of input and output pins, all clearly labelled and grouped together. Figure 1 shows the front panel for loggers with a serial number below 048000200 (top) and for the newer models (bottom).

The meaning of most pins will be clear, so I will just highlight a few:

- The '*VDC Supply*' pins are outputs to power your sensors or whatever you want to connect to the logger. These pins are under software control. We will get to that later.
- The relays are -of course- also under software control.
- The '*NMEA input*' is an optically isolated, RS422 input. This is a 'true' NMEA port, fully up-to-specification. But it can also be used for other protocols using RS422. If you connect pin 28 (B) to ground, you can even use pin 27 (A) for RS232 input!
- Note that the other four serial ports are identical and can independently be used as RS232, RS422 or RS485. On these ports, NMEA and other serial protocols can be used.
- The two 'Digital Input' pins (51 and 52) can be used for pulse counting (e.g. counting the pulses from a Reed relay in a tipping bucket rain gauge, see rain_pulse: driver), or to detect the state (low or high) of the input signal (see input_state: driver).
- There is also an SMA antenna connector. Fortunately! The logger has an internal 'worldwide' modem supporting LTE-M1, NB1 and GPRS. This roughly translates into 4G and 2.5G. Most providers in most countries will support at least one of these standards, but you may want to check. And also make sure your antenna is suitable for the used frequencies. Older 2G antennas may not perform well on 4G frequencies.

The newest models differ a bit from the first batch with serial numbers below 048000200:

- Nano SIM instead of micro SIM.
- 3 x 12V + 1 x 5V supply outputs, instead of 4 x 12V.
- The reset button now gives a soft-reset, instead of a hard reset.
- An internal ADC for monitoring the supply voltage has been added.
- Vref output has changed from 2.5V to 5V.
- The application will only start if you connected external power. With power just from USB, you can configure the logger and work with the REPL, but you cannot start the logger application. On older models the application can run without external power, but you risk strange errors when the internal modem is switched on and there is not enough USB power available.
- Text on the front panel indicates the SD and SIM cards, as well as the coin battery.

A final note about the hardware:
The power outputs are powered from the logger's external power input and not from USB. So you need to apply external power in order to use the power outputs. The voltage on the 12V outputs will not go above 12 V when the supply goes above 12 V, but they will follow the supply voltage when it goes below 12 V.

For more information on the logger hardware, please have a look at the hardware manual on the product page of our website: OMC-048.

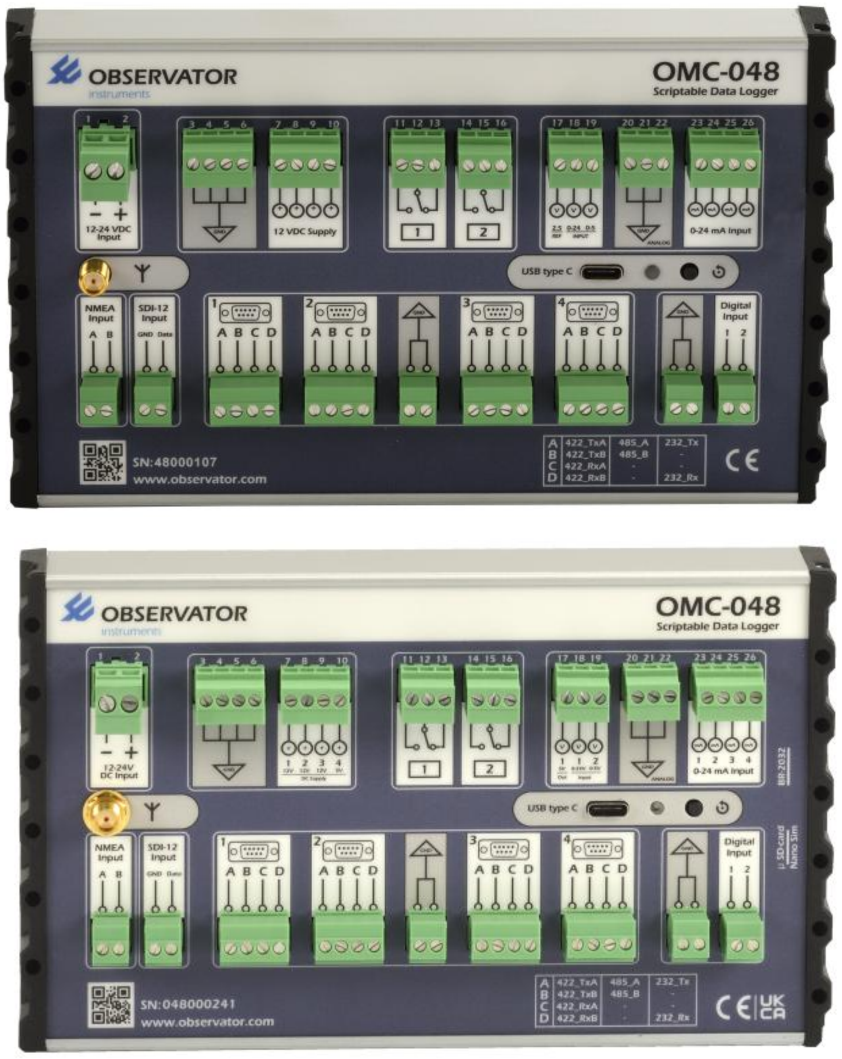


Figure 1. The front panel of the OMC-048, first series on top, second series at the bottom.

2 The logger software at a glimpse.

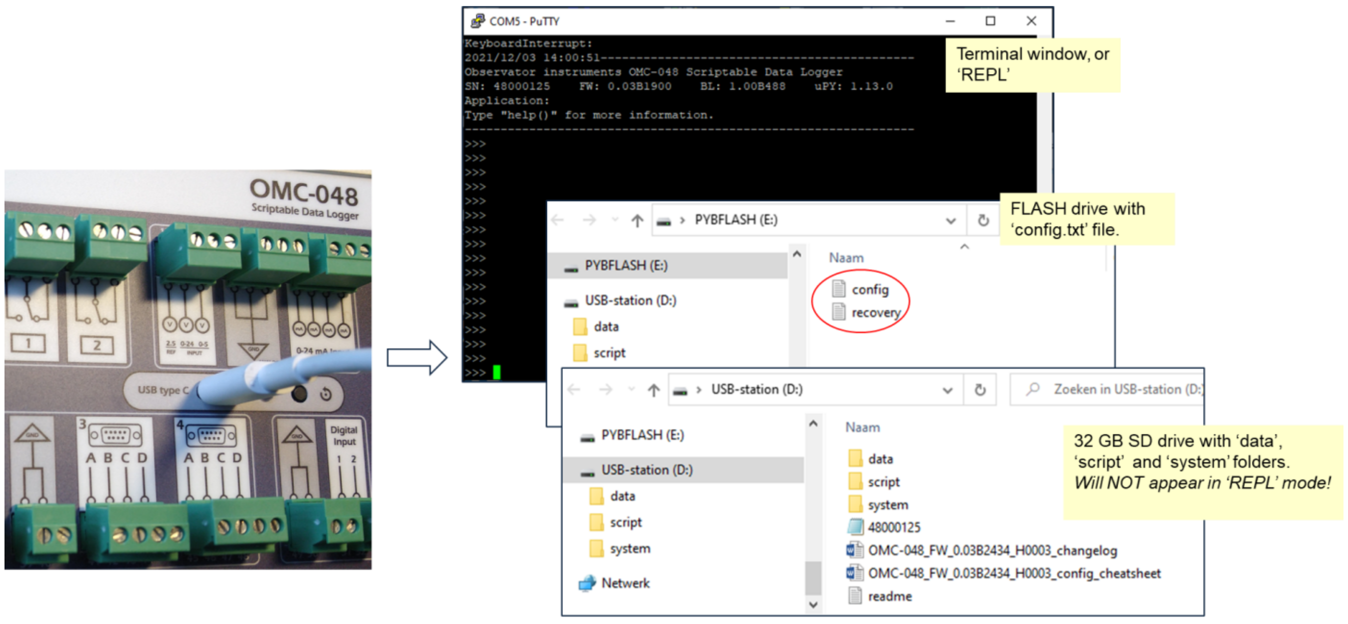
The software is where the OMC-048 really stands out from the crowd. As we all know, putting wires between devices is the easy part; making them all work together is the hard part. The logger and the sensors should ‘understand’ each other. And you want to be able to tell the logger when and how sensor data should be collected and where this data should go to and in which way. How do you tell that to the logger? And what sensors can the logger talk to?

It is important to understand that the OMC-048 was developed to handle complex situations and a wide range of connected devices. We also made sure that we -and you, if you want- can add new drivers and functions. In that sense, the logger will never be finished, but its functionality will keep growing. This flexibility comes at a small price: you will need to take a little effort to understand the basics, before you can appreciate the power of the logger. It may seem a bit complex at first, but once you know how it works, it is more fun than work!

When you connect the logger to your PC using a USB cable, the logger will appear as a mass-storage device. Or, actually, <u>two</u> drives may appear in your file explorer window (see section 3 for the details), plus a COM port:

- A ‘*flash*’ drive, where the configuration file ‘config.txt’ is stored (and some more files, which are not relevant at this moment).
- An ‘*SD*’ drive. Here you can find, among others: files containing the logged data; log files containing system information; and you can find much of the Python code that makes up the software.
- The logger will open a virtual COM port on your PC. Using a terminal program (many are available for free), you can: start/stop the logger; follow ‘live’ what the logger is doing; perform debugging; and more. This is called ‘the REPL’ and will be explained later in detail.

Figure 2 shows how it may look on your screen, after you connect to the logger by USB.



```
KeyboardInterrupt:
2021/12/03 14:00:51---------------------------------------------
Observator instruments OMC-048 Scriptable Data Logger
SN: 48000125    FW: 0.03B1900    BL: 1.00B488    uPY: 1.13.0
Application:
Type "help()" for more information.
----------------------------------------------------------------
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```

OMC-048
Scriptable Data Logger
USB type C
COM5 - PuTTY
Terminal window, or ‘REPL’
PYBFLASH (E:)
USB-station (D:)
data
script
Naam
config
recovery
FLASH drive with ‘config.txt’ file.
USB-station (D:)
Zoeken in USB-station (D:)
PYBFLASH (E:)
USB-station (D:)
data
script
system
Netwerk
Naam
data
script
system
48000125
OMC-048_FW_0.03B2434_H0003_changelog
OMC-048_FW_0.03B2434_H0003_config_cheatsheet
readme
32 GB SD drive with ‘data’, ‘script’ and ‘system’ folders.
Will NOT appear in ‘REPL’ mode!

Figure 2. When the OMC-048 is connected by USB, a COM port connection and one or two disk drives are opened.

If you get error messages from Windows about the USB connection, you may not have sufficient power available on your USB port. In that case, try another USB port (preferably USB-C) or supply external power to the logger. Particularly laptops may suffer from this problem.

Most users will only need to understand the *configuration file* (config.txt) for customizing the logger. However, in some cases you may need to customize a *reference table*, as briefly explained below.

Note:
The *configuration file* (config.txt) is the most important means for you to 'customize' the logger for your application. In this file you can define:

- which sensors are connected to which pins;
- how often these sensors should be sampled;
- where the data should be send to (which modem and which address) and how often;
- many other details.

The logger has several drivers for specific sensors, but what if you have 'another' sensor? One that the logger does not know? Well, the logger has several drivers that can be customized using so called '*reference tables*'. For example, suppose you have a sensor that outputs a particular ASCII string over RS232 or 422. In that case, you need to 'tell' the driver what this string looks like, so the logger knows how to read the string. For this, you need to modify (fill-in) the reference table of the driver. Don't worry if this sounds complex. In chapter 9.3 I will show you some examples that will clarify matters! If your sensor is too special for the drivers using reference tables, you can add your own driver in Python code.

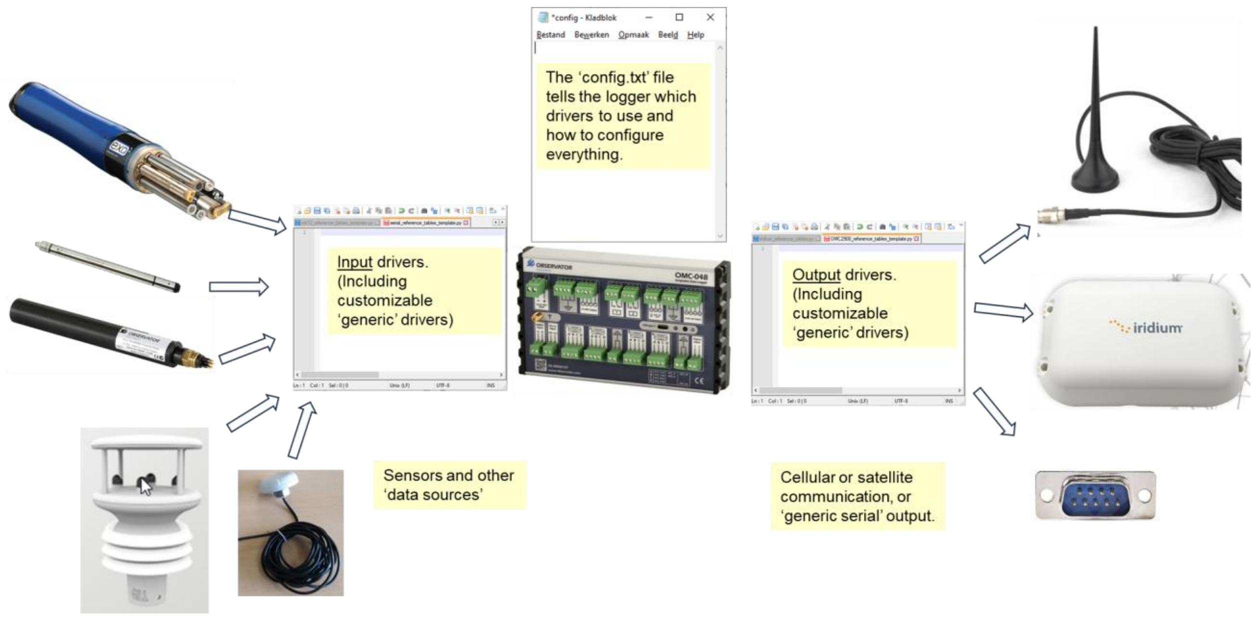

*config - Kladblok
Bestand Bewerken Opmaak Beeld Help
The 'config.txt' file tells the logger which drivers to use and how to configure everything.
Input drivers. (Including customizable 'generic' drivers)
Output drivers. (Including customizable 'generic' drivers)
Sensors and other 'data sources'
Cellular or satellite communication, or 'generic serial' output.
OBSERVATOR
OMC-048
iridium

Figure 3. The 'config.txt' file and Python 'generic' drivers can be modified to control the logger's behaviour.

In addition to getting data <u>into</u> the logger (typically from sensors), you may also want to get data <u>out</u> of the logger. It would not be very useful without that option, right? Well first of all, the logger will store the data on the SD card. There is enough space (32 GB) to last a lifetime for most applications. There are several ways to get the stored data out of the logger:

- You can connect by USB and simply copy the data files to your computer.
- You can use the internal modem and configure the logger to automatically send the data over the cellular network. At this moment the logger supports:
    - FTP in the ‘native’ OMC-045 format. This is for compatibility reasons with OMC-Data-Online. Third party tools can also read these files.
    - Secure (TLS) TCP for communication with Blue2Cast server software. From there, users can import the data in their own system through an API. Or they can use Blue2Cast to directly access the data.
- You can connect the logger to an external Iridium modem and send the data over the Iridium satellite network. Depending on the Iridium provider you choose, you can get the data by mail or through an API or whatever other means the provider offers.
- You can output the data as a serial ASCII stream over one of its four serial ports. It’s up to you how to use this data (maybe connect to the serial input of a LAN port, to an ASCII display, or to a PC running a terminal program or something more sophisticated). You can define your own output format.
- Finally, you can take out the SD card and plug it into your computer or SD card reader. This will even work if the logger itself is defect.

Now we talked about the USB interface, about configuring the logger and about getting data into and out of the logger. What else? Ah yes: What if something goes wrong or you want to change something? Usually happens when the logger is far, far away, right? Well, we have given the logger many handy features to support debugging. And you can do all these things <u>remotely</u> as well as by using a USB cable. Because debugging is so important and because the features are so comprehensive, a whole chapter has been devoted to debugging!

The tool for remote control of the logger is described in chapter 16. For now, it is only relevant to know that you need to add a few lines to the config.txt file to tell the logger to periodically make contact with the server. Once this is done, you can use any computer or mobile device to access the ‘OMC-048 remote control’ page on the Blue2Cast website. On this webpage, you can connect to available loggers and perform basically all actions you can also do with an USB connection. Again, refer to chapter 16 for a more complete description.

3 USB modes

Please look again at Figure 2. There it shows that one or two disk drives will pop-up in your file explorer window. It depends on the 'USB mode' if the SD card of the logger will pop-up or not. The 'FLASH' drive will always pop-up. It can also be called 'PYBFLASH'.

Why are there different USB modes? Well, with only one mode, the problem is that a running logger application will access the SD card at the same time Windows is accessing the SD card. This can cause conflicts. So you have to choose between either Windows or the logger application accessing the SD card. (Or both, with the risk of conflicts) So there are now three USB modes:

1. **Repl** mode means that the logger application can access the SD card, but Windows can't.
2. **Storage** mode means that Windows can access the SD card, but the logger application can't.
3. **Debug** mode means that both can access the SD card, but this may give trouble if you don't know the details. So better not use it.

Refer to below table for the details.

The USB mode is only relevant if a USB cable is connected. Without this connection, the 'USB mode' has no meaning.

The required mode can be selected in the config.txt file:

```
Omc048:
  usb_mode: storage          #debug/storage/repl
```

If you don't know which USB mode to select, select 'repl'. For updating SW you will need to use 'storage'. When you see the SD card in Windows, you know you are in storage (or debug) mode.

USB mode	Meaning	Used when?	(Dis)connecting USB
repl	You have access to the REPL, where you can start/stop the application and watch it running. You (Windows) do not have access to the SD card.	• Normal field use with modem. • Using the REPL to see what the logger is doing. • Most debugging work.	If you disconnect USB while the application is NOT running (e.g. after control-C), the application will start. Otherwise connecting or disconnecting has no side effects.
storage	You have access to the SD card (storage), but the application will NOT run. You can use the REPL, but you cannot start the application.	• Normal field use without modem. • Copying/reading data files and/or system log files. • Software/firmware update. • Working with the REPL without running application.	If you disconnect USB, the application will start. When you reconnect USB, the application will stop.
debug	You can do everything: Start/stop the application in the REPL and access the SD card.	• Debugging, but only when you understand the risks involved.	Connecting or disconnecting USB has no side effects: the logger will not start/stop or change mode.

4 So, where do I start?

Okay, now you have the logger, you have some sensors and -hopefully- you know more or less what you want them to do. How to get to a working system from here? It depends a bit on your starting point.

If your sensors and logger have been prepared & configured by Observator, you are off to a flying start. You will only have to connect everything as instructed, and maybe modify the config file a little for your particular SIM card. Having everything prepared by Observator may be the easiest way, but it is much more fun to do the preparation steps yourself! So that is described below, but keep in mind that you may be able to skip some steps if they have been prepared for you.

Clicking the blue underlined text takes you directly to the right chapter.

Updating the software and firmware
When you start with a new application, I recommend you upgrade to the newest software version. If you just want to make a small change to an existing application that is running well, you don't need to update. Updating software is described in chapter 15.

Configuration Essentials
Anyone wanting to configure the logger should read this chapter 6.

Timing and scheduling
Before you can understand how you can complete the 'config.txt' with the correct timing and scheduling for all sensors and other stuff, you need to understand how timing & scheduling works. A though subject, but chapter 7 takes you through it.

Logging schedule and cellular communication
This chapter is also relevant if you use other communication channels, or just store data on the SD card, because it explains the mechanism for storing data in files. This is chapter 8.

Testing and debugging
In many cases you will also need to understand a bit about testing and debugging. Chapter 14.

Once you made it through the above chapters, you can probably continue by just reading the parts you need.

5 The terminal window (REPL)

5.1 What is the REPL?

The OMC-048 is a small computer with a lot of input-output capabilities. Just like your computer, it can run many different application programs (apps). A computer has an ‘operating system’ (like Windows™ or Android™). The operating systems offers functions that application programs can make use of. For example: If you have a Windows PC, Windows offers functions like a printer driver and internet access.

In the OMC-048, the REPL can be seen as the ‘operating system’. Python is a programming language. It is now time you finally learn what REPL means! It means: *Read, Evaluate, Print* and *Loop*. The REPL allows you to type (Python) commands. The REPL will *Read* what you typed, *Evaluate* (execute) the command, *Print* the result on your screen, and finally *Loop* back to reading the next command you may type. The commands you can type are not just the commands Python understands; Observator also added a lot of OMC-048-specific commands that you can use. Chapter 17 gives more detailed info on this.

Normally the OMC-048 will be running the logger application. That’s why we usually call the OMC-048 a logger. But, as chapter 17 explains, you can also run other application programs that turn the OMC-048 into something else then a logger. Here we limit ourselves to the logger application.

5.2 How to get access to the REPL

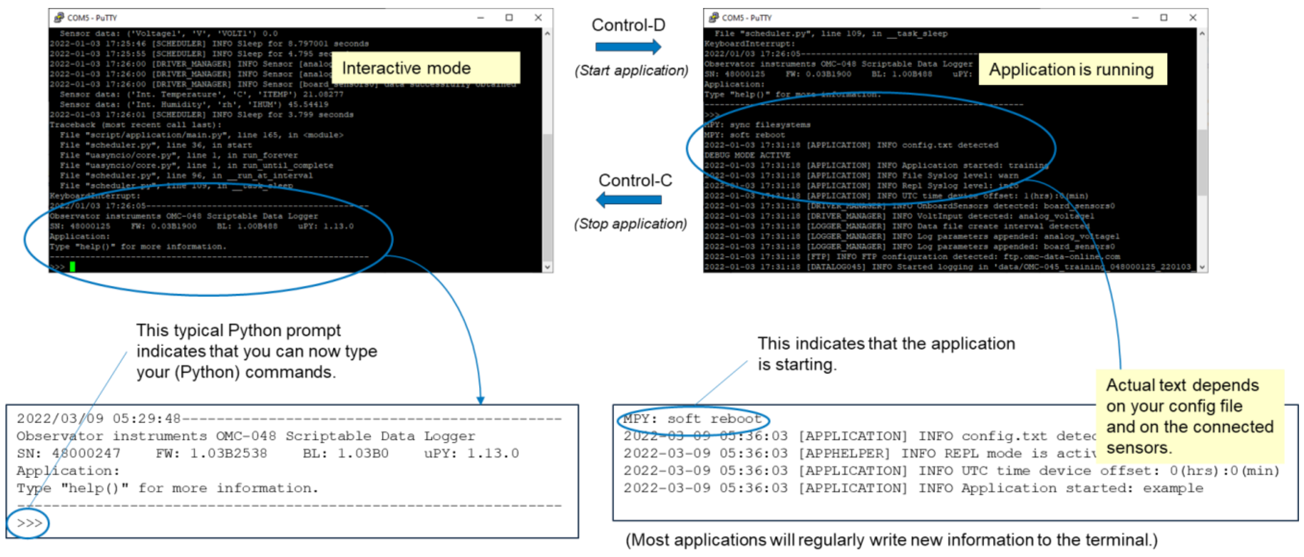
If you connect your PC to the logger by USB, a (virtual) COM port will become available on your PC. You can see this in the Device Manager > Ports. You can use a terminal program of your choice to connect to this port and the REPL will become visible. A few names of the many available terminal programs are: PuTTY, GNU, Windows Terminal and Tera Term. All these programs will allow you to connect to the COM port of the logger, which you can find in the Device Manager.

Using remote control (chapter 16), you can also get access to the REPL.

Once in the REPL, the logger application may either be running or not. If not, the REPL is in *interactive mode*.

Note:
If the logger application is running, you can stop it by typing **control**-C. You can (re-)start a stopped application by typing **control**-D. *(Some terminal programs require you to use capital C and D, so you must press the control and shift keys simultaneously will hitting C or D. Others don’t.)*

Figure 4 shows you what the terminal windows may look like, if you switch between modes in the REPL.


Interactive mode
Control-D
(Start application)
Control-C
(Stop application)
Application is running
This typical Python prompt indicates that you can now type your (Python) commands.
This indicates that the application is starting.
Actual text depends on your config file and on the connected sensors.
(Most applications will regularly write new information to the terminal.)

COM5 - PuTTY

```
  Sensor data: ('Voltagel', 'V', 'VOLT1') 0.0
2022-01-03 17:25:46 [SCHEDULER] INFO Sleep for 8.797001 seconds
2022-01-03 17:25:55 [SCHEDULER] INFO Sleep for 4.795 sec
2022-01-03 17:26:00 [DRIVER_MANAGER] INFO Sensor [analog
2022-01-03 17:26:00 [DRIVER_MANAGER] INFO Sensor [analog
2022-01-03 17:26:00 [DRIVER_MANAGER] INFO Sensor [board_sensors0] data successfully obtained
  Sensor data: ('Int. Temperature', 'C', 'ITEMP') 21.08277
  Sensor data: ('Int. Humidity', 'rh', 'IHUM') 45.54419
2022-01-03 17:26:01 [SCHEDULER] INFO Sleep for 3.799 seconds
Traceback (most recent call last):
  File "script/application/main.py", line 165, in <module>
  File "scheduler.py", line 36, in start
  File "uasyncio/core.py", line 1, in run_forever
  File "uasyncio/core.py", line 1, in run_until_complete
  File "scheduler.py", line 96, in __run_at_interval
  File "scheduler.py", line 109, in __task_sleep
KeyboardInterrupt:
2022/01/03 17:26:05--------------------------------------------
Observator instruments OMC-048 Scriptable Data Logger
SN: 48000125    FW: 0.03B1900    BL: 1.00B488    uPY: 1.13.0
Application:
Type "help()" for more information.
---------------------------------------------------------------
>>> 
```

COM5 - PuTTY

```
  File "scheduler.py", line 109, in __task_sleep
KeyboardInterrupt:
2022/01/03 17:26:05--------------------------------------------
Observator instruments OMC-048 Scriptable Data Logger
SN: 48000125    FW: 0.03B1900    BL: 1.00B488    uPY:
Application:
Type "help()" for more information.
---------------------------------------------------------------
>>>
MPY: sync filesystems
MPY: soft reboot
2022-01-03 17:31:18 [APPLICATION] INFO config.txt detected
DEBUG MODE ACTIVE
2022-01-03 17:31:18 [APPLICATION] INFO Application started: training
2022-01-03 17:31:18 [APPLICATION] INFO File Syslog level: warn
2022-01-03 17:31:18 [APPLICATION] INFO Repl Syslog level: info
2022-01-03 17:31:18 [APPLICATION] INFO UTC time device offset: 1(hrs):0(min)
2022-01-03 17:31:18 [DRIVER_MANAGER] INFO OnboardSensors detected: board_sensors0
2022-01-03 17:31:18 [DRIVER_MANAGER] INFO VoltInput detected: analog_voltagel
2022-01-03 17:31:18 [LOGGER_MANAGER] INFO Data file create interval detected
2022-01-03 17:31:18 [LOGGER_MANAGER] INFO Log parameters appended: analog_voltagel
2022-01-03 17:31:18 [LOGGER_MANAGER] INFO Log parameters appended: board_sensors0
2022-01-03 17:31:18 [FTP] INFO FTP configuration detected: ftp.omc-data-online.com
2022-01-03 17:31:18 [DATALOG045] INFO Started logging in 'data/OMC-045_training_048000125_220103_
```

```
2022/03/09 05:29:48---------------------------------------------
Observator instruments OMC-048 Scriptable Data Logger
SN: 48000247    FW: 1.03B2538    BL: 1.03B0    uPY: 1.13.0
Application:
Type "help()" for more information.
----------------------------------------------------------------
>>>
```

```
MPY: soft reboot
2022-03-09 05:36:03 [APPLICATION] INFO config.txt detec
2022-03-09 05:36:03 [APPHELPER] INFO REPL mode is activ
2022-03-09 05:36:03 [APPLICATION] INFO UTC time device offset: 0(hrs):0(min)
2022-03-09 05:36:03 [APPLICATION] INFO Application started: example
```

Figure 4.   In the REPL, go from ‘Interactive mode’ to ‘application is running’ and back.

6 Configuration Essentials

This chapter is essential for anyone who wants to be able to configure the logger for a particular application. Look at the title, if you don't believe me.

Section 6.1 explains the 'config.txt' file. In the config file will specify drivers for sensors and other devices. Various drivers use common settings, which are explained in section 6.2. In section 6.3 the use of the 'config template' is explained. This allows you to quickly have a good starting point for your configuration file.

6.1 The outline and principles of the configuration file

The config.txt file is located on the FLASH drive that will pop-up whenever you connect the logger by USB. Once you have customized this file for your application, it makes sense to have a copy of this file on your computer or -even better- at some disk that you share with your colleagues. So everyone can see what this logger is doing and can use the file as a starting point for other loggers (maybe by simply copying it).

You can edit the confix.txt with a simple text editor like Notepad of Notepad++. The format of the file is called 'yaml'. If your editor understands yaml (like Notepad++ does), you can use advantages like syntax coloring and maybe even syntax checking, depending on what your editor offers. In Notepad++, select 'Syntaxis > yaml'.

Notes:

- A language-sensitive editor can help you prevent making mistakes.
- In Notepad++, select 'Syntaxis > yaml' to edit the config.txt file
- Before you change a working config.txt, it is very wise to make sure you have a copy!

The logger will try to find a file called 'config.txt' each time it is started. So you can make copies with names like 'config_john.txt' or 'config_fast.txt' on the flash. They will be ignored by the logger.

Notes:

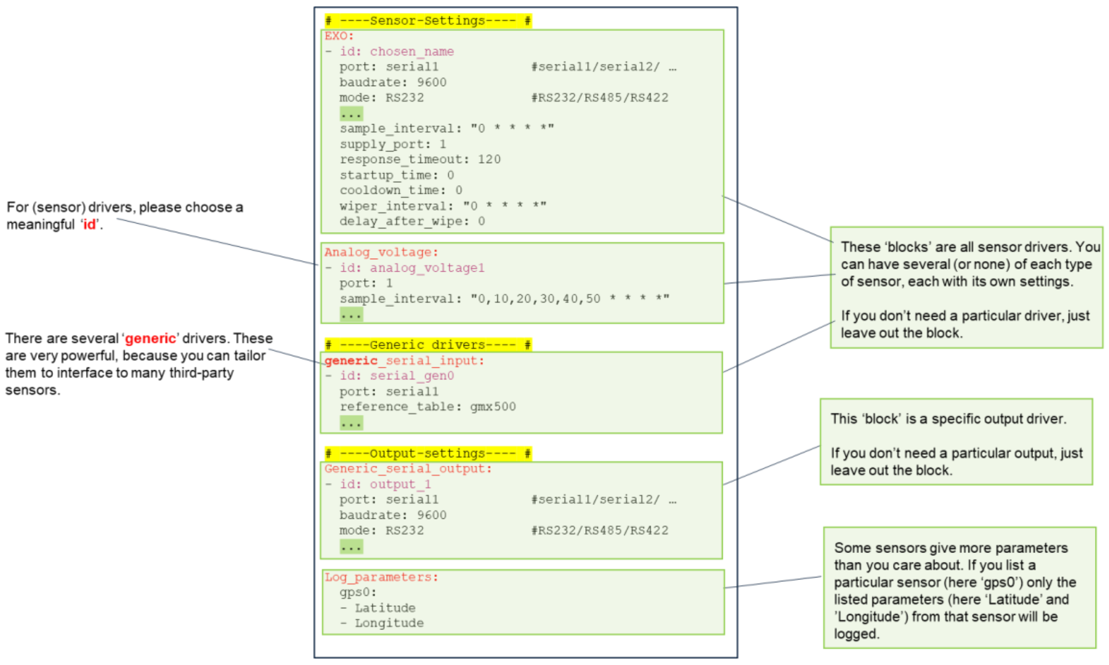
- The file format is sensitive to spaces and does not use tab's, so make sure you have the right number of spaces everywhere.
- Keywords (like *Omc048, info, False, id, port* and so on) are case sensitive.
- Every '-' and ':' has a meaning, so these should be in the correct place.
- Don't use special characters like '+' and '-' in names that you choose.

If you get any of these wrong, something will go wrong.

Figure 5 and Figure 6 show you what a configuration file looks like. Note that these figures do not show a complete and meaningful configuration file. They are only intended to show you some important parts of the file and the general way it is constructed. You can use the 'config template' file on our website for making your own config.txt file. See section 6.3

It is recommended to start any new project from the 'config_template' file (section 6.3). This prevents cluttering your config file with old or meaningless text from previous versions or projects. So please don't use 'old' projects as a starting point for a new project.

The file consists of text ‘blocks’ containing ‘name: value’-pairs. You can mix the order of these text ‘blocks’. You need to include only the blocks that you need.



```
# ----Sensor-Settings---- #
EXO:
- id: chosen_name
  port: serial1             #serial1/serial2/ …
  baudrate: 9600
  mode: RS232               #RS232/RS485/RS422
  ...
  sample_interval: "0 * * * *"
  supply_port: 1
  response_timeout: 120
  startup_time: 0
  cooldown_time: 0
  wiper_interval: "0 * * * *"
  delay_after_wipe: 0

Analog_voltage:
- id: analog_voltage1
  port: 1
  sample_interval: "0,10,20,30,40,50 * * * *"
  ...

# ----Generic drivers---- #
generic_serial_input:
- id: serial_gen0
  port: serial1
  reference_table: gmx500
  ...

# ----Output-settings---- #
Generic_serial_output:
- id: output_1
  port: serial1             #serial1/serial2/ …
  baudrate: 9600
  mode: RS232               #RS232/RS485/RS422
  ...

Log_parameters:
  gps0:
  - Latitude
  - Longitude
```

For (sensor) drivers, please choose a meaningful ‘id’.

There are several ‘generic’ drivers. These are very powerful, because you can tailor them to interface to many third-party sensors.

These ‘blocks’ are all sensor drivers. You can have several (or none) of each type of sensor, each with its own settings.

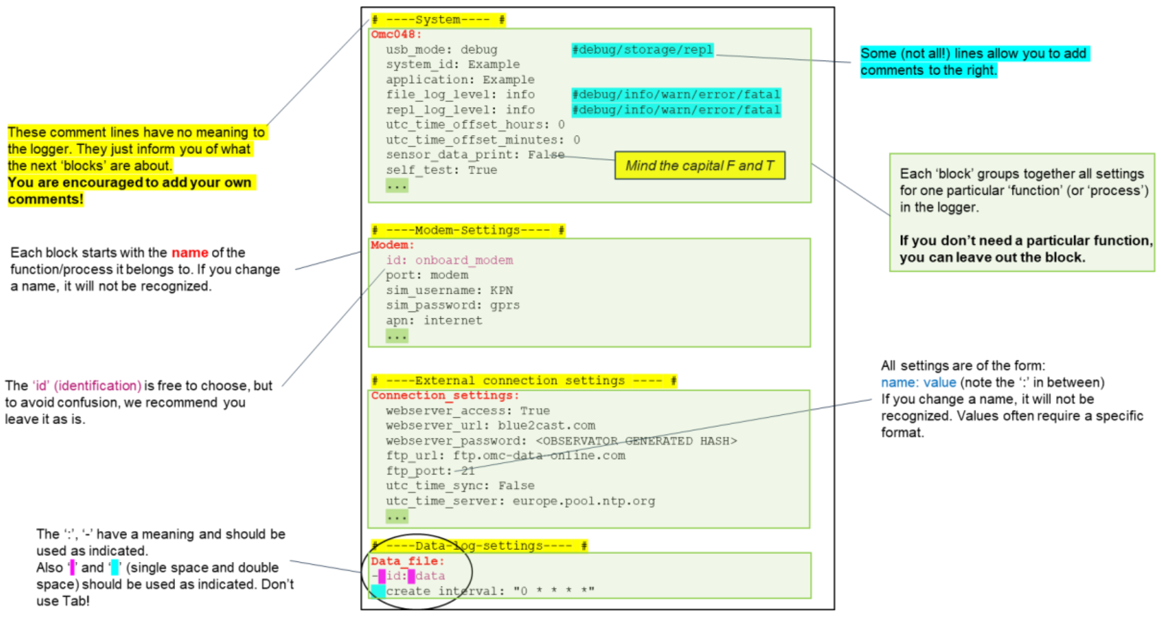
If you don’t need a particular driver, just leave out the block.

This ‘block’ is a specific output driver.

If you don’t need a particular output, just leave out the block.

Some sensors give more parameters than you care about. If you list a particular sensor (here ‘gps0’) only the listed parameters (here ‘Latitude’ and ’Longitude’) from that sensor will be logged.

Figure 5. Outline of the configuration file, part 1 (this is not a complete example!).



```
# ----System---- #
Omc048:
  usb_mode: debug             #debug/storage/repl
  system_id: Example
  application: Example
  file_log_level: info        #debug/info/warn/error/fatal
  repl_log_level: info        #debug/info/warn/error/fatal
  utc_time_offset_hours: 0
  utc_time_offset_minutes: 0
  sensor_data_print: False
  self_test: True
  ...

# ----Modem-Settings---- #
Modem:
  id: onboard_modem
  port: modem
  sim_username: KPN
  sim_password: gprs
  apn: internet
  ...

# ----External connection settings ---- #
Connection_settings:
  webserver_access: True
  webserver_url: blue2cast.com
  webserver_password: <OBSERVATOR GENERATED HASH>
  ftp_url: ftp.omc-data-online.com
  ftp_port: 21
  utc_time_sync: False
  utc_time_server: europe.pool.ntp.org
  ...

# ----Data-log-settings---- #
Data_file:
- id: data
  create interval: "0 * * * *"
```

Mind the capital F and T

These comment lines have no meaning to the logger. They just inform you of what the next ‘blocks’ are about.
You are encouraged to add your own comments!

Each block starts with the **name** of the function/process it belongs to. If you change a name, it will not be recognized.

The ‘id’ (identification) is free to choose, but to avoid confusion, we recommend you leave it as is.

The ‘:’, ‘-’ have a meaning and should be used as indicated.
Also ‘ ’ and ‘ ’ (single space and double space) should be used as indicated. Don’t use Tab!

Some (not all!) lines allow you to add comments to the right.

Each ‘block’ groups together all settings for one particular ‘function’ (or ‘process’) in the logger.

If you don’t need a particular function, you can leave out the block.

All settings are of the form:
name: value (note the ‘:’ in between)
If you change a name, it will not be recognized. Values often require a specific format.

Figure 6. Outline of the configuration file, part 2 (this is not a complete example!).

As mentioned before, the config.txt is stored on the flash drive of the OMC-048 and this drive is always available when you plug-in the USB cable, independent of the USB mode.

Each time the logger application starts (either after power-on or after Control-D), it will first read the config.txt file.

6.2 Common settings for drivers

Before describing all blocks of the config file in detail, it is good to first explain a few settings that are used in most drivers. Below a hypothetical driver is shown that contains almost all possible settings. A real driver will only need a small subset of these settings!

```
# ----Sensor-Settings---- #
<Driver_name>:                Replace <Driver name> with one of the available driver names
- id: S1                      Give a unique id to each sensor. Unique over the whole config file!
  port: serial1               Define the port of this sensor
  // Timing                   Timing is explained in chapter 7
  Sample_interval: "0 0  * * *"
  response_timeout: 120
  startup_time: 0
  cooldown_time: 0
  cron_offset: False
  // Power/switching
  supply_port: 1              Define the power port for this sensor, if used
  supply_port_always_on: False If 'False', the power will be switched on/off as defined
  relay_port: 2               Define the relay port for this sensor, if used
  relay_port_always_on: False If 'False', the relay will be switched on/off as defined
  // Averaging
  average_interval: "30 * * * *"      Averaging at the specified time points
  moving_average_window: 6            Moving average window
  minimum_required_samples: 30        For (moving) averaging.

- id: S2                      Multiple sensors (unique id!) can use the same driver
  port: serial2               Define the port of this sensor
 … <left out some text here>
```

6.2.1 One driver, multiple sensors (id's)

Note that in this example, there are two sensors ('S1' and 'S2') that use the same driver. For example, if the '*<Driver_name>*' is *EXO*, both sensors should be EXO sondes. If the '*<Driver_name>*' is *Analog_current*, both sensors should be sensors with an analog current output.

6.2.2 Serial ports

There are many sensors (and their corresponding drivers) that use a serial port. Whether it is RS232, NMEA, SDI-12, Modbus or whatever serial protocol. These drivers all share some common properties. The name of the driver may be 'EXO:' or 'navilock_md6' or 'generic_nmea' or one of the many other available drivers, but they all share the below settings that you can modify.

```
  port: serial1       #serial1/serial2/serial3/serial4/nmea/sdi12
  baudrate: 9600
  mode: RS232         #RS232/RS485/RS422
  bits: 8
  parity: None        #0/1/None (0=even, 1 =odd, None = no parity bit)
  stop: 1
  rxbuf: 128
  txbuf: 128
```

The settings ‘rxbuf’ and ‘txbuf’ determine the sizes of the input and output buffers. If you are communicating with a sensor that may produce large chunks of data (more than 128 bytes) it is recommended to increase the size of the buffer, such that it can hold the complete chunk.

6.2.3 Analog ports

There are several analog voltage and current ports. The “port: ” should simply indicate the number of the port you want to use. For example, if the driver name is ‘Analog_voltage’, “port: 1” is the analog voltage input port 1 (pin 18). If the driver name is ‘Analog_current’, “port: 1” is the analog current input port 1 (pin 23).

It is recommended to avoid special characters (like ‘%’) in names like log_name, log_unit, log_tag, but if you have to use one, put it between single or double quotes. For example:

```
  log_unit: "%"
```

6.2.4 Supply_ports

The logger has 4 supply outputs (or ‘power ports’) that can be used to power sensors or other peripherals. These outputs can be controlled from the various drivers, by including the following lines into the driver:

```
  supply_port: 1
  supply_port_always_on: False
```

With ‘supply_port’ you indicate the number (1 to 4 for pins 7 to 10) of the power output you want to use. If you set supply_port_always_on to True, the port will, as you could have guessed, always be ‘on’, once you started the application. If set to False, the port will be switched on and off as determined by the driver. The timing of this switching is explained in chapter 7.

Nothing stops you from using *Power Port 1* for *serial3* and *Power Port 3* for *serial2*, but you can do yourself a favor by making a more logical choice 😉.

6.2.5 Relay_ports

The logger has 2 relay ports. The ‘off’ position is the one that is drawn on the front panel of the logger. The relay port operates the same as the supply_port described above. It shares the response_timeout, startup_time and cooldown_time with the supply_port. So you can use either the supply_port or the relay_port or both. They will use the same timing.

Note that the relays are bi-stable. This means that they will remain in their last position after power is removed. However, during a power-down the logger application will attempt to switch the relay ‘off’. If the supply voltage drops too fast, the microprocessor may not have time to accomplish this and the relay may remain in the ‘on’ position. If you interrupt the application with control-C (REPL in interactive mode), the relay retains its last position. Also, if you are running your own Python code (and not the logger application), the relays will also retain their position in case of a power failure.

6.2.6 Timing

These settings determine the sample_interval, the power on/off timing of the supply_port and/or the relay_port in relation to the communication with the sensor. Timing is explained in chapter 7.

6.2.7 Moving_average

As the example below shows, a moving average filter can be added to any input driver, by adding two settings : `moving_average_window` and `minimum_required_samples`. See examples.

```
Analog_current:
- id: AC1
```

```
port: 1
sample_interval: "0,10,20,30,40,50 * * * *"
…… <some lines left out>
moving_average_window: 6
minimum_required_samples: 4
```

In this example, the stored (logged) values are the average of the last 6 samples. If, for some reason, not all 6 samples have been received, the average is calculated over the samples that did arrive. But if the minimum number of required samples (4) has not been received, the average is not valid, and the value 'None' is logged. Every 10 seconds a sample is taken, and every 10 seconds a value is logged.

Note: Calculating a (moving) average requires storing the samples in the working memory of the processor. This memory space is limited. If you have a lot of parameters, and define large averaging windows, you may run into memory allocation errors.

6.2.8 Averaging

The data-items of all sensors can be averaged over a given average_interval. The *minimum_required_samples* setting gives the minimum count of samples whereby the data is considered to be valid. If not, the value 'None' is logged. Note that only the average is logged. The original values are not logged. The average is calculated at, and gets the time stamp of, the time points specified in average_interval. In below example the sensor is sampled every second, the average is calculated every 30 seconds and is required to include at least 30 samples. The result will be that every 30 seconds a value is logged. Note that this way of averaging reduces the amount of samples that is logged, in contrast to using moving_average.

```
sample_interval: "* * * * *"
average_interval: "30 * * * *"
minimum_required_samples: 30
```

Note: Calculating a (moving) average requires storing the samples in the working memory of the processor. This memory space is limited. If you have a lot of parameters, and define averaging intervals containing a large number of samples, you may run into memory allocation errors.

6.3 Using the 'config template' to build your own config.txt file

The 'config template' can be downloaded from the product page of the OMC-048 on our website. It is intended to help you create a new configuration file for your new project.

In many cases you may have received the logger with a configuration file for a specific application (if you ordered it that way from Observator). In that case, you don't need to make a new configuration file, and you may just want to make some minor modifications.

If you want to start a new project, it is NOT wise to take some old config.txt file as a starting point for making modification. If you do so, you may drag along a lot of lines that are either not needed, or are doing things that you don't understand or don't want. It also makes the file unintelligible.

Start a new project from the 'configuration template' file and NOT from an old *config.txt* file.

The 'configuration template' file is a word document with on the left side code that you can copy to your new config file. On the right side, some brief explanation of the code on the left is given. The procedure is the following:

Start with an empty text document. You can do this is Notepad, but I recommend using a language sensitive editor like Notepad++, that understands the YAML format (with Notepad++, you need to select ‘Syntaxis>YAML)’.

Next, go through the ‘configuration template’ document from beginning to end, and copy the text blocks that you need from top to bottom in your config.txt file.

Finally, go through your config file again, and fill-in or modify the text to your requirements. You will need to fill-in the correct input ports, power ports, baud rates, names, id’s and so on. And, one of the most important thing to fill-in: You need to specify at what times specific actions (like taking a sample or transmitting it) need to be performed. That is the subject of the next chapter. The comments in the template file may be enough to help you once you are an experienced user; otherwise you may need to refer to this configuration manual.

7 Timing and scheduling

As explained in paragraph 6.2, drivers for sensors (or other peripherals) typically include settings for the timing of the sensor. Often a sensor will be powered from one of the switched power outputs of the logger. In that case, the switching of the power should be timed correctly. Paragraph 6.2 already mentioned the settings related to sensor timing; this chapter will explain the timing of the complete logger including the sensor timing.

7.1 Logger internal time

The logger has an internal clock that keeps running on the internal battery coin cell, even if you disconnect external power. If you receive a logger brand-new from the factory, the clock may never have been set. You can either set the clock according to the instructions in the manual, or wait for it to automatically synchronize (if enabled, see below). Note that, if you don't use any time synchronization mechanism, the internal clock will inevitably slowly drift away from the correct time.

The internal clock is used to time all events -like when to take a sample and when to transmit a file- and determines the time stamps that the data will get.

The offset with respect to UTC can be set in the 'Omc048' block with these self-explaining lines:
utc_time_offset_hours: 0
utc_time_offset_minutes: 0

You are free to choose your time zone, but note that this can become confusing if you have loggers working in different time zones and sending data to servers in again different time zones with users watching the data in yet again different time zones! Also, some counties switch time back and forth because of 'Daylight saving time' (or 'Summer Time'). This all makes 'time' a rather relative concept (google 'Einstein' if you want to know more). So, if you are working in multiple time zones, take care of how you handle 'time'.

If you are using the OMC-048 in combination with Observator's Blue2Cast software, the logger MUST be set to UTC. Setting all loggers to UTC assures that all data, from all loggers all over the world, is saved in the Blue2Cast data base in UTC. Anyone logging into Blue2Cast (from any place in the world) can choose between seeing the data time-stamped in the original (UTC) time, or in his local time zone. So the graphs and tables will use the local time, if you selected this in Blue2Cast.

7.1.1 Manual synchronization using the REPL

If you are unable to use automatic synchronization, or you don't want to wait for it to occur, you can manually set the clock. In the REPL, hit control-C to get into interactive mode.

Now type:

```
import omc048
rtc = omc048.RTC()
rtc.datetime()
```

The response of logger will be something like this (example):
(2020, 8, 4, 2, 16, 23, 15, 103)

The format is: (Year, Month, Day, Weekday*, Hour, Minute, Second, Sub second)
* Weekday: 1 = Monday – 7 = Sunday

Now enter the correct date & time, in this example Thursday (weekday 4), the 11th of August 2022 16:29:00. Type:

```
rtc.datetime((2022,8,11,4,16,29,0,0))
```

Mind the double () !
Also mind the last ‘0’, which indicates sub-seconds. Hit enter when the clock ticks 16:29:00 (in this example).

To check if the update went okay type:

```
rtc.datetime()
```

The response of the logger:
(2022, 8, 11, 4, 16, 29, 3, 217)
If this is the correct date and time you’re done. Otherwise enter the correct date and time again.

Once finished, you can restart the logger application by typing control-D.

Note: Observator offers a small app for your Windows PC, that can be used to easily set the time of the logger to your PC time. Ask your sales representative for it!

7.1.2 Automatic time synchronization

Automatic time synchronization of the internal clock can be enabled in the config file:

- In the ‘Connection_settings’ block. This only possible when using the internal cellular modem. Synchronization will be dome using an NTP server. See section 10.4
- In specific drivers for sensors that include GNSS (GPS). Note that, when testing inside a building, the GNSS signal may not be available and synchronization will not occur. Examples of drivers for sensors including GNSS are: the **Navilock:** (or **GPS:**) driver in section 11.4 or the MaxiMet **GMX_501**: driver in section 11.5.

It may take a while before synchronization occurs.

7.2 Time specifications

The timing of all logger actions is determined by *time specifications* in the config.txt file. These specifications look like this: `sample_interval: “0 * * * *”` (example only).

The format of a time specification is:
`“s m h d w”` meaning: On second s of minute m of hour h of day d (1-31) of weekday w (1=Monday)

The wildcard ‘*’ can be used to indicate ‘every’. Thus
`“* * * * *”` means: On every second of every minute of every hour of every day of every weekday

A few examples:

Time specification	Reads as	Time series
0 * * * *	On second 0 of every minute of every hour of every etc.	00:00:00, 00:01:00, 00:02:00, …
10 0,10,20,30,40,50 * * *	On second 10 of minute 0,10,20,30,40 and 50 of every hour of every etc.	00:00:10, 00:10:10, 00:20:10, ….
0 1 * * *	On second 0 of minute 1 of every hour of every etc.	00:01:00, 01:01:00, 02:01:00, …
55 59 * * *	On second 55 of minute 59 of every hour etc.	00:59:55, 01:59:55, 02:59:55, …

7.3 Sensor timing

The most important part of the sensor timing, is the sample_interval. As described in the previous section, sample_interval : “0 * * * *” means that the sensor will be sampled every minute, on the minute (second zero of every minute). Actually, this is only true if the sensor is fast, as described in the next sub-section. The other cases (sensors that may take some time to respond) are described in the subsequent sub-sections.

Note: This section assumes that you use the power ports (*supply_port*) to power your sensors, but you could just as well use the relays (*relay_port*).

7.3.1 Fast sensors

The timing of a sensor is determined in the config file. It its simplest form, a single timing specification like this is all that is needed:

- `sample_interval: "0 * * * *"` (second 0 of every minute).
- `sample_interval: "5 0,30 * * *"` (second 5 of minute 0 and 30 of every hour).

This simple form will do for sensors that are always ‘on’ and ready to immediately produce data. Examples of this are the *internal sensors* (internal temperature, humidity and the coin cell voltage). Also the *analog inputs* will immediately give a value.

7.3.2 Sensor with switched power and startup-time

As described in paragraph 6.2, the logger can also provide power to sensors (pins 7 to 10). Guess where the timing of that is determined? Right! The timing specification for a sensor that is powered by the logger could look like this:

```
sample_interval: "55 * * * *"
startup_time: 5
```

This means that the power is switched on at the moments determined by ‘sample_interval’ (thus 5 seconds before every minute in this example). After the ‘startup_time’ has expired (in this case 5 s), the logger will read the sensor value. So this will happen exactly on the minute (but see next paragraph). Immediately after reading the data, the power is switched off. You could use these settings if you have an analog sensor that needs a few seconds after power-on before it outputs a stable value.

7.3.3 Sensor with delayed response

Most digital sensors, like SDI-12 of ModBus, will have an unknown delay between the moment you issue the ‘read data’ command (or whatever command that particular sensor needs) and the moment you receive the data. So you can only determine when to ask for the data, but you don’t know when you will get the data. For example, let’s assume a digital sensor takes 7 seconds to respond and is always powered. In this case, your timing specification could look like this:

```
sample_interval: "53 * * * *"
response_timeout: 10
```

Now, if the sensor takes indeed exactly 7 seconds to respond, data will be sampled exactly on every whole minute. But if the delay of the sensor varies a few seconds, the actual sample time will also vary. The logger will wait for a maximum of ‘response_timeout’ seconds for data to arrive. If data arrives in time, it is accepted with the time_stamp of that particular moment. If data does not arrive in time, the sample is skipped.

If a power switch was used to power this sensor, the power would be switched of immediately after reading the sample, or when the response_timeout expires (whichever happens first). Except if a cooldown_time is specified as well.

7.3.4 Sensor with a demanding attitude

Suppose that you are switching the power to a sensor with the following properties:

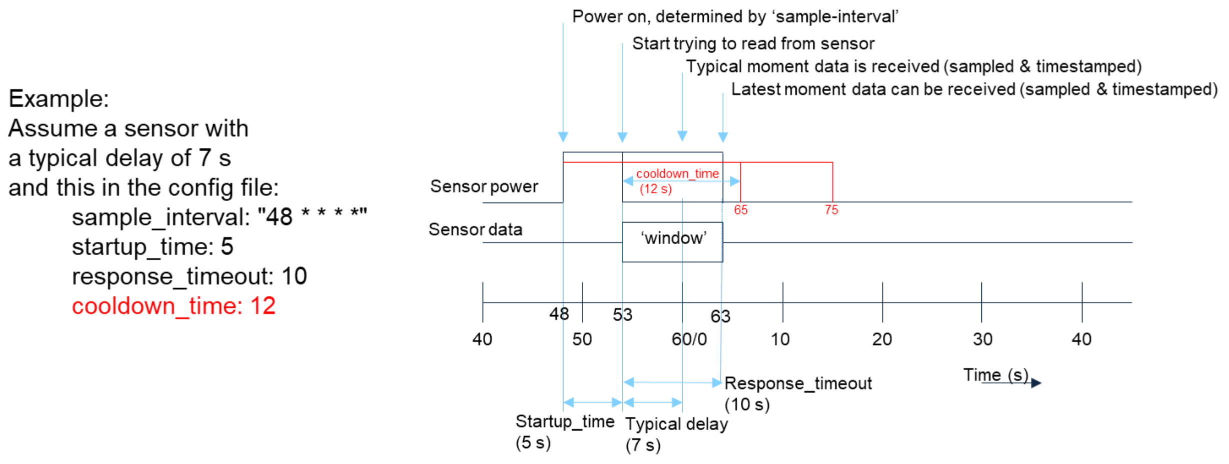
- After power-on, it needs 5 seconds before you can send commands (like ‘read data’) to it.
- It has a delayed response, as described in the preceding section, of typically 7 seconds, but always less than 10 seconds.
- After reading the data, the sensor needs 12 seconds to nicely shut down before you can safely switch off the power.

Now the timing specification could look like this:

```
sample_interval: "48 * * * *"
startup_time: 5
response_timeout: 10
cooldown_time: 12
```

The timing diagram is shown in Figure 7. As you can see, there is a ‘window’ in which the sensor will respond and the power will be switched off. If the sensor would always respond in exactly 7 seconds, the ‘window’ would close exactly on the minute (second 60/0).

The cooldown_time simply delays the power-off by 12 seconds from wherever it would occur. So, at the latest on 63+12=75 (= 15) seconds.



Example:
Assume a sensor with
a typical delay of 7 s
and this in the config file:
sample_interval: "48 * * * *"
startup_time: 5
response_timeout: 10
cooldown_time: 12
Power on, determined by ‘sample-interval’
Start trying to read from sensor
Typical moment data is received (sampled & timestamped)
Latest moment data can be received (sampled & timestamped)
Sensor power
cooldown_time
(12 s)
65
75
Sensor data
‘window’
48
53
63
40
50
60/0
10
20
30
40
Time (s)
Response_timeout
(10 s)
Startup_time
(5 s)
Typical delay
(7 s)

Figure 7. Timing of a sensor with a startup_time and a delayed response (in red also a cooldown_time).

In this relatively complex sensor timing example, the duration of the whole sample process -without cooldown_time- is at most 15 seconds (the sum of startup_time and response_timeout). With cooldown-time, the whole thing takes at most 15+12=27 seconds. So the sample interval should be at least this value, plus some margin. The time stamps of the sample will be between second 53 and 63 (second 3 of the next minute)

Note:
the time stamp that will be put on the data is determined by the actual moment that the data is received (sampled). If the delay of the sensor can vary, you cannot control this moment exactly. So if you would like your samples to have a time stamp exactly on the whole minute (for example), actual time stamps may be a few seconds before or after the whole minute, if your sensor delay varies a few seconds. See section 7.3.5 for a solution.

7.3.5 Cron_offset: Creating 'nice' (rounded) time stamps

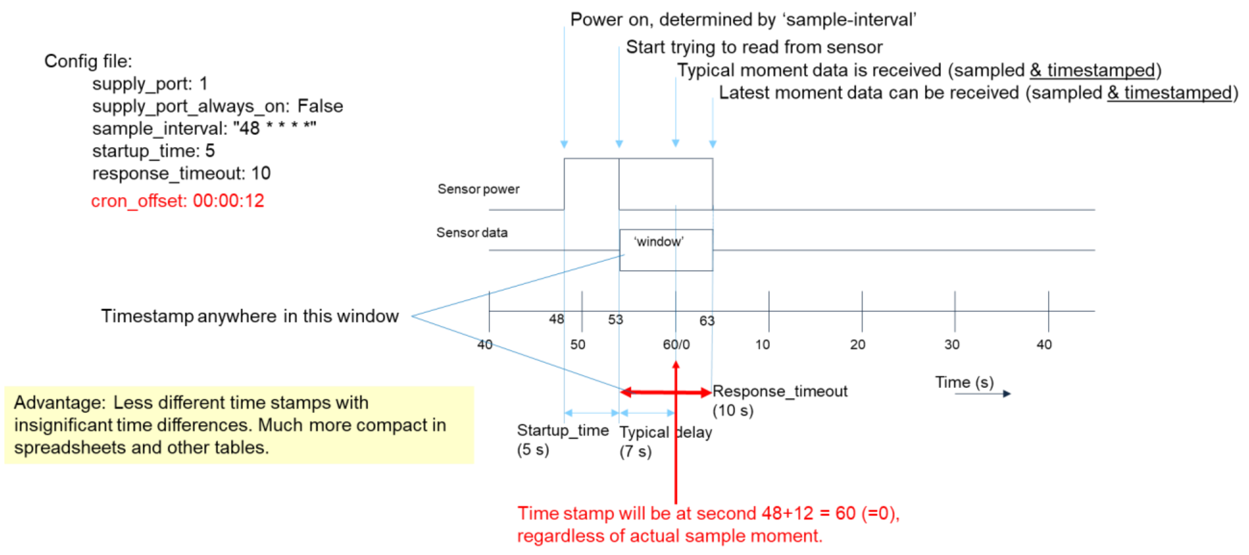
Depending on your sensors, you may be able to get each sample time-stamped at exactly the moment you want. For example, exactly at second 0 of minute 0, 10, 20 and so on. But, as we saw in the preceding sections, sensors with variable delay will inevitably lead to time stamps 'around' the nominal value. This is not a problem, but is a true reflection of what happened.

However, if the data is collected in a spreadsheet, it may be annoying to have multiple lines with time stamps that differ only by an insignificant amount. For example, instead of having one line per 10 minutes with all sensor values, you may have several lines with time stamps that differ just a single second and each contain just a single value.

By using 'cron_offset', you can control the value of the time stamp, regardless of the delay of the sensor. Figure 8 is a copy of Figure 7, but with cron_offset added. The figure speaks for itself! The format of cron_offset is hh:mm:ss. Thus 00:10:20 results in a time stamp with an offset of 10 minutes and 20 seconds from the scheduled sample moment (sample_interval). In the example in Figure 8, the sample_interval at second 48 is offset by 12 seconds, resulting in a timestamp at whole minutes.

Include:

cron_offset: 00:00:12	to add 12 s to the time stamps given by 'sample_interval'.
cron_offset: False	if you explicitly don't want to use it. You can also leave out the whole line.



```
Config file:
        supply_port: 1
        supply_port_always_on: False
        sample_interval: "48 * * * *"
        startup_time: 5
        response_timeout: 10
        cron_offset: 00:00:12
```

Advantage: Less different time stamps with insignificant time differences. Much more compact in spreadsheets and other tables.

Time stamp will be at second 48+12 = 60 (=0), regardless of actual sample moment.

Power on, determined by 'sample-interval'
Start trying to read from sensor
Typical moment data is received (sampled & timestamped)
Latest moment data can be received (sampled & timestamped)
Sensor power
Sensor data
'window'
Timestamp anywhere in this window
48
53
63
40
50
60/0
10
20
30
40
Time (s)
Response_timeout (10 s)
Startup_time (5 s)
Typical delay (7 s)

Figure 8. With 'cron_offset' you can control the values of the time stamps, regardless of sensor delay.

8 Logging schedule and cellular communication

This chapter is also relevant if you don’t use cellular transmission. It describes how data is stored in files on the SD card. If you use cellular transmission, these same files are transmitted. If you don’t use cellular transmission, the files simply stay on the SD card. The use of Iridium, or data output over serial ports, has no effect on these files.

If you are not interested in the use of the internal cellular modem, but only in the data files, you can ignore the parts about the cellular modem in this chapter.

8.1 The basics

The OMC-048 has an internal cellular modem. To use this modem, you need to include these blocks in your config file:

- Modem:
- Connection_settings:
- Data_file:

These blocks are described in chapter 10, but in this chapter I will explain the timing relations between the *transmit_interval* (specified in ’connection_settings’), the *create_interval* (specified in ‘Data_file’) and the sensor timing (described in chapter 7).

Basically the process is this:

- Whenever the logger application is started, a data file is opened. From that moment on (old) files are closed and (new) files are opened on the specified schedule (*create_interval*).
- Sensor drivers produce data on their own schedule (*sample_interval*). The data is written to the data file that is open at the moment the driver wants to write the data.
- Data transmission also runs on its own schedule (*transmit_interval*). A transmission will attempt to transmit all files that are waiting for transmission (see below). The file that is currently open will not be transmitted (because sensor drivers are busy writing data to it).

Data files are created on the SD card in the /data folder. This is where they ‘line up’ for transmission. Once transmitted, they are moved to a daily folder in the /data/transmitted folder. If multiple files are waiting for transmission, they will all be transmitted (and moved to the daily folder in /transmitted) in the order they were created (oldest first). See Figure 9. For an explanation of the file name and the file format, refer to section 18.1.

Notes:

- **Communication breakdown:** When you visit a logger that suffered from a long-term communication breakdown, a huge amount of data may have built-up in the /data folder. You may consider copying this by USB to your computer and deleting it from SD before fixing the communication problem. Otherwise all data files will be send over the cellular network as soon as the cellular communication is functioning again.
- **Logging without transmission:** If you did not schedule any transmission (so you are using the logger only for internal logging and not for transmitting the data), the files will still be created in the /data folder. But whenever a file is closed as scheduled, the file is moved to a daily folder in the /data folder. If you interrupt the logger application (control-C or power-down), the file is not moved. So you may find some incomplete data files in the /data folder if you have been using control-C. See Figure 9.

Finally, note that all data from all sensor drivers (configured in the config.txt file) is stored in the data file. The whole data file is transmitted. Thus there is no need (and no possibility) to list which parameters from which sensors you want to be stored and/or transmitted: you get them all. But by using the Log_parameters block (section 11.14) you can include only specific parameters from specific sensors.

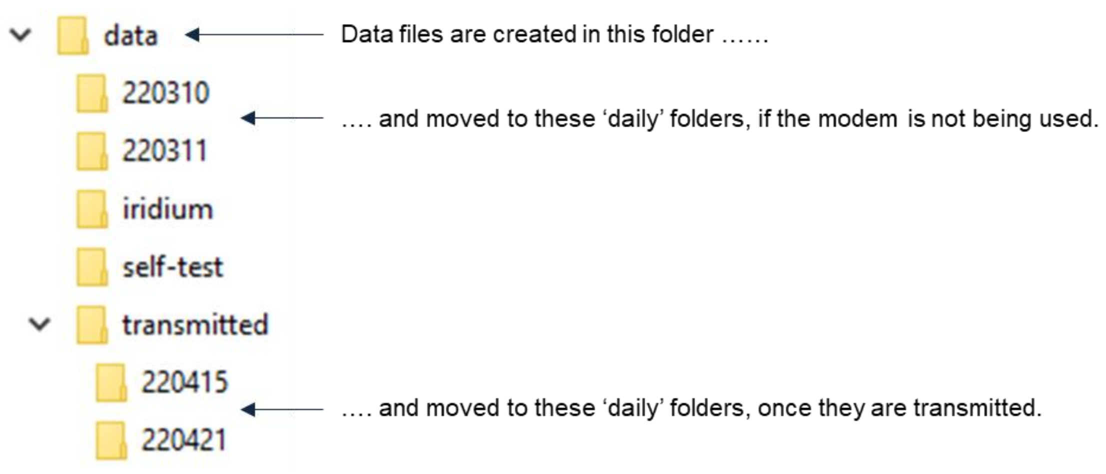

data
220310
220311
iridium
self-test
transmitted
220415
220421
Data files are created in this folder ……
…. and moved to these 'daily' folders, if the modem is not being used.
…. and moved to these 'daily' folders, once they are transmitted.

Figure 9. Location of the data files and folders, with and without cellular communication.

I will explain the scheduling by giving two examples. Then, in section 8.4, I will give some general guide lines for composing an overall schedule.

8.2 A simple schedule to start with

Below I only showed the lines from a config file that are relevant for the timing schedule.

From config.txt (the format has been modified for readability):

```
Connection_settings: transmit_interval  : "50 0,10,20,30,40,50 * * *"
Data_file:           create_interval    : "25 0,10,20,30,40,50 * * *"
Onboard:             sample_interval    : "0 0,5,10,15,….,50,55 * * *"
Analog_voltage:      sample_interval    : "0 * * * *"
```

If you read from bottom to top, you can see there are two sensors with a *sample_interval*, there is a data file *create_interval* and there is a *transmit_interval*. (Actually, the word 'interval' is a bit misleading, because what is actually defined is an infinite series of time-points. But never mind 😉.)

- The *sample_interval* means exactly what you would expect: it indicates the time-points at which a sample is taken from the sensors and is written to file. As you saw in section 7.3.3, sensors that introduce a delay will be sampled later than the indicated time points.
- The *create_interval* indicates the time-points at which the current data file will be closed and the next one will be opened. Data from the sensors is always written to the file that is open at that moment.
- The *transmit_interval* indicates the time-points at which an FTP or TCP transmission is started. It may take a while before the connection is made and the actual transfer starts. Only closed files can be transferred. (Transmitting an open file would mean that the file is transferred while sensor data is still being written.)

Note: Each time the application is started, it will immediately open a new data file, so it has a place to store its samples. Thus the first file after starting the application may cover a (much) smaller interval than the following files. Similarly, if you stop the logger (control-C or power down), you may be left with an incomplete file.

Okay, so what will happen with the schedule shown above? Take a look at Figure 10. As you can see, all samples from minute 1 to minute 10 (including) go into file number ‘N’. The sample from minute 0 went info file number N-1 and sample 11 will go into file N+1. The data transfer starts after the closing of file N-1 and the opening of file N. So the file N-1 will be transferred. Any older files (if they have not been transferred earlier for some reason) will also be transferred. File N will have to wait till the next data transfer, as shown in the figure.

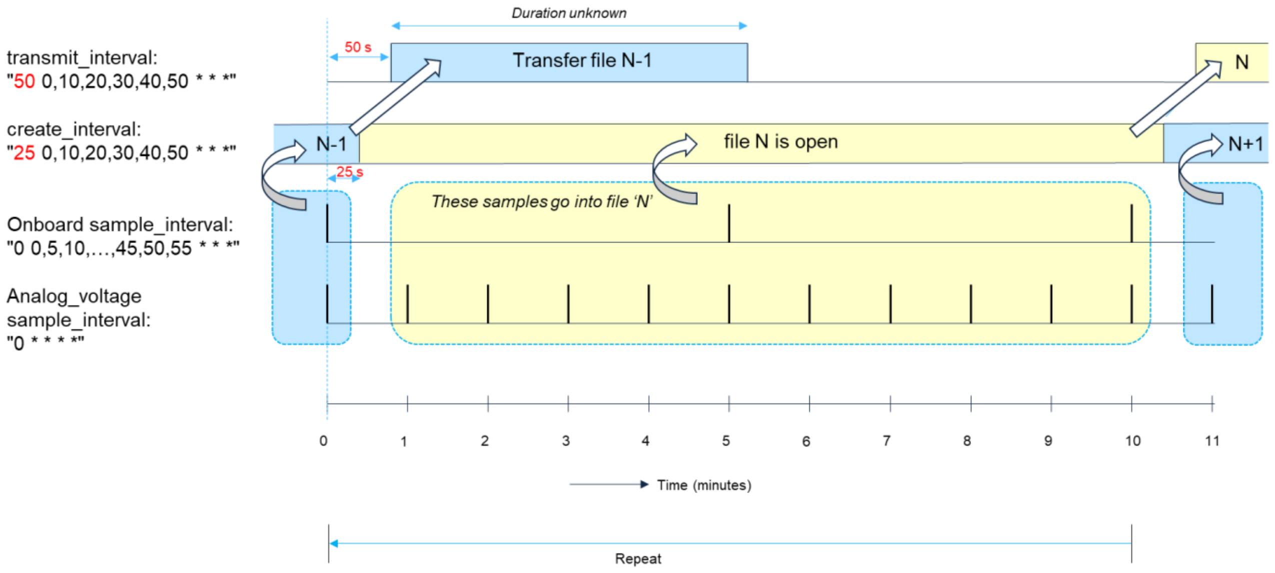


Figure 10. Timing diagram for a simple timing schedule (fast sensors that can be sampled without delay)..

It is important to understand that if multiple events are scheduled at exactly the same moment (the same second), the logger will execute them in arbitrary order, because things cannot happen at exactly the same time. To clarify this, consider the following two modifications to this example:

1. Suppose that you change the ‘second’ of all time specifications to 0. Thus: `create_interval   : "0 0,10,20,30,40,50 * * *".` Then there is no way of telling if the samples at second 0 of minute 0 go into file N-1 or N, because the file closing/opening occurs at ‘the same’ moment as the writing of the sample to the file. Maybe one sample goes into file N-1 and the other in file N. It is undetermined.
2. Suppose that, you would put the transmit_interval at the same second as the create_interval. For example, both at second 25. Then the open/close of the file happens at ‘the same’ moment a file is selected for transmission. In this case, there is no telling if file N-1 is closed in time for transmission, or if it will have to wait for the next transmission. Actually, because the transmit process has an internal power-on delay of 20 seconds for the modem, you can schedule the create_interval and transmit_interval at the same moment. If you schedule both at second 25, the transmit process will access the data file at second 25+20=45.

So, to make sure everything happens in the order you want, put things on slightly different moments. Like we did in Figure 10 by shifting the file-create 25 s with respect to the samples, and shifting the transmit_interval another 25 s (which could also be 0 s, because of the internal 20 s delay). If you don’t care which sample ends up in which file and if the file is transmitted in one interval or another, you don’t have to bother with these time shifts. Nothing will go wrong if you schedule multiple ‘events’ at the same time; they will just happen in an arbitrary order.

8.3 A schedule with sensors with delayed response

Below I only showed the lines from a config file that are relevant for the timing schedule.

From config.txt (modified format for readability):

```
   Connection_settings:  transmit_interval   : "0 2 * * *"
   Data_file:            create_interval     : "0 1 * * *"
   Onboard:              sample_interval     : "0 0,10,20,….,50 * * *"
   navilock_md6:         sample_interval     : "0 5,15,25,….,55 * * *"
```

As you can see, we now transmit every hour (on minute 2). And we transmit the file that was closed a minute ago (on minute 1 of every hour). The onboard samples are taken on minute 0, 10 and so on. So far it is simple and Figure 11 should be easy to understand.

The complication is in the GPS. This is a sensor with a variable delay, as described in section 7.3.3 and section 7.3.4. Suppose the timing specification for this sensor is:

```
   sample_interval     : "0 5,15,25,….,55 * * *"
   startup_time        : 20
   response_timeout    : 280
```

As explained in section 7.3.4, the whole sampling process now takes at most 20+280=300 seconds, or 5 minutes. If the GPS is fast, the process may take just slightly more than a 20 s (the startup_time), but if the sensor is slow, it can take the full 5 minutes.

Figure 11 shows the timing schedule. The GPS process is shown as blocks of 5 minutes. As just explained, these 'blocks' could last anything from 20 s to 5 minutes. Notice that this schedule assures that it is predetermined which samples go into which file and when this file will be transmitted.

Figure 11 also shows, in red, what would happen if you change the GPS timing specification as indicated in red. Now the sample starting at second 0 of minute 0 could be on time for 'file N-1' (if GPS is very fast), or could end up in 'file N'. You may or may not care about this. The sample will not get lost; it will just arrive earlier or later on your server.

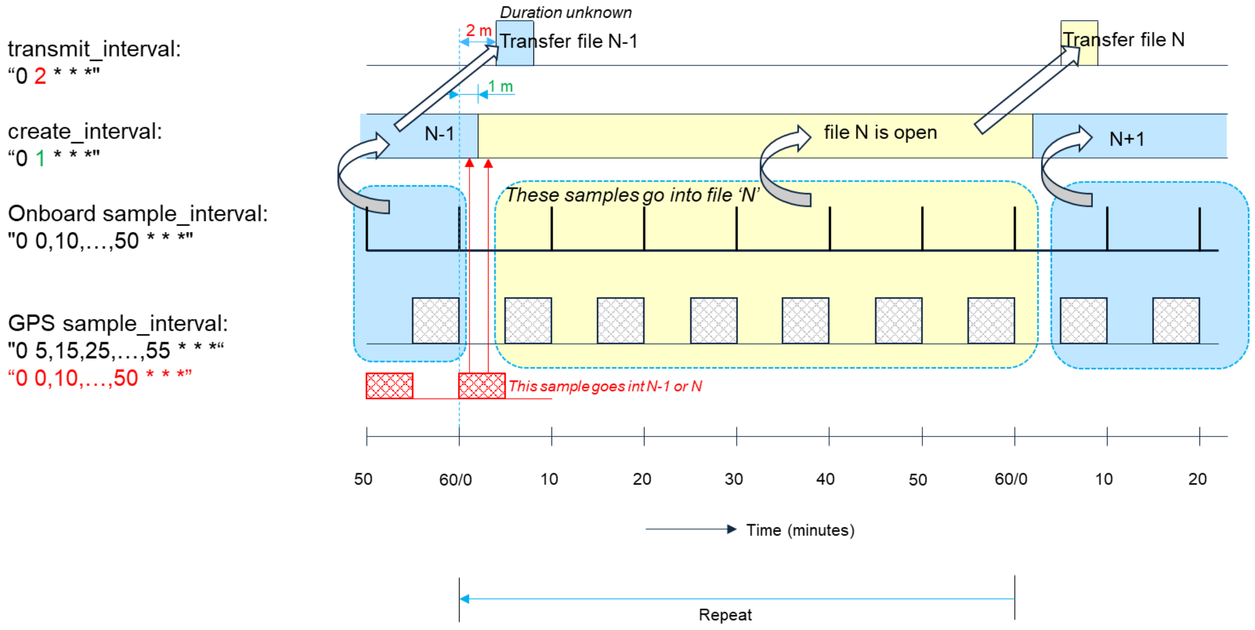


Figure 11. A schedule for sensors with delayed response.

8.4 General guidelines for designing a timing schedule

As the preceding sections have shown you, you have great flexibility and control over the timing of your system. As long as your settings for the sensor timing match those of the connected sensor, there are little or no 'rules' that limit you. You can select the intervals of all sensors (sample_interval), files (create_interval) and outputs (transmit_interval) independently of each other. Anything will work, but not everything makes sense.

To assist you in creating a sensible overall-schedule, I will present some guidelines. This will be sufficient for most cases, but note that you are not limited by these guidelines. You can deviate, if you know what you are doing. If you don't know what you are doing: don't do it.

First of all, you need to determine the *sample interval* for each sensor:
For each sensor, determine the maximum duration of taking a sample. Refer to section 7.3. You may need to consult the sensor manual about power-up time and response delay. You may decide to keep the sensor powered continuously. In complicated cases, or if the sensor manual is not clear enough, you may need to experiment with the sensor to figure out how long the whole process takes, from powering-on the sensor till powering-off the sensor. For simple sensors (like the onboard sensors), the trivial answer is: it takes about 0 s. Once you know this maximum duration, you can choose the sample interval. The sample interval should obviously be larger than the maximum duration of taking a sample.

Once you know the desired sample interval, you can write the timing specification (sample_interval: "s m h d w") for the sensor. Probably you want to aim for getting time stamps on nice round values. This has been demonstrated in section 7.3. You can also fill-in the startup_time, response_timeout, cooldown_time and any other sensor-specific timing settings.

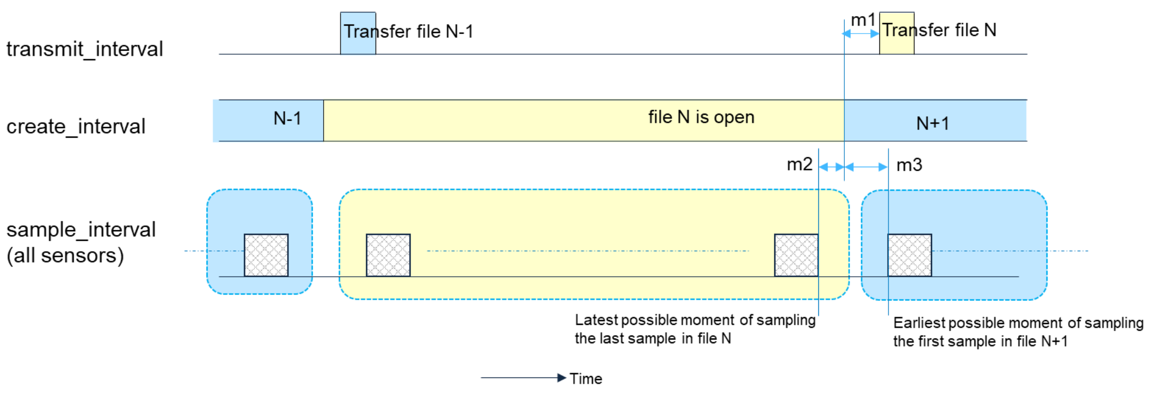
Next, you have to decide on the *transmission interval* and on the *file create interval*.
In most cases you will want these intervals to be the same, so here they are treated as one. In choosing an interval, there are several things to consider:

- If you want to receive the data as quickly as possible, you should schedule the transmission at the same interval as the sensors. But don't make this interval shorter than 5 minutes.
- Each transmission has an overhead. The shorter the transmission interval, the more transmissions, the bigger the overhead. So, if you transmit at a very short interval, your SIM cost and your power consumption will be relatively high.
- If you don't need the data 'quickly', you could transmit just 'occasionally'. There are several practical reasons (I won't go into the details now) to transmit at least twice per day. Like at 03:00 and 15:00 hours. (Note that you could also transmit just once per month, but what is the advantage over twice per day?)
- If you are logging a lot of data, I recommend to transmit at a relatively short interval, to avoid files getting too big. Particularly if the network coverage is poor, it is wise to attempt transmissions more often and with smaller file sizes. You may now wonder: how much data is 'a lot of data' and how long is a 'short interval'? Right. As a rough indication, 100 kB is a lot and you should make the interval as short as is needed to get the file size below this value. In practice, if you suffer from erratic communications, make the interval shorter.

Now you have established the *transmit_interval* and the (equal) *file create_interval,* you can write the timing specification ("s m h d w") for both. But take note of the following paragraph.

If you want to make sure samples end up the right file and not one earlier or later, you need to take care of what was explained in section 7.3.4. There it was explained that, for each sensor, there is a *window* during which the actual sampling moment can occur. For some sensors this window may be 0 s wide, but for others it may be many seconds or even minutes wide. Figure 12 shows how you can position the file create_interval and the transmit_interval w.r.t. the windows of the sensors. For the margins m2 and m3, I recommend you take at least 10 seconds. If the sample interval is too short for that, you can take lower values. For the margin m1 you can take zero, because the transmit process will take about 20 s before attempting to read the data file. So this gives you a margin of 20 s anyway. If the interval is many minutes, you may increase the margins to a minute. Figure 11 gives an example of that.

If you don't care if samples get transmitted in one file or the other, you also don't have to care about the precise positioning of the create and transmit intervals. It will work anyway, but maybe data arrives on your server later than you expect, if you don't take care.



transmit_interval
Transfer file N-1
m1
Transfer file N
create_interval
N-1
file N is open
N+1
m2
m3
sample_interval (all sensors)
Latest possible moment of sampling the last sample in file N
Earliest possible moment of sampling the first sample in file N+1
Time

Figure 12. The transmit and create intervals should best be shifted w.r.t. the sample windows.

9 Iridium satellite communication

9.1 The basics

Typically, you will use either Iridium or cellular communication and not both. However, you can use an external Iridium modem next to -and independently of- the internal cellular modem. This means that data files will be created for both independently and the transmissions can be scheduled at independent intervals.

To use the external Iridium modem, you need to:

- Connect the Iridium modem (section 9.2)
- Include these 3 blocks in the config.txt file (chapter 10):
    - Iridium_modem:
    - Iridium:
    - Iridium_file:
- Create a customized '*iridium_reference_tables.py*' file (section 12.5).

Note: If the logger was prepared for remote control using Blue2Cast (for a previous application), you need to remove the remote control option! The remote control option is not possible without cellular communication, but if the logger is configured to use remote control, it will waste time and power on useless attempts to connect to the remote control server. Refer to section 16.4 to learn how to remove the remote control settings.

Iridium communication is somewhat similar to what happens with cellular communication, but there are a few major differences.

One major difference is that not all sensor data is written to file by default. Instead, you need to create a customized "iridium_reference_tables.py" file, specifying exactly what to write to file. You can create this file from the "iridium_reference_tables_template.py" Python file. This is described in section 12.5.

The reason for this difference with cellular communication, is that satellite communication is very expensive -per transmitted byte- compared to cellular communication. So you may want to limit the amount of parameters to transmit, as well as the number of bytes spend on each parameter. The iridium_reference_table allows you to control all this.

The handling of data files for Iridium is also a major difference with 'normal' data files. Data files for Iridium transmission are created in the /data folder. So far no difference with cellular. They have a different name and another format, refer to section 18.2. Once a file is transmitted, it is moved to a daily folder in data/iridium (not 'data/transmitted'). If the transmission fails, it is also moved to that folder. This means that during the next scheduled transmission, no attempt will be made to transmit the file again. Only the latest file (still in /data) will be transmitted. The advantage of this scheme is, that in case of bad coverage (many failing transmissions) you don't end up with a pipeline of un-transmitted files that clog up the transmission. The disadvantage is, of course, that you may lose data. But this is inevitable with a bad connection, given the fact that Iridium gives you only a limited SBD packet size and a minimum time between packets (see 9.3).

Once you get access to the SD card (either by USB or remote, once cellular communication is established), you can recover all files (transmitted by Iridium or not) from the /data/iridium folder. Even though everything is there, you may have no tools to read all the data and convert it to .csv or another usable format. Therefore it is wise to also use 'Data_file' to create data files that can be read by Blue2Cast. This works in parallel, and independent of, the Iridium files. It makes sense to include the below lines to get daily data files in the 'OMC-045 data format':

```
Data_file:
- id: data
  create_interval: "0 0 0 * *"
```

Note: You cannot have the sampling of the sensors at another interval for Iridium then for the normal logging in the data files (as used for cellular communication).

Advise:
Even if you only want to use Iridium communication, do use ‘Data_file’ anyhow. You can use this to create -for example daily- files that are easier to read and process, in case you need to recover (un-transmitted) data from the SD card. Note that Blue2Cast can read these files, but not the Iridium files.

Before digging into the timing details of the Iridium communication, don’t forget that you need to physically connect the Iridium modem to the logger! Let’s start with that in the next section.

9.2 Connecting the Iridium Edge modem ‘OMC-IEM’

The external Iridium Edge modem (OMC-IEM) uses one serial port and one power port of the logger. Below table shows the connection of the modem to the logger. Serial port 1 and power output 1 are used as an example; you can use any port you like. Obviously, your connections must match with your settings in the config file.

OMC-IEM connection example for Serial port 1 and Power port 1.

IEM pin #, name	Color	OMC-048 pin #, name
1 spare	White	
2 Ground	Brown	39, GND
3 RS232 RX	Green	31, Serial1, TX
4 RS232 TX	Yellow	34, Serial1, RX
5 Power (9-32V)	Grey	2, +VDC (or other power supply)
6 On/Off (Off<1.5V, On >1.5 V or floating)	Pink	7, Power Port 1 + 22 kΩ resistor to GND.
7 Network available	Blue	
8 Power Detect	Red	

Note: The resistor is to assure that, with the power port from the logger switched off, the voltage of IEM pin 6 (logger pin 7 in this example) is really below 1.5 V, so the modem goes to sleep.

Once connected to the logger, you can check the interface using port redirection (see section 14.2). The OMC-IEM is set to **19200 Bd, 8 bits, No parity, 1 stop bit**. The modem should respond to the AT command ‘AT’ with ‘OK’. You can also try other AT commands, if you are familiar with them. For example, ‘AT+CGSN’ returns the IMEI number and ‘AT#SERVINFO’ shows ‘Serving Cell Information’.

If you don’t get a response from the modem on your AT commands, check the following:
1. Is there 12 V (nominally) on IEM pins 5 ánd 6 with respect to Ground?
2. Is RX correctly connected to TX and the other way around? (NOT RX to RX and TX to TX !)

Once this is done, you can continue with modifying the config file and the reference table (.py file). Note that, for testing actual satellite communication:

- The modem should be outside and have a clear view of the sky.
- You need to register the modem with its IMEI number with an Iridium provider.

9.3 Iridium timing and scheduling

The major difference with cellular communication is that the intervals for sampling the sensors, creating the data file and transmitting the data file, should all be equal (with the proper shifting in time). Things should happen in this refreshingly simple order:

1. Open a new data file
2. Get data from the sensors
3. Close the data file (open a new one)
4. Transmit the data file

This is a lot simpler than with cellular communication, but also less flexible. For example, you cannot have sensors running at different intervals. (Only sensor not included in the Iridium data file can run independently).

There are two restrictions of the Iridium SBS network that we need to be aware of:

- The Iridium network does not allow registrations at an interval shorter than 3 minutes. To be safe, I recommend a transmission interval equal or greater than 5 minutes.
- A single message cannot contain more than 338 bytes.

A typical timing diagram is shown in Figure 13. (The sensor with the response_timeout could of course also have a startup_time, but that is not relevant for the basic idea shown here.) Note that each sensor is sampled precisely once in each create_interval.

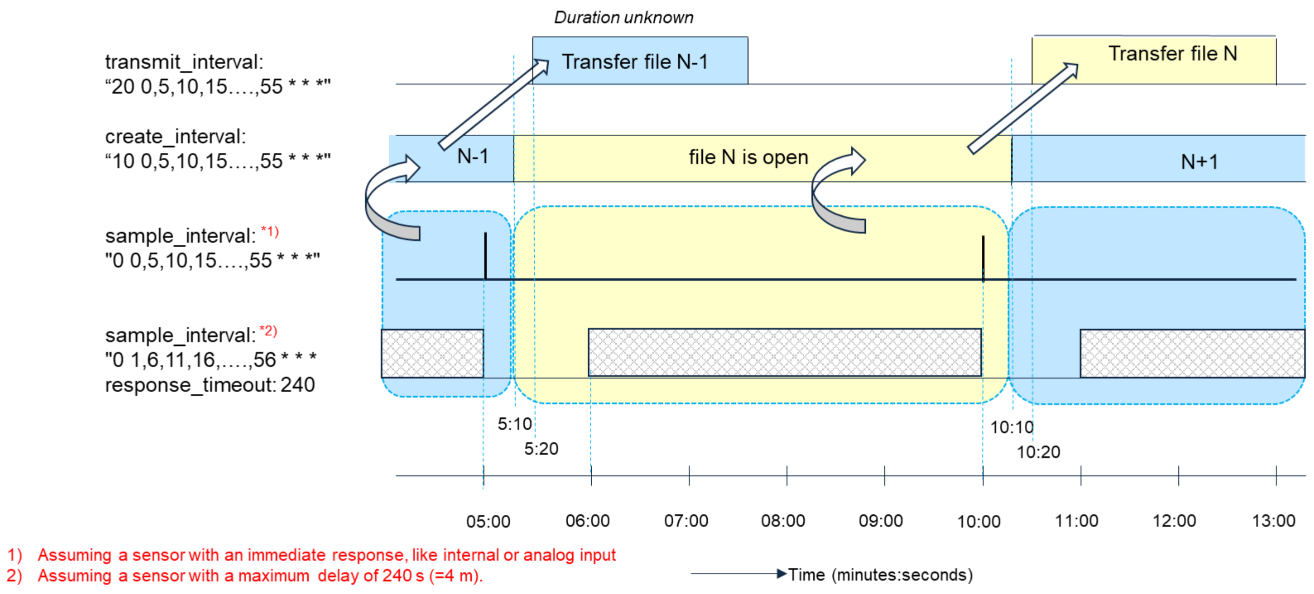


Figure 13. Typical timing for Iridium. The 5 minute interval can be increased the obvious way.

Note that parameter are written to file in the order in which they are sampled. Thus, if you want them to appear in a particular order in the Iridium file, you need to restrict the timing such that this order is guaranteed. Normally the order should not be relevant, because the file contains <tag,value> pairs. See sub-section 18.2.2.

10 The config file, common blocks

Chapter 6 already explained what the function of the config file is, and how it is build up. This chapter presents some of the blocks that can be included in the config.txt file. Blocks that correspond to *specific drivers* are described in chapter 11 and blocks that correspond to *generic drivers* are described in chapter 12.

Note that in section 6.2 some 'common settings for drivers' were described. These are not repeated every time in the descriptions of the drivers.

Obviously, not all blocks will be present in a 'real life' config file. As described in section 6.3, you are encouraged to use the 'config template' to create your own config.txt file from scratch, including only the blocks you need.

10.1 Omc048:

This block is always required. It is typically located at the top of the config.txt file.

```
Omc048:
  usb_mode: debug                # Explained in chapter 3
  system_id: Example             # Free to choose. Will appear in the file name.
  application: Example           # Free to choose.
  file_log_level: info           # Explained in section 14.1 and 14.3
  repl_log_level: info           # Explained in section 14.1
  utc_time_offset_hours: 0       # Explained in section 7.1
  utc_time_offset_minutes: 0     # Explained in section 7.1
  sensor_data_print: False       # Explained in section 14.1
  self_test: True                # Explained in section 14.4
```

10.2 Data_file:

In most cases you will want the logger the save the data in data files on the SD card. These are also the files that are transmitted by the internal modem. If you don't include the data_file: block, the logger will anyway open a data file at startup. All data will go into this single file, till the logger is stopped. This is not convenient, so it is recommended to always include this block.

Basically, the data-file block tells the logger at which points in time one data file should be closed and the next one be opened. The currently open file is used to collect values from the sensors. It is located in the 'data' folder on the SD card. The file name contains the time stamp at the moment of creation (=opening of the file). Once closed, the file can be transmitted and/or moved. Transmitted files are moved to a daily folder in /data/transmitted. If cellular communication is not used, the files are moved to a daily folder in /data.

Below example will create a new file on second zero of minute zero of every hour.

```
Data_file:
- id: data
  create_interval: "0 0 * * *"
```

10.3 Modem:

Many applications will use the internal cellular modem. It is called 'modem' and requires the below block in the config.txt file.

```
Modem:
  id: onboard_modem
  port: modem
  sim_username: sim_usr
  sim_password: sim_pwd
  apn: sim_apn
  network_technology: GSM + LTE
  iot_technology: CAT-M1
  # band_select: 5,0,168695967
  cellular_diagnostics: True
```

Only one port is currently supported for the modem, which is the internal port named ‘modem’.

Make sure that your SIM card and its settings (user name, password and APN) correspond to the technology you select.

`network_technology:`

‘GSM + LTE’	Default. The modem will automatically select between ‘GSM’ and ‘LTE’
‘GSM’	Force to the ‘old’ GPRS technology (= 2.5G).
‘LTE’	Force to 4G LTE-M1.

Note that ‘4G LTE’ is different from ‘4G LTE-M1’; Some networks may support LTE (for mobile phones) but not support LTE-M1 (for IoT applications). The OMC-048 cannot be used on these networks, unless they do support NB-IoT or GPRS.

`iot_technology:`

‘CAT-M1’	Default: The modem will use the technology indicated by network_technology.
‘NB-IoT’	Force to NB-IoT.
‘CAT-M1 + NB-IoT’	The modem will automatically select which technology to use.

`band_select:`

Do NOT use this setting, unless you are very knowledgeable on this subject! Also refer to the modem AT reference manual, section: 3.2.14. AT#BND - Select Band.

`Cellular diagnostics`

With ‘cellular_diagnostics: True’ you will switch on an internal virtual sensor which samples several diagnostic values that help you judge the quality of your cellular network connection. These values will be logged like any other sensor value. The sensor has the id of the modem. In this example the sensor id is ’onboard_modem’.

The available parameters with their name, unit and tag are:

- 'Signal strength', 'dBm', 'SSTR'
- 'Provider', '', 'PRVD'
- 'Cellular technology', '', 'CTEC'
- 'Roaming', '', 'ROAM'

Note that you cannot specify a sample_interval for this sensor, like you can for other (internal) sensors, because the ‘cellular’ sensor always runs with the transmission interval. In other words, each time the modem is transmitting data, it samples the cellular diagnostic data (if set to True).

By using cron_offset (in the ‘connection_settings:’ block) you can modify the time stamp of the cellular diagnostic parameters. See section 7.3.5.

In the ‘Log_parameters’ block (section 10.7) you can make a selection of these diagnostic parameters, if you don’t want all of them. For example, if you only want the ‘Signal strength’ to be logged and the id of the modem is ‘onboard_modem’:

```
Log_parameters:
  onboard_modem:
  - Signal strength
```

Finally, note that a signal strength of -80 dBm is (a lot) less strength than -60 dBm. Thus -60 dBm is stronger (better) than -80 dBm.

10.4 Connection_settings:

This block relates to the connections made by the internal cellular modem:

```
Connection_settings:
  data_protocol: FTP
  ftp_url: ftp.omc-data-online.com
  ftp_port: 21
  ftp_username: username
  ftp_password: password
  transmit_interval: "0 0,5,10,15,20,25,30,35,40,45,50,55 * * *"
  utc_time_sync: True
  utc_time_server: europe.pool.ntp.org
  utc_time_server_port: 123
  webserver_access: True
  webserver_url: blue2cast.com
  webserver_port: 5000
  webserver_password: <Contact Observator>
  cron_offset: False # affects time stamp of cellular_diagnostics
```

The settings fall into four categories:

1) Data protocol

If ‘data_protocol: FTP’ the logger will transfer the data files to the specified FTP address. The example shows the FTP server hosted by Observator. You will need the username and password of the FTP server. If ‘data_protocol: SSL’) you don’t need to fill-in the FTP details, because data will be transferred to the Blue2Cast server over the SSL connection.

2) Time synchronisation

NTP is a protocol for time synchronization over the internet. If ‘utc_time_sync: True’, the logger will regularly synchronize its internal clock. Synchronization will occur at the transmit interval (because then the modem is ‘on’ anyway), but only if the previous sync was more than 4 hours ago. Also, if the internal clock is less than 2 seconds off, the synchronization is skipped.

Note that time servers do not always respond, so some synchronization attempts may fail before one succeeds. This is no problem, because the internal clock only drifts slowly. Above example shows a server that works well in The Netherlands, but you may select a server that works well in your region.

Note that you can also use GPS (if connected) to synchronize. Refer to chapter 7 for more on the internal clock.

Finally, note that if you use the Blue2Cast webserver, the logger MUST use the UTC time zone. Otherwise you are free to choose your zone by specifying a time offset in the Omc048: block. See sections 7.1 and 10.1)

3) Webserver access

If ‘webserver_access: True’, the logger will make a secure TCP connection (SSL) to the specified Blue2Cast webserver at the transmit_interval. This allows remote control of the logger.

If ‘data_protocol: SSL’, the logger will also transfer its data over SSL to the Blue2Cast server.

Above example shows the webserver hosted by Observator. You will need a password for connecting to a Blue2Cast server, refer to chapter 16.

4) Cron_offset

To give the cellular diagnostics a nice rounded time stamp.

10.5 Input_power_monitor:

Available for loggers with SN>48000200 only.

The power monitor allows you to let the logger shut down below a certain supply voltage, and restart above another (higher) voltage. It uses the value measured by the onboard 'Supply voltage' sensor. Please refer to section 10.6 and take note of the remark about this voltage being slightly below the voltage on the logger's power pins.

This feature is intended for battery-powered systems, with or without (solar) charger.

To use the power monitor, include below lines:

```
Input_power_monitor:
- id: power_mon
  hysteresis_high: 12.3
  hysteresis_low: 11.5
  sample_interval: "second minute hour day week-day"
```

hysteresis_low and sample_interval

At the specified time points (sample_interval), the monitor checks if the supply voltage is below the hysteresis_low value. If so, the logger goes into a low-power mode. Open files are closed and the power output ports are switched off. The two internal relays are also switched off. Processes running at that moment (a transmission, or sampling of a sensor and operating its wiper, or whatever else) are aborted immediately. This all reduces the power consumption as much as possible.

Hysteresis_high

Once in low-power mode, the logger checks every 3 seconds if the supply voltage is above the hysteresis_high value. If so, the logger goes back to its normal mode. Processes will start at their scheduled times. They will NOT be resumed from the point where they were interrupted.

Note:
As long as a USB connection is established, the logger will not actually go into low-power mode, because then the USB connection would be lost. Instead, if you watch the REPL, you can see that the logger reports going into low-power mode, but the power ports and the relays are NOT switched off. A real low-power mode is only possible when there is no USB connection.

Notes:

- Allow for sufficient hysteresis (=difference between the high and low values) to avoid repeated on/off switching of the Input_power_monitor.
    - The hysteresis_low should normally correspond to the battery getting dangerously low (slightly above the point where a solar charge controller may switch off power completely).
    - The hysteresis_high should correspond to a reasonably well-charged battery (possibly while it is charging). Typically, this value should be at least 1 V above the low value (depending on the used components).

* Choose the sample_interval well. It may not be wise to activate the Input_power_monitor in the middle of all kind of processes, like wipers that are running, or a data transmission that is in progress. If possible, chose the time points such that they fall in ‘dead’ intervals. For example, if your sensors and other devices (including the modem) are activated for some minutes every hour on the hour, activate the monitor somewhere between 2 successive hours, e.g. sample_interval: ‘0 30 * * *’.

10.6 Onboard:

To use the onboard sensors, include below text in the config file:

```
Onboard:
- id: board_sensors
  sample_interval: "0,20,40 * * * *"
```

Adapt the sample_interval to your needs. The internal sensors are (name, unit, tag) :

* 'Int. Temperature', 'C', 'ITEMP'
* 'Int. Humidity', 'rh', 'IHUM'
* 'Int. Coin cell voltage', 'V', 'IVBAT'
* 'Supply voltage', 'V', 'VSUP' (only for loggers with SN>x0200)

The coin cell is the one inside the logger, behind the right side panel.

For ‘board_sensors’ you can chose whatever name you like.

The supply voltage is measured by an internal AD converter, located behind the power protection circuit. It therefore measures an internal voltage that is about 0.2 to 0.3 V below the external voltage (on the logger’s power pins 1 and 2).

To log only a selection of the above parameters, see 10.7.

10.7 Log_parameters:

This block is optional. It is typically located at the bottom of the config.txt file.

Sometimes a (serial) sensor gives more parameters than you are interested in. If you specify *Log_parameters* for a specific sensor, only the listed parameters will be logged. Let’s have a look at an example:

Example:

```
Log_parameters:       # This is a keyword and should  be exactly like this
  Gps0:               # The id of the sensor that ‘talks too much’
  - Latitude          # All parameters that should be logged for this sensor
  - Longitude         # ,,
  board_sensors:      # The id of another sensor that ‘talks too much’
  - Int. Temperature  # All parameters that should be logged for this sensor
  - Int. Humidity     # ,,
  onboard_modem:
  - Signal strength
```

This example assumes that you are using a GPS sensor with id: gps0 and that you are using the internal sensor with id: board_sensors and that you switched-on 'cellular_diagnostics' for the modem with id: onboard_modem.

Note that you have to identify each sensor by its **id**, not by the driver name. After all, one driver can be used for multiple sensors, but each will have a unique id. The parameters are identified by their **names**. To figure out what parameter names are available for a given sensor, please refer to the sensor_data_print setting explained in section 14.1.

11 Specific drivers

This chapter describes drivers that are specific for a particular sensor or other device.

Note that in section 6.2 some 'common settings for drivers' were described. These are not repeated every time in the descriptions of the drivers.

11.1 EXO:

The EXO driver is a dedicated driver for EXO™ sondes from Xylem/YSI. In this section, I assume that you are familiar with these sondes. You may need to refer to the EXO manual for more information.

The EXO driver works with all sondes (EXO1, EXO2, EXO3) including the 'short' versions. The driver uses RS232 communication with the DCP adapter. For SDI-12 communication, see 12.3 . Daisy chaining multiple EXO's on a single DCP is not supported, but you can connect an EXO to each serial port.

To use an EXO, you need to perform the following steps:

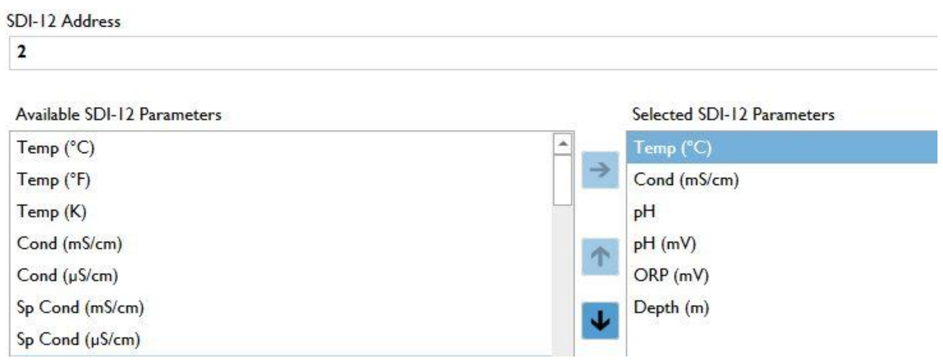
1) Prepare the sonde using KOR software.
2) Connect the sonde to the logger, using a DCP module.
3) Add the 'EXO:' block to the config file.

These steps are described in the following sub-sections.

11.1.1 Prepare the sonde using KOR software

Preferably (not mandatory) you first place the wiper and sensors that you intend to use. Connect the sonde to your PC (USB or Bluetooth) and connect to it by KOR. Create or edit a deployment taking these points into account:

- The wiper will be controlled by the logger, so in KOR you can set the wiper interval to 0. This switches off automatic wiping, but the logger can still command a wipe.
- The sampling interval only applies to internal sampling (EXO internal logger). If you don't need internal logging, you can arbitrarily select an interval of one hour. It does not affect the sampling by the OMC-048.
- In the DCP section, you should select the desired output parameters. Even though it is called 'SDI-12', it also applies to the RS232 output of the DCP. The SDI-12 address is not relevant. The order of the parameters is also not relevant. Make sure you selected the right parameters <u>in the desired unit</u> (like mS/cm or µS/cm). See Figure 14.
- Select the desired averaging (default, rapid, accelerated).
- Save the deployment to the sonde. You don't need to start (internal) logging.



SDI-12 Address: 2

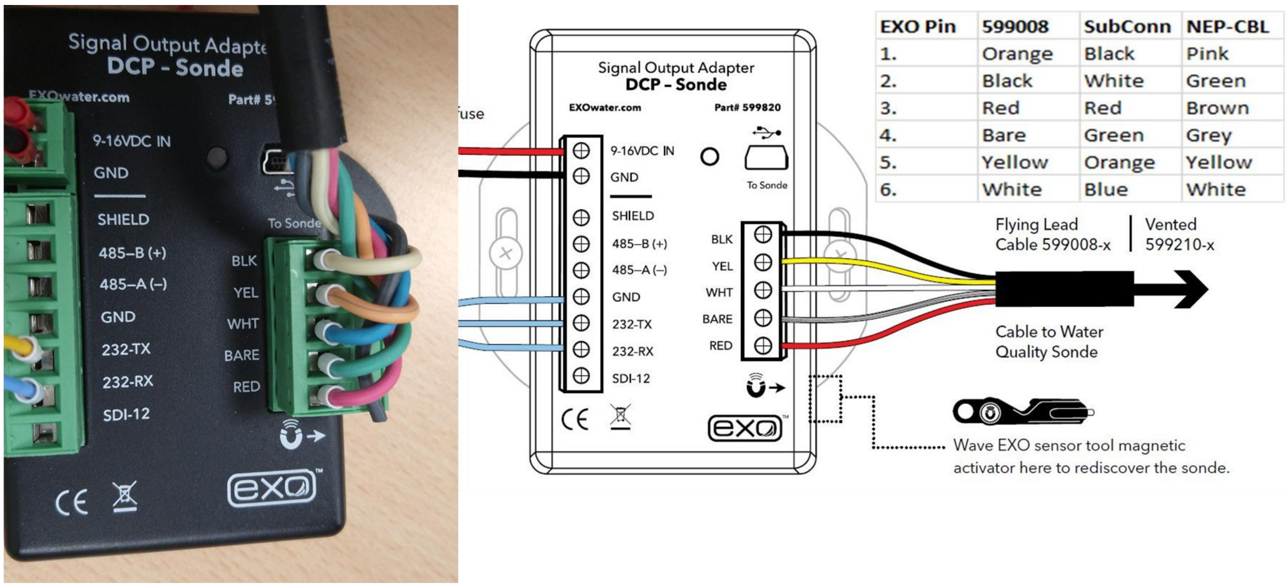
Available SDI-12 Parameters	Selected SDI-12 Parameters
Temp (°C)	Temp (°C)
Temp (°F)	Cond (mS/cm)
Temp (K)	pH
Cond (mS/cm)	pH (mV)
Cond (µS/cm)	ORP (mV)
Sp Cond (mS/cm)	Depth (m)
Sp Cond (µS/cm)	

Figure 14. Example of parameter selection in the KOR software for DCP output.

Select exactly the parameters you want. Not less, but preferably also not more!

11.1.2 Connect the sonde to the logger, using a DCP module

The EXO manual describes how to connect the EXO to the DCP. The colors printed on the DCP module correspond to the colors of the wires in the YSI Field cables 599008-x. The table in Figure 15 (top-right) indicates these colors (column '599008') as well as the corresponding pin numbers on the EXO connector. The table also shows the corresponding colors of two other cables (Subconn n and NEP-CBL) used frequently by Observator. The picture insert on the left show how the 'Subconn n' wires should be connected. Note that EXO pin 1 (color either Orange, Black or Pink) is not connected. If you are in doubt which color is which wire, use an ohm-meter to find out which color corresponds to which pin (first column in the table) of the connector mating to the EXO connector.



EXO Pin	599008	SubConn	NEP-CBL
1.	Orange	Black	Pink
2.	Black	White	Green
3.	Red	Red	Brown
4.	Bare	Green	Grey
5.	Yellow	Orange	Yellow
6.	White	Blue	White

Figure 15. EXO to DCP connections for various cables, commonly used by Observator.

The DCP module must be connected to the OMC-048. Below table shows the connections in case serial port 1 and power port 1 are used.

Example: Connecting the DCP to Serial 1 and Power Port 1.

DCP pin	OMC-048 pin	Comments
9-16 VDC IN	Power Port 1 (pin 7)	Convention: a red wire
GND	GND (pin 3)	Or a GND pin close to the serial port. Convention: a black wire.

GND		Not connected, to avoid ground loops.
232-TX	Serial 1, RX (pin 34)	DCP output ->logger input
232-RX	Serial 1, TX (pin 31)	DCP input <- logger output

It is important to make sure that the baud rate of the DCP module is set to 9600 (or at least corresponds to the setting you make in the config file). Mostly (but not always!) this is the case with modules straight from the factory. The DCP module contains more settings that are relevant, but you don't need to bother with these, because the logger will set them properly. But it can do so only if the baud rate is correct.

Note: The baud rate of 9600 works fine for sample intervals longer than 5 seconds. If you have a shorter sample interval, like 1 s, you need to set the DCP to a higher baud rate. We recommend 115200 Bd. You need to set the same baud rate in the config file.

Assuming the baud rate is correct, you can check if everything is set up correctly. If the baud rate turns out to be wrong, you have to first correct that, as described further down.

If you are confident that everything is correct, you can skip below check and move on to section 11.1.3. But if something goes wrong there, you still have to do below checks to find out what the problem is.

Checking the connections

- Make sure you completed all the connections and check them carefully. Also make sure the Sonde has been prepared properly.
- Make sure you understand about using the REPL for controlling the power port and making a serial port redirection ('tunnel'), as described in section 14.2.
- Make a USB connection to the logger and make sure you have terminal access to the REPL in interactive mode (see section 14.2).
- Apply power to the logger. Switch on the power to port 1 (to power-on the DCP) and open a tunnel (redirection) to serial port 1. The commands are summarized below.
- After power-on, the DCP needs about 30 s to 'discover' the EXO. During these 30 seconds, the red LED on the DCP will be constantly on.
- Once the sonde is 'discovered', the red LED will flash at 1 s interval. Now you can issue these commands:
    - <enter>, a few times: to see if you get the '#' prompt. If not, you have to check/change the baud rate as explained below.
    - Para: to get a list of parameters that will be reported by the sonde.
    - Data: to get the actual data from the sonde.

If this all works correctly, you can move on to section 11.1.3. If it doesn't, it makes no sense to move on. You will have to go back to the beginning of this chapter (without getting a reward).

These are the REPL commands to switch Power Port 1 on and to redirect serial 1 with baud rate 9600:

```
import omc048
p = omc048.power_supply(1)
p.on()

s = omc048.serial(1)
s.init(9600, s.RS232)
s.redirect()
```

(Control-C ends the redirection and p.off() switches off the power port.)

Notes:

- If you don't get the '#' prompt, probably the baud rate is wrong, but also check if the wiring is correct and if the DCP is powered.
- You cannot correct typing mistakes to the DCP; just hit enter and try again.
- If you get 'No sonde' it either means that the EXO is not (correctly) connected, or you are too impatient and the 30 s needed by the DCP to discover the sonde have not yet passed.
- Sometimes the DCP seems to be missing the first character you type. For example, you see 'ata' when you type 'data'. This is because the DCP goes to sleep after 30 s of inactivity. The first character will wake up the DCP, but is lost otherwise. In sleep mode, the LED flashes at 10 s interval.

Setting the baud rate of the DCP module.
For setting the baud rate of the DCP, you don't need to have the EXO connected, but it won't harm. Instead of using the logger with port redirection (as described above), you can also use the RS232 COM port of your computer, or use an USB-to-RS232 adapter. In that case, the COM port pinning is:

- Pin 5 = GND, connect to DCP GND

- pin 2 = RX, so connect to TX of the DCP
- pin 3 = TX, so connect to RX of the DCP.

In the below steps, I assumed you connected everything as described above and you started the port redirection. Then:

- Hit 'enter' a few times to see if you get a response from the DCP. If you get the '#' prompt of the DCP, the baud rate is correct.
- If you don't get the prompt but either no response at all, or some 'weird' characters, the baud rate must be wrong. In that case, try all possible baud rates (1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200) till you find the right one. You do this by repeatedly:
    - Typing Control-C to end the redirection
    - Typing these commands to start the redirection again, but replace '9600' with the new Baud rate:
        - `s.init(9600, s.RS232)`
        - `s.redirect()`
    - Hitting 'enter' a few times to see if you now get the '#' prompt.
- Once you get the '#' prompt, use the DCP command "setcomm 5 1" to configure the DCP to 9600 Bd, 8 bits, no parity, 1 stop bit. (Or "setcomm 9 1" for 115200 Bd.)
- Set your terminal program also to 9600 Bd (`s.init(9600, s.RS232)`) and confirm you now get the prompt at this baud rate.

Now you can return to 'checking the connections'!

11.1.3 Add the 'EXO:' block to the config file

I will give two examples:

1. For two EXO sondes with a sample interval of 10 minutes.
2. For one sonde with a sample interval of 1 second and using averaging.

Example 1: two EXO sondes with a sample interval of 10 minutes

```
EXO:
- id: EX1
  port: serial1
  mode: RS232
  sample_interval: "0 0,10,20,30,40,50 * * *"
  supply_port: 1
  supply_port_always_on: False
  wiper_interval: "0 0,10,20,30,40,50 * * *"
  delay_after_wipe: 25
  response_timeout: 120
- id: EX2
  port: serial2
  supply_port: 2
  … <left out some text here, see EX1 above>
```

In this example the driver named 'EXO' is used twice, with id's EX1 and EX2. Thus EX1 and EX2 use the same driver (EXO), but each can be configured independently of the other (thus also with a different interval). Obviously, both sondes use a different serial port and a different power port. The default port settings are all correct, including a baud rate of 9600 Bd, so these don't need to be specified.

If both sensors now output the same parameters, like DO and T, these parameters will be tagged DO_EX1 and T_EX1 for the first EXO, and DO_EX2 and T_EX2 for the second EXO. Thus when the data arrives on the server (or in the data files on SD), the parameter tags will be unique.

Note:
The logger will read all parameters you specified in the KOR software (DCP output). Thus you don't need to specify them in the config file. (But you exclude some in the Log_parameters block, if you want).
After changing (the settings of) the EXO, you need to restart the logger, so it will detect the changes.

Each time the logger application starts, the logger will:

- Configure the DCP (but the baud rate must already be correct).
- Issue the 'para' command to find out what parameters the EXO will deliver.

Once the application is running, the logger will perform these actions every 10 minutes:

- At second 0 (for example on 13:40:00), power to the DCP is switched on. Then wait for 30 s to give the DCP time to connect to the EXO.
- A 'wipe' command is issued. Wait till the wipe is finished. This typically takes 27 seconds, but adapts to the actual wiping time.
- Wait for 'delay-after_wipe' seconds (in this example 25 s), to give the EXO sensors time to stabilize after the disturbance possibly caused by the wiper. This delay is suitable if 'Default' filtering was selected in KOR. You can decrease this value if 'rapid' or 'accelerated' was selected.
- Get the data from the EXO with the time stamp of that moment (in this case 30+27+25 = 82 seconds after power on, so for example 13:41:22).
- Switch off the power to the DCP.

You can modify the example for any interval of 5 minutes or larger. For shorter intervals, you better use example 2. You can also change the sample interval to start 82 seconds earlier, so the time stamps will be on (or close to) round values (like 13:40:00). And maybe use 'cron_offset' to get it exactly right. See section 6.2 for details.

Note that the wiper_interval must specify time points that coincide with sample time points. But you don't have to wipe at each sample, so this would also work: `wiper_interval: "0, 0,20,40 * * *"`

Example 2: one sonde with a sample interval of 1 second and averaging.
If the interval becomes smaller than 5 minutes, it is no use (and in some cases not possible) to keep switching the power to the DCP and EXO sonde on/off all the time.

```
EXO:
- id: EX1
  port: serial1
  mode: RS232
  baudrate: 115200
  sample_interval: "* * * * *"
  supply_port: 1
  supply_port_always_on: True
  wiper_interval: "0 0,10,20,30,40,50 * * *"
  delay_after_wipe: 0
  average_interval: "0 * * * *"
  minimum_required_samples: 12
```

In this example, the sample interval is every second. The power port is always on. The wiper will wipe every 10 minutes.

Note that the baud rate is set to a much higher value than the usual 9600 Bd. Thus make sure the DCP is set to this higher value as well. See 11.1.2.

During the wipe, no samples will be read by the logger. After the wipe, the logger will immediately start reading data (`delay_after_wipe: 0`), unless you specify a delay_after_wipe value. Note that the EXO sensors can do internal averaging in different modes (rapid, accelerated, default). Refer to the EXO manual. This means that a disturbance caused by the wiper may have an effect on the sensor data for several seconds after the wipe has finished. So you may want to tune delay_after_wipe to the filtering mode of your sensors. Refer to the EXO manual for the filter times.

If you include the last two lines, the logger will not store every data sample, but only the 1-minute average values. See 6.2.

11.2 manta:

The manta driver is a dedicated driver for Manta™ sondes from Eureka. I assume that you are familiar with these sondes.

To use the Manta sonde, you need to:

- Prepare (configure) the sonde.
- Connect the sonde to the logger.
- Add the 'manta:' block to the config file.

These steps are described in the following sub-sections.

11.2.1 Preparing the manta sonde

Before connecting to the logger, make sure you configured the sonde as desired. Refer to the documentation from Eureka.

11.2.2 Connecting the manta to the OMC-048

The communication uses RS232 at 19200 Bd. To connect the manta to the logger:

- Choose any of the 4 available serial ports on the logger. Connect TX of the sensor to RX of the logger and vice versa. Signal levels are RS232.
- Connect the GND of the sensor to a logger-GND nearest to the serial port.
- You can power the sensor from a logger power output port (this is recommended if you have enough power ports available), but you can also power the sensor directly from a suitable power supply.

After connecting the sonde to the logger, you may use the same procedure as described for the EXO (section11.1.2) to make a port redirection and check the communication. In this case, assuming you connected the Manta to serial port 1 and power port 1, the REPL commands would be:

These are the REPL commands to switch Power Port 1 on and to redirect serial 1:

```
import omc048
p = omc048.power_supply(1)
p.on()

s = omc048.serial(1)
s.init(19200, s.RS232)
s.redirect()
```

(Control-C ends the redirection and p.off() switches off the power port.)

You can issue commands like VER: or READ. Refer to the Eureka ASCII command guide. Once you confirmed the wiring and the communication, you can continue by completing the config file.

11.2.3 Adding the Manta driver to the config file

Add below text to the config file.

This example is for serial port serial1 and Power Port 1.

```
manta:
- id: manta1
  port: serial1
  baudrate: 19200
  sample_interval: "0 0,10,20,30,40,50 * * *"
  response_timeout: 60
  supply_port: 1
  supply_port_always_on: False
  wiper_interval: "0 0,10,20,30,40,50 * * *"
  delay_after_wipe: 10
```

As with the EXO, the wiper time-points must coincide with the sample time points, but you don't need to wipe at each sample. So this would also work: `wiper_interval: "0 0,20,40 * * *"`

Note that these setting assume that the Manta is able to give data within 60 s after power on. You may want to add a 'startup_time'. If you want to sample at a short interval, you need to keep the sensor powered (`supply_port_always_on: TRUE`).

11.3 NEP5000:

This driver is dedicated to the NEP-5000 turbidity sensor. It is suitable for sensors with either RS232 or RS485 output.

Note: NEP-5000 sensors can also be equipped with an analog voltage, analog current or SDI-12 interface. If you want to use any of these, you can use the corresponding (generic) driver. This section only applies to the RS232 or RS485 interface.

To use the NEP5000 driver, you need to:

- Prepare (configure) the sensor.
- Connect the sensor to the logger.
- Add the 'NEP5000:' block to the config file.

This steps are described in the following sub-sections.

11.3.1 Prepare the NEP-5000

If the sensor has not been configured for you by Observator, you may need to change its configuration. In that case, refer to the NEP-5000 manual. You will need the configuration kit ('Blue box') and the NEP-5000 configuration software.

The sensor should be set to streaming (or free flow) mode and to perform a wipe at power-on. Set the baud rate to 9600. Make note of your choice for either RS232 or RS485/422 (recommended). The output sentence should be set to "#,Sensor_ID,NTU,Temperature". You can also leave out the temperature. At power on, the logger will adapt to either option. Set the data interval to, e.g. 5 seconds.

The statistical output from the NEP-5000 is not supported by the NEP5000 driver.

Note that the temperature output of the NEP-5000 is just an approximate indication of the temperature of the internal electronics, and NOT an accurate measurement of the water temperature, unless you purchased the ‘NEP-TEMP’ option for this sensor.

11.3.2 Connect the NEP-5000 to the OMC-048

The sensor can have either a subconn connector or a glanded cable (NEP-CBL). If the sensor has a subconn connector, you need a cable with a mating connector. Observator delivers two types of cable with this sensor: NEP-CBL (grey) or the thicker and more expensive OMC-497 cable (black). These cables have different colors for the wires. If you use a subconn on the other end of the cable as well (the logger-side of the cable), we deliver a mating chassis-part part with colored wires. Below table shows the coloring scheme for each of these options. When in doubt, always use an Ohm meter to check the pinning and/or check the NEP-5000 documentation.

NEP-5000 connections for various cables and connectors used frequently by Observator. This table does NOT apply to sensors with SDI-12 !

NEP-5000 Subconn pin #	NEP-CBL wire colors	OMC-497 cable wire colors	Subconn chassis part	Souriau UTS710E6S chassis part	NEP-5000 Signal
1	Brown	Black	Black	B (white)	Power, +12 to +24 V
2	Green	White	White	A (black)	GND
3	Pink	Red	Red	D (green)	RS232 TX or RS485+
4	Blue	Green	Green	C (red)	RS232 RX or RS485-
5	Grey	Orange	Orange	E (orange)	Calibration/wiper
6	White	Blue	Blue	F (blue)	Depends on order code.
	Yellow				

To sensor needs to be connected to the logger for power and for either RS232 or 485 (depending on your sensor configuration.

Power connection: Select any serial port; connect sensor GND to the logger GND nearest to the serial port; connect the sensor Power to one of the 12V Power Ports.

For RS232: connect the sensor’s TX to the logger RS232_RX. There is no use in wiring for bidirectional communication.

For RS485: Connect the sensor’s RS485- to the logger’s RS485_A and RS485+ to the logger’s RS485_B Note that, even though RS485 is bidirectional, we only use it unidirectional. (Because the sensor is set to streaming mode and not to polled mode.)

11.3.3 Add the NEP5000 driver to the config file

Example configuration for the NEP-5000:

```
NEP5000:
- id: NEP5000
  port: serial2
  mode: RS485
  sample_interval: "0 * * * *"
  supply_port: 2
  supply_port_always_on: False
```

If the sensor is configured for RS232, you obviously should select '`mode: RS232`'.

11.4 navilock_md6: (Alias 'Gps:')

This is a dedicated driver for multi-GNSS receivers using the NMEA protocol. It has been tested only with the Navilock NL-8022MP receiver, but it will probably also work with other NMEA GNSS receivers.

This manual, as well as the online documentation, use the name 'navilock_md6:'. It refers to the same driver code as the name 'Gps:'. So you can use them interchangeably.

The driver supports the reading of several NMEA sentences and reads a large number of parameters from them. These include the Lat/Lon position and the number of received satellites, as well as the time information. The time information can be used to synchronize the logger's internal clock.

To use the GPS receiver, you need to:

- Prepare (configure) the receiver.
- Connect the receiver to the logger.
- Add the 'navilock_md6:' block to the config file.

These steps are described in the following sub-sections, based on the NL-8022MP receiver.

11.4.1 Prepare the NL-8022MP receiver

If you purchase the receiver from Observator (OMC-ANT-G09), together with item OMC-ANT-G09**C** (description: '*The OMC-ANT-G09 will be configured for use with OMC data loggers.*'), you do not have to bother about this preparation. You can continue with the next sub-section.

In all other cases, you need a special USB cable (OMC-ANT-G09U) and you need to install the u-center software from Navilock.

The driver supports the reading of TXT, GGA, VTG and RMC sentences. Other sentences should preferably be switched off. Also, for the 8022MP receiver it is important to switch off the initial text messages that it will otherwise output after power-on. Finally, the baud rate should be set to 9600 Bd (or at least correspond to the setting in the config file). You can waste a lot of time in trying to figure out how to do all these things, so I highly recommend having Observator do this (OMC-ANT-G09**C**).

11.4.2 Connect the NL-8022MP receiver to the OMC-048, SN<xx0200

OMC-048 loggers with serial number till 048000200 have 4 output power ports for 12 V, but the GNSS receiver needs 5V. Therefore you need the converter OMC-ANT-G09DC. For higher serial numbers, see the next section.

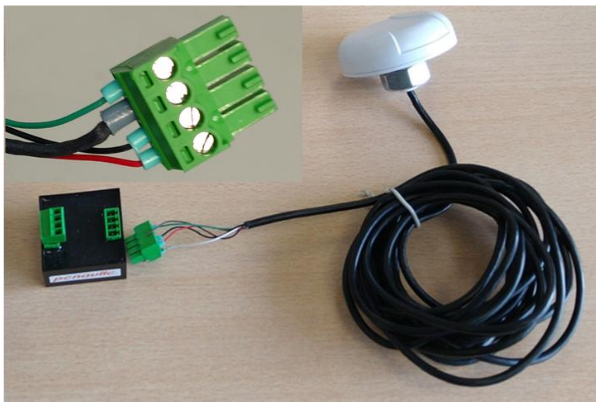


Figure 16. Connection of the receiver to the converter with (left) and without (right) the grey converter cable.

The receiver has a male MD6 connector. The converter is delivered with a grey cable with a female MD6 connector. You can either use the grey cable with the mating connector, or you can cut-off the connector from the black receiver cable (make sure you first completed section 11.4.1!). Figure 16 shows pictures of both cases.

Below table shows the connection from the receiver to the converter. You can either use a serial port (the table uses serial 4 as an example), or the NMEA port.

Connecting the receiver to the converter.

MD6 pin #	Color in receiver cable	Function (receiver)	Color in grey converter cable	Pin label on converter
1	Black	GND	Red	5Vdc GND/-
2	Red	VCC	Orange	5Vdc +
3	-	n.c.	-	
4	White	RX	-	
5	Green	TX	Black	GPS RX
6	-	n.c.	-	
-	shield	-	shield	GPS -

The connections from the converter to the logger using Power Port 4 and serial 4 <u>or</u> NMEA.

Converter pin label	Logger pin (port 4)	Logger pin (NMEA)
NMEA OUT + TX	RS232 RX (pin 48)	NMEA_A (pin 27)
NMEA OUT - GND	-	NMEA_B (pin 28)
12-24 Vdc -	GND (pin 49)	
12-24 Vdc +	12V DC supply (pin 10)	

Once connected, you can use port redirection to see if you receive the messages from the receiver. Refer to section 14.2.

11.4.3 Connect the NL-8022MP receiver to the OMC-048, SN>xx0200

OMC-048 loggers with serial number above 048000200 have 3 ports for 12V, but also one for 5 V (power port 4). The receiver needs 5V, so you can use port 4 without converter. For lower serial numbers, see the previous section.

The receiver has a male MD6 connector. For connection to the logger, either cut-off the connector, or use a mating connector. For use with certain logger housings, Observator may replace the connector by a Souriau connector.

Below table shows the connection from the receiver to the logger. It also indicates the wiring in case of a logger housing with a Souriau chassis part.

Connecting the receiver to the logger (*serial4* as example).

MD6 pin #	Color in cable	Function (receiver)	Souriau UTS78E4S chassis	Logger (*serial 4*)	Comments
1	Black	GND	A (black)	GND (*49 or 50*)	GND closest to the serial port.
2	Red	VCC	B (white)	Power Port 4, pin 10	5Vdc +
3	-	n.c.		-	
4	White	RX	D (green)	232_TX (*45*)	Data to receiver
5	Green	TX	C (red)	232_RX (*48*)	Data from receiver
6	-	n.c.		-	
-	shield	-			

Once connected, you can use port redirection to see if you receive the messages from the receiver. Refer to section 14.2 for port redirection.

11.4.4 Add the 'navilock_md6:' block to the config file

To use the GNSS receiver, include below text in the config file. This example assumes you use serial port 4 and Power Port 4.

```
navilock_md6:         (or use Gps:)
- id: GPS
  mode: RS232
  port: serial4
  baudrate: 9600
  sample_interval: "0 0 * * *"
  startup_time: 60
  response_timeout: 300
  supply_port: 4
  supply_port_always_on: False
  utc_time_sync: True
```

In this example the logger will power-on the receiver every hour (on second 0 of minute 0 of every hour). It will start reading the messages from the receiver after 60 seconds, and keeps reading till it receives a valid location (fix), or until it times out after 300s. Then the power will be switched off. Thus, the receiver can be on from anything between one second (in the unlikely case it immediately has a fix) till 300+60 s.

The longer you set the response_timeout, the more likely you are to get a fix. You could also keep the supply on all the time, but this will consume a bit more power. But you will be able to sample at an interval as short as 1 second.

A smaller startup_time may reduce power consumption, because you may get a fix within the startup time. A larger startup_time may increase the accuracy of the fix, because you don't accept the first fix (that may include just a few satellites), but wait a bit longer for a potentially better fix.

Also note the last line of above example: `utc_time_sync: True`. This means the logger's internal clock will be synchronized to the GNSS on each valid sample it receives. As with synchronization to an NTP time server, the logger will not sync if the internal clock is off by less than 2 seconds.

Default all values generated by the GNSS receiver are logged. If you set ‘sensor_data_print: True’, you can see all parameters and their names in the REPL. Add below text to the config file to log only the three listed parameters.

```
Log_parameters:
  GPS:
  - Latitude
  - Longitude
  - Satellites
```

11.5 gmx_501:

The gmx_501 driver is a dedicated driver for the MaxiMet™ GMX501 ‘Compact Weather Station’ *with* internal GPS, from Gill Instruments. Other MaxiMet sensors *without* GPS (including the GMX501 *without* GPS), can be handled by the ‘generic_serial_input’ driver (section 12.1). The Gmx_501 driver uses the generic_serial_input driver. Therefore this section is relevant for the whole Maximet range, even if you don’t use the Gmx_501 driver, but only the generic_serial_input driver.

I assume that you are familiar with MaxiMet sensors. You may need to consult the MaxiMet manual.

The gmx_501 driver can handle the GPS sensor of the GMX501. The driver can interpret the Lat/Lon as well as the date&time from GPS. It can even synchronize its internal clock to the GPS clock. In fact, this is the only reason for having the Gmx_501 driver, because all other things are handled by the generic_serial_input driver. The generic_serial_input driver itself cannot interpret the GPS data.

Note:
If the sensor does not get a GPS fix, its output string will not contain GPS data and the logger will ignore the complete string. So you will not get wind-data if you don’t have a GPS fix! This could happen when testing inside a building, or when the startup time is set too short for the sensor to obtain a fix.

Communication is over RS232 or 422 (not SDI-12) with the GILL-protocol.

To use a GMX sensor with the OMC-048, you need to:

1. Prepare (configure) the GMX
2. Connect the GMX to the OMC-048
3. Create a customized ‘serial_input_reference_tables.py’ file
4. Add the gmx_501 driver to the config.txt file

These steps are detailed in the following sub-sections.

11.5.1 Prepare (configure) the MaxiMet

You may need the MetSet software from Gill Instruments to modify or check the settings.

The sensor must be configured to:

- Gill protocol (default)
- RS232 (default) or RS422 (I recommend RS422)
- 19200 Bd (default) or 9600 Bd
- Output rate once per second(default)/minute/hour

Make note of the settings you made, because they have to correspond with the settings you are going to make in the config file.

Next you have to select the parameters that you want the sensor to output. The default format of a GMX501 with GPS is (refer to the MaxiMet manual):
Node, Relative Wind Direction, Relative Wind Speed, Corrected Wind Direction, Corrected Wind Speed, Pressure, Relative Humidity, Temperature, Dewpoint, Solar Radiation, GPS Location, Date and Time, Supply Voltage, Status, Checksum.

Example:
Q,310,000.04,033,000.59,1032.1,040,+020.6,+006.7,0001,+50.762988:-001.539893:-0.80,2015-06-09T09:24:34.9,+05.1,0004, 3D

You may change this for your requirements. Other GMX models will have other strings

Note:
It is important that you know exactly what the output string is. You have to create a 'serial_input_reference_tables.py' file that corresponds to the output string.

For the **gmx_501 driver** it is essential that the *GPS Location* and *Date & Time* are on position 11 and 12 (counting from 1 for 'Node'), as shown in the default string. So you can change the string as you like, but do not change the location of these two parameters in the string. (Or you have to change the gmx_501.py file accordingly.)

Note:
Make sure you noted/saved the exact configuration of the sensor for later use. If, at some future time, you need to replace the sensor with a new one, you need to make sure it is configured exactly the same. (Or you need to reconfigure the logger for the new situation).

11.5.2 Connect the Maximet to the OMC-048

Connection is straight forward. Use any of the 4 serial ports and connect:

- For RS232: the sensor's RS232/TXD to the logger's RS232_RX and vice versa;
- For RS422: the sensor's RS-422/TXD+ and TXD-- wires to the logger's RS422_RXA and RS422_RXB wires. (The polarity with RS422 is always a 50% change.)

You can also use the NMEA port instead of one of the four serial ports.

For power, connect the ground wire to a ground pin close to the chosen serial input. For power, you can either use one of the 12V power output ports, or connect directly to a power supply. You can use this latter option if you want the sensor to be powered all the time anyway.

Once connected, you can use port redirection to see if you receive the correct messages from the sensor. Refer to section 14.2. If you are using RS422, you need to redirect using 's.init(19200, s.RS**422**)'.

11.5.3 Create a customized 'serial_input_reference_tables.py' file

Please refer to chapter 12 and specifically section 12.1 for an explanation on generic drivers. You need to have read that explanation before you can fully understand this sub-section.

As mentioned before, the gmx_501 driver uses the ‘generic_serial_input’ driver and only adds the interpretation of the GPS location and date&time to it. If you don’t have GPS, you should use the generic driver on its own. The gmx_501 driver uses the GMX501 table in the ‘serial_input_reference_tables.py’ file. If you use the ‘generic_serial_input’ driver directly, you can define a new table with a name of your choice.

The reference table needs to be adapted to the specific output you selected for the sensor. If you use a GMX501 with GPS and the default output string (as shown in section 11.5.1), the example GMX501 table in ‘serial_input_reference_tables_template.py’ is correct. But it’s safer to double check. If you change the table because your sensor does not output the default string, make sure the *GPS Location* and *Date & Time* stay on position 11 and 12, because that is where the GMX501 driver expects them.

Note that, as always with generic drivers, it is your responsibility to make sure that the reference table matches exactly the sensor output.

11.5.4 Add the gmx_501 driver to the config.txt file

I will give two examples for adding the gmx-501 driver to the config file. Obviously, you need to adapt them to the actual connections you made, as well as to the settings you made in the sensor.

If you use the ‘generic_serial_input’ driver directly (and not the gmx_501 driver), refer to section 12.1.2.

Example 1: Fast sampling
This example assumes you are using port serial1 and Power Port 1.

```
gmx_501:
- id: gen0
  port: serial1
  mode: RS422
  baudrate: 19200
  supply_port: 1
  sample_interval: "* * * * *"
  response_timeout: 2
  utc_time_sync: True
  supply_port_always_on: True
```

Note that this example keeps the power on and samples at a 1 second interval. You could also power the sensor directly from a power supply, to keep the Power Port free for other purposes.

Example 2: Slow sampling
If you use a longer interval, you could also switch the power on/off each sample. Below example uses an interval of 15 minutes. It is suitable for sampling averaged parameters from the sensor.

```
gmx_501:
- id: gen0
  port: serial1
  mode: RS422
  baudrate: 19200
  supply_port: 1
  sample_interval: "0 5,20,35,50 * * * "
  startup_time: 600
  response_timeout: 10
  utc_time_sync: True
  supply_port_always_on: False
```

After switching on the power, the logger will wait for 600 seconds (startup_time), to give the sensor time to produce valid average values (you should check this with the sensor settings and the sensor manual). AND to obtain a valid GPS fix. So actual sampling will occur on minutes 0, 15, 30 and 45 of every hour.

If the GPS did not get a fix in the 600 seconds (10 minutes) startup_time, only the other sensor data will be logged. The GPS will be missing. Obviously, a longer startup_time gives more change on a GPS fix, but also uses more energy. Note that the driver will not wait for a fix, like the navilock_md6 (GPS) driver (section 11.4) does. It will finish after receiving the first sensor data (with or without GPS) after expiry of the startup_time.

Note that both examples use ‘`utc_time_sync: True`’. This works the same as for the GPS (navilock) driver, see section 11.4.4. If you use synchronization with GPS, there is no need to sync to an NTP time server as well.

Finally, you can use the Log_parameters block to exclude some parameters from being logged. Alternatively, you can also remove (or ‘comment out’ with ‘#’) the corresponding lines from the generic table.

11.6 nortek_cp:

This driver is written for the Nortek™ Aquadopp and AWAC current profiles. It supports reading the current profile data in binary mode. It ignores wave data, if present.

To use the nortek_cp: driver, you need to:

- Prepare (configure) the Aquadopp or AWAC.
- Connect the instrument to the logger.
- Add the ‘nortek_cp:’ block to the config file.

11.6.1 Prepare (configure) the Aquadopp or AWAC

You need the configuration software (‘Aquapro’ or other) supplied by Nortek to make this configuration.

Make sure the sensor is configured at the same baud rate as set in the config file of the OMC-048. Also make sure the instrument outputs the desired current profile data in BINARY mode. Preferably do not output possible wave data, as it will be ignored. Make note of the interval you selected for averaging (for example: 30 minutes).

The Aquadopp measurement must be started with the “ST” command in the terminal window of the “Aquapro” user-interface. The user interface will give an “Ack Ack”, letting know the sensor is set to measurement mode. Do not stop the measuring mode!

11.6.2 Connect the instrument to the logger

Connect the power from the instrument directly to the power supply. Thus the logger will not switch the sensor power on/off.

Connect the RS232/422 lines to one of the serial ports. Use the ground line nearest to the port.

Because data from the instrument is binary, you cannot (in an easy way) use port redirection.

11.6.3 Add the ‘nortek_cp:’ block to the config file

Include below text in the config file. This example uses port 3 at 19200 Bd. Set the sample interval to correspond to the averaging interval of the instrument, in this example 30 minutes. Make sure the response_timeout is set to a slightly larger value, for example 31 minutes (1860 seconds). Note that the internal clock of the instrument and the clock of the logger are not synchronized, so they do not run 'in phase'. Make sure the RX buffer is large enough to hold the binary file (rxbuf in bytes).

```
# ----Sensor-Settings---- #
nortek_cp:
- id: aqdp1
  port: 3
  baudrate: 19200
  rxbuf: 1024
  sample_interval: "15 0,30 * * *"
  response_timeout: 1860
```

11.7 ysi_6_series:

This driver is written for the 6-series water quality sensors from Xylem™. This very successful family of sensors is now retired and has been replaced by the EXO series (section 11.1). However, the 6-series is still in use and is fully supported by this driver. Please contact Observator if you want to use this driver.

11.8 modbus_opus:

This driver is specifically for the OPUS sensor from TriOS. It uses the generic 'modbus' driver. If you use the OPUS together with a wiper, refer to chapter 13.

11.8.1 Prepare (configure) the OPUS

As always, before you connect a sensor to the OMC-048, make sure the sensor is configured correctly. In particular, the sensor should not -self trigger, because the modbus_opus driver will issue a 'trigger' -command to start each measurement.

Also select -and note- the serial port settings (baud rate) and the modbus slave address..

11.8.2 Connect the instrument to the logger

Connect the serial (RS485) wires and the ground to the OMC-048. If you want to have the power switched by the logger, note that the sensor can draw current peaks that exceed the limit of the power outputs of the logger. It is therefore necessary to use either the internal relays, or an external relay driven by one of the power ports.

If you use the OPUS together with a wiper, refer to chapter 13.

11.8.3 Create a customized reference table

The modbus_opus driver uses the generic modbus driver. This means you have to customize the reference table for your particular sensor. Please refer to section 12.4 for details. You can obtain an example reference table from Observator.

11.8.4 Add the 'modbus_opus:' driver to the config file

Below configuration example assumes that:

- After power-on, the sensor needs to warm-up for 12 seconds before it can receive a measurement command. The correct warm-up time may depend on the model and the configuration, so you should verify this for your particular sensor. Or use trial-and-error.
- After issuing the trigger (=write to a particular register) for starting a measurement, it takes 25 seconds before the measurement is ready. This number highly depends on your configuration. The 25 s may

be correct for an OPUS measuring only a few parameters. For an OPUS with a lot of parameters, it may be more than a minute. So you should verify the correct delay for your particular sensor, or use trial-and-error.

- You use an external relay to power the sensor. Alternatively, you can use an internal relay. Then replace ‘supply_port’ with ‘relay_port’ (section 6.2).

Example configuration:

```
modbus_opus:
- id: Opus
  supply_port: 2
  supply_port_always_on: False
  startup_time: 12
  sample_interval: "0 0,5,10,15,20,25,30,35,40,45,50,55 * * *"
  port: serial2
  baudrate: 9600
  rx_buffer: 1024
  mode: RS485
  protocol: RTU
  reference_table: TriOS_OPUS
  measurement_wait_time: 25
  response_timeout: 85
```

As mentioned before, make sure the `measurement_wait_time` is correct for your particular OPUS configuration. The 25 s is just an example!

After the measurement_wait_time has expired (25 s), the logger will read the parameters from the sensor. If the read fails (timeout after 1 s), the logger will retry. It will send the trigger to start a measurement again, wait again for 25 s and try to read again. It will keep retrying, till the response_timeout has expired. In this example, there is time for 3 retries (3 x 26 s = 78 s, which fits in 85 s).

You can (temporarily) use ‘`register_print: True`’ and ‘`sensor_data_print: True`’ to verify if the reading of all parameters is done correctly.

Side note: You may wonder why this driver doesn’t use the ‘measurement timeout’ register of the OPUS to determine when the measurement is ready. The reason is that this register didn’t work well during our testing.

11.9 modbus_spectrolyser:

This driver is specific for the Spectrolyser V3 from s::can.

11.9.1 Configure the Spectrolyser

Refer to the documentation of the manufacture on how to connect to the instrument using your web browser. In short:

- Apply power to the sensor. The WiFi hotspot will be waiting for a connection for only 10 minutes (default setting).
- Connect you phone of PC to the WiFi hotspot SP3xxxxxxx (serial number of the sensor). The password is 'spectrolyser'.
- In your browser, visit 192.168.43.1
- Logout and login as 'expert' with password 'scan'.

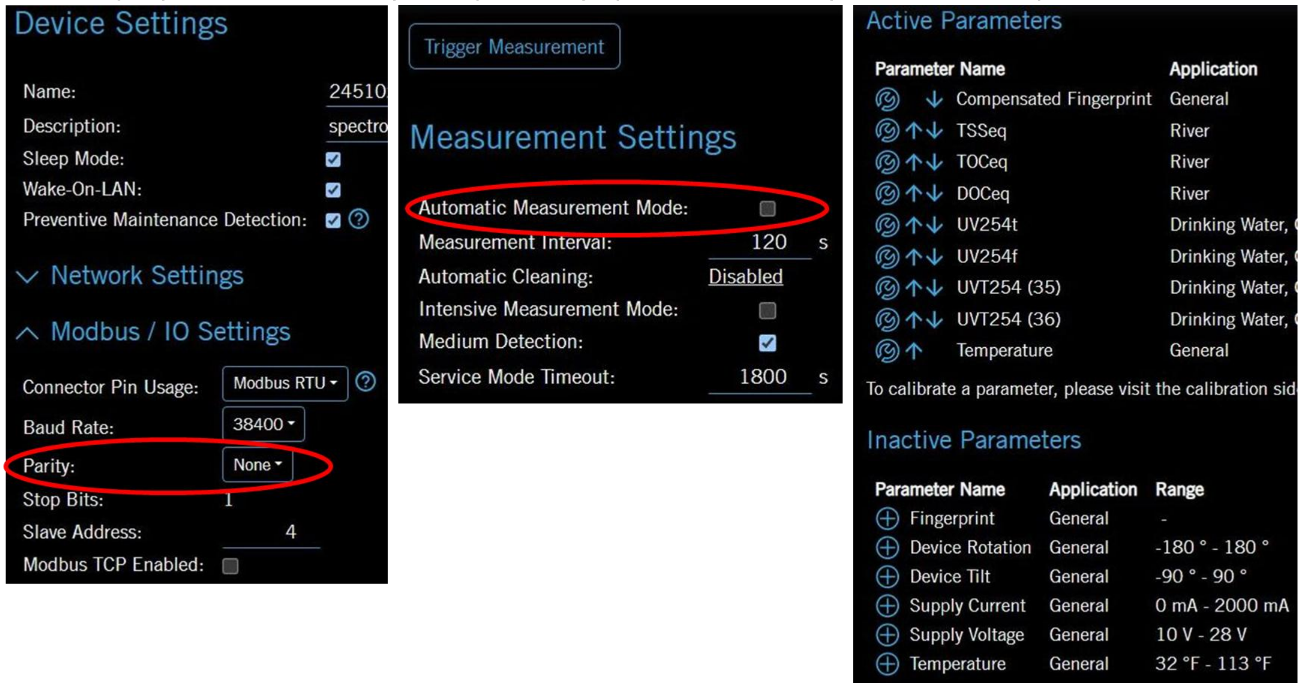
- Go to the tab 'Service' > Device settings; got to 'service mode' and click ‘edit settings’.
- Change the modbus IO settings to Parity: None

Take note of the spectrolyser settings. The factory defaults are:
- modbus address: 4;
- baud rate: 38400;
- parity: Odd; For FW_1.03B2803, only parity None is supported!
- Mode: RTU.

Make sure the sensor is set to:
- use Modbus (IO settings) with Parity: None
- is on ‘manual measurement mode’ and not on ‘automatic’. (The ‘manual’ mode allows the logger to control if and when measurements should be made.)
- sleep mode is active.

Finally, select which parameters you want to measure by making them ‘active’ parameters. Take note of the order in which they are listed. Figure 17 shows an example of the relevant settings. You will need this listing in creating a customized reference table (section 11.9.3).

Note: According to the spectrolyser manual, the sensor can handle only 8 parameters on Modbus. So select only 8 parameters. Plus, optionally, the fingerprint, which is not part of the max 8 parameters.



Device Settings

Name:	24510
Description:	spectro
Sleep Mode:	✓
Wake-On-LAN:	✓
Preventive Maintenance Detection:	✓

Network Settings

Modbus / IO Settings

Connector Pin Usage:	Modbus RTU
Baud Rate:	38400
Parity:	None
Stop Bits:	1
Slave Address:	4
Modbus TCP Enabled:	☐

Measurement Settings

Automatic Measurement Mode:	☐
Measurement Interval:	120 s
Automatic Cleaning:	Disabled
Intensive Measurement Mode:	☐
Medium Detection:	✓
Service Mode Timeout:	1800 s

Active Parameters

Parameter Name	Application
Compensated Fingerprint	General
TSSeq	River
TOCeq	River
DOCeq	River
UV254t	Drinking Water,
UV254f	Drinking Water,
UVT254 (35)	Drinking Water,
UVT254 (36)	Drinking Water,
Temperature	General

To calibrate a parameter, please visit the calibration sid

Inactive Parameters

Parameter Name	Application	Range
Fingerprint	General	-
Device Rotation	General	-180 ° - 180 °
Device Tilt	General	-90 ° - 90 °
Supply Current	General	0 mA - 2000 mA
Supply Voltage	General	10 V - 28 V
Temperature	General	32 °F - 113 °F

Figure 17. Example of a configuration of a spectrolyser.

11.9.2 Connect the instrument to the logger

You can connect to any of the 4 serial ports of the OMC-048, pins A and B (RS-485). Refer to the documentation of your spectrolyser to find the right pins/wires. In our test case:

Spectrolyser:	**OMC-048**
Brown(7) – RS485+	485-B
Blue(5) – RS485-	485-A

Because the sensor can draw peak-currents that are too high for the power ports, it is recommended to connect the instrument directly to a power supply. So it will always be powered. It is also possible to power the sensor through one of the relay ports. In that case, the wiring using relay port 1 (example) is:

Spectrolyser:	**OMC-048**
Black(8) – Ground	GND
White(6) - +12V Power	relay pin 11. And connect a power supply to relay pin 12

Note that the manufacturer recommends that power is not switched on and off all the time. Instead, the logger will send a 'sleep' command to the sensor after each measurement. This will send the sensor to sleep until the next measurement is due.

For battery-powered application, the advantage of using the relay port, is that the power_monitor can be used to switch off the sensor's power in case the battery runs empty.

In our test case, we found the following current consumption at 12 V supply voltage:

- Between 100 mA and 200 mA when stand-by. Average 150 mA. Also during start-up procedure (70 s).
- During a measurement, about 450 mA during 5 seconds.
- In sleep mode about 7 mA, but higher peaks during LED flashes. Average about 10 mA.

If you also want to connect a wiper, refer to section 13.2.4.

11.9.3 Create a customized reference table

The modbus_spectrolyser: driver uses the generic_modbus: driver, which in turn uses the 'modbus_reference_tables.py' file. This file must be customized. So, as always, copy 'modbus_reference_tables_template.py' to 'modbus_reference_tables.py'. Remove the tables you don't need, and create a table like this:

```
# Make sure this matches with the selection made in the spectrolyser.
P =  {
  #0: slave_addr, register_type, start_register, nr_of_registers
  0: (0x04, register_type.INPUT_REGISTER, 128, 64),
  #x: name, unit, tag, start_addr of chunk, format, factor, offset
  1: (("TSS", "mg/l", "TSS") , 2 , formats.FLOAT, 1, 0),
  2: (("TOC", "mg/L", "TOC")  , 10, formats.FLOAT, 1, 0),
  3: (("DOC", "mg/L", "DOC")  , 18, formats.FLOAT, 1, 0),
  4: (("UV254t", "abs/m", "UV254t"), 26, formats.FLOAT, 1, 0),
  5: (("UV254f", "abs/m", "UV254f")  , 34, formats.FLOAT, 1, 0),
  6: (("UVT35", "", "UVT35")  , 42, formats.FLOAT, 1, 0),
  7: (("UVT36", "", "UVT36")  , 50, formats.FLOAT, 1, 0),
  8: (("Temp", "C", "Temp")   , 58, formats.FLOAT, 1, 0)
    }
```

Notes:

- This table assumes that you use slave address 4. Change if you like.
- Make sure the parameter and tag names in this table correspond to the selection you made in the spectrolyser, e.g. Figure 17 right-most screenshot.

As you may notice, this table does not include the 221 spectral lines (fingerprint) that the specrolyser also calculates. Because there is a modbus limitation of 125 registers (62 floats) in one message, the reading of the fingerprint (221) floats must be split over 4 tables. For example, with 55+55+55+56 = 221 floats. With this many parameters, the limit of the OMC-048 internal RAM (memory) is almost reached. You can create the tables yourself, in a manner similar to the ‘P’ table above. But it is easier to ask Observator to give you an example ‘modbus_reference_tables.py’ file, specifically for the spectrolyser.

You cannot have multiple spectrolsers on the same driver.

11.9.4 Add the ‘modbus_spectrolyser:’ driver to the config file

To add the 'modbus_spectrolyser:’ driver, add below text to the config file (example):

```
modbus_spectrolyser:
- id: s
  port: serial2
  baudrate: 38400
  mode: RS485
  rxbuf: 1024
  protocol: RTU
  relay_port: 1
  relay_port_always_on: True
  startup_time: 70
  sample_interval: "35 9,19,29,39,49,59 * * *"
  measurement_wait_time: 25
  reference_table: P,A,B,C,D
  spect_sleep_duration: 555 # 600-25-10-10 =
  response_timeout: 10
```

Some notes:

- Make sure the ports settings (including baudrate and parity) match the settings in sensor. The default parity setting in the driver is None; the default of the spectrolyser is Odd. This version of the FW only supports ‘None’.
- If the sensor’s active parameters include the whole spectrum (‘fingerprint’), the RxD buffer size should be increased to the shown value (rxbuf: 1024).
- The example assumes that the relay port is used to switch the power continuously on. This can be useful in combination with the power_monitor. See section 11.9.2.
- The startup_time is only relevant after power-on. When the logger application starts, it will switch-on the relay, wait for the startup_time and then try to communicate with the sensor.
- After sending a ‘measurement command’, the logger will wait for ‘measurement_wait_time’ before attempting to read data. Make sure this ‘measurement_wait_time’ is at least the time your sensor needs for a measurement.
- For the reference tables, refer to the preceding section.
- After receiving the data from the sensor, the logger will send the sensor to sleep for ‘spect_sleep_duration’. According to its manual, the sensor should wake up at least 10 s before receiving a command. To calculate the ‘spect_sleep_duration’, use this formula (between brackets the numbers from above example in seconds): spect_sleep_duration = sample interval (600) - measurement_wait_time (25) – sensor wake up time (10) – safety margin (10) = 555

11.10 modbus_seametrics:

This driver is specific for the ‘Smart Sensors’ from Seametrics. These are mostly PT (Pressure and Temperature) and CT (Conductivity and Temperature) or CTP sensors. They all have SDI-12 as well as Modbus interfaces and you can use either with the OMC-048.

The ‘specific’ thing about the ‘modbus_seametrics’ driver, is that it reads the data from the sensor twice. The first ‘read’ will result in old or invalid data which is discarded by the driver. But the ‘read’ will trigger a new measurement to be made by the sensor. The second ‘read’ (1.5 second after the first) will return the fresh data that will be logged. Refer to the Seametrics documentation for details.

The modbus_seametrics driver uses the generic modbus driver. This means you have to customize the reference table for your particular sensor. So please refer to section 12.4 for details.

To use this driver, include the following text in your config file (example):

```
modbus_seametrics:
- id: modbus_1
  protocol: RTU
  register_print: True
  sample_interval: "0 * * * *"
  supply_port: 1
  startup_time: 1
  port: serial1
  mode: RS485
  baudrate: 9600
  reference_table: PT12,CT2X
```

This example assumes that you have two sensors connected to serial port 1 and powered from power port 1, and that you customized the tables PT12 and CT2X for these sensors.

After switching on the power to the sensors (on second 0 of every minute), the driver will wait for 1 second (startup_time, configurable) before reading (stale) data from both sensors. After waiting another 1.5 second (fixed in the code), it will read the fresh data from both sensors.

11.11 modbus_wimo:

This driver is specific for the WiMo multiparameter sondes from NKE Instrumentation. To use these sondes with the OMC-048, follow these steps:

1. Prepare the sonde.
2. Connect the sonde to the logger. You can use serial 1,2,3 or 4. Obviously, you need to use the RS232 or RS458 pins, depending on what your sensor uses. Connect the power wire to one of the 12V power ports from the logger. (Or use a relay port.) Connect the ground wire to logger-ground.
3. In the ‘scrip/modules’ folder, copy the file “modbus_reference_tables_template.py” to “modbus_reference_tables.py”. You do NOT need to create a table for the sensor, like you would need to do for ‘generic’ modbus sensors; the specific modbus_wimo: driver already contains the table.
4. Add below text to the config.txt file. Make sure you set the address, port details, and the power port correct.

Example for the config file:

```
modbus_wimo:
- id: wimo
  address: 0x80
  wiper_interval: "0 * * * *"
  sample_interval: "0,10,20,30,40,50 * * * *"
  port: serial1
  mode: RS232
  baudrate: 9600
  protocol: RTU
  register_print: True
  supply_port: 1
  supply_port_always_on: True
```

After starting the logger application, study the REPL to check if all goes well.

A few notes:

- You can use multiple sondes on the same bus; just make sure they have different addresses and list these addresses in the config file like this: ‘address: 0x80, 0x81’.
- The parameter names will include the name of the driver as well as of the modbus address (wimo and 0x80 in above example).
- Each time the logger application starts, the driver will ask the sensor about its parameters and it will subsequently log these. Thus, if you add/remove sensors from the sonde, you don’t need to change anything in the logger; it will automatically detect the sensors when the application is started.
- The wiper_interval must coincide with the sample_interval. Refer the EXO (11.1) for more details.

11.12 Analog_voltage:

The logger has two analog voltage inputs:

- Pin 18 (0-24), named ‘1’;
- Pin 19 (0-5V) , named ‘2’.

Note that pin 17 is an output.

To use these, include this in the config file:

```
Analog_voltage:
- id: AV1
  port: 1
  sample_interval: "0,10,20,30,40,50 * * * *"
  min_in: 0
  max_in: 24000
  min_out: 0
  max_out: 24
  log_name: V1
  log_unit: V
  log_tag: V1
-id: AV2
  port: 2
  <some text left out here>
  log_tag: V2
```

If you want to use only one voltage input, you can leave out the ‘-id: AV2’ line and everything below it. Both ports should be given an unique id (here AV1 for port 1 and AV2 for port 2). Also, take care that you assign a unique log_tag (unique over all log_tag in the config file, not just for Analog_voltage).

To avoid confusion, it is wise to assign a unique log_name as well. Obviously, the log_unit need not be unique.

The port 1 sensor generates a mV value between 0 and 24000. You can convert this by a linear transformation into the desired range and give it an appropriate unit. The example gives an output in the range 0 to 24 V. But, suppose you connected a water level sensor for which 10 mV corresponds to 2.5 m water column and 16530 mV corresponds to 5.9 m water. In this case, you would use for example:

```
  min_in: 10
  max_in: 16530
  min_out: 2.5
  max_out: 5.9
  log_name: level
  log_unit: m
  log_tag: L1
```

If you want to power the connected sensor from the logger, you can include these lines (obviously modify to your wishes) per sensor (id):

```
  supply_port: 3
  supply_port_always_on: False
  startup_time: 10
```

In this case, the sample will be taken 10 s after the power is switched on.

See also the next section, because it shares a lot of similarities with this section (including this line).

11.13 Analog_current:

The logger has four analog current inputs: pins 23 to 26, named 1 to 4. To use any of these, include this in the config file:

```
Analog_current:
- id: AC1
  port: 1
  sample_interval: "0,20,40 * * * *"
  min_in: 0
  max_in: 24
  min_out: 0
  max_out: 24
  log_name: Cp
  log_unit: mA
  log_tag: C1
  supply_port: 3
  supply_port_always_on: False
  startup_time: 10
```

If you want to use several current inputs, you can add several times the '- id: AC1' line and everything below it. All ports should be given an unique id (here AC1 for port 1; AC2 would make sense for port 2). Also, take care that you assign a unique log_tag (unique over all log_tag in the config file, not just for Analog_current).

To avoid confusion, it is wise to assign a unique log_name as well. Obviously, the log_unit need not be unique.

Above example assumes that you connected the sensor to Power Port 3 and that you want to read the sensor's value 10 s after powering it on.

The current sensors generate a mA value between 0 and 24. You can convert this by a linear transformation into the desired range and give it an appropriate unit. Above example gives an output in the range 0 to 24 mA. But, suppose you connected a solar radiation sensor for which 4 mA corresponds to 0 W/m2 and 20 mA corresponds to 1000 W/m2, you would use, for example:

```
  min_in: 4
  max_in: 20
  min_out: 0
  max_out: 1000
  log_name: NetRad
  log_unit: W/m2
  log_tag: NR
```

See also the previous section, because it shares a lot of similarities with this section (including this line).

11.14 rain_pulse:

This driver is intended for a tipping-bucket rain gauge, but is also applicable in other cases where you want to count the pulses on one of the ‘digital input’ pins. There are two such pins: port 1 corresponds to pin 51 and port 2 to pin 52. Both pins have an internal pull-up resister to 3.3V. Thus you can connect a switch (like the Reed-relay of a tipping-bucket rain gauge) directly between one of the digital input pins and ground.

The falling edges will trigger the counter to increment by the amount given by *mm_per_pulse*. So you can set this value to 1 if you merely want to count pulses (instead of the amount of rain).

The value of the counter is sampled at the time points given by *sample_interval*. The counter can be reset as specified by the *reset_interval* (for example daily at midnight, as shown below.)

```
rain_pulse:
- id: yourchoice
  port: 1
  mm_per_pulse: 0.2
  reset_interval: None  # or eg. reset_interval: "0 0 0 * *"
  sample_interval: "second minute hour day week-day"
```

11.15 input_state:

This driver uses the same pins as the rain_pulse: driver. See the previous paragraph. Using this driver, you can log the input state of each pin. For example:

```
input_state:
- id: switch1
  port: 1
  state_invert: False
  sample_interval: False
  trigger_on_event: True
- id: switch2
  port: 2
  state_invert: False
  sample_interval: "0,10,20,30,40,50 * * * *"
```

- *sample_interval*: defines the cron interval on which to sample the input of the digital IO (state can be “0” of “1”). It’s an required parameter, which must be disabled by defining “False” when a sample interval is not applicable.
- *trigger_on_event:* can be set “True” when a sample is to be taken as soon as the state changes. The change is detected immediately, only the sample is processed as soon as the datalogger wakes up from sleep. This means this setting is not optimal for fast changing states.
- *state_invert:* is an optional parameter to invert the states. When set to “False” (default) = “0” is “open circuit”, “1” closed to GND. This is inverted when set to “True”

It is possible to use trigger_on_event in combination with sample_interval

Beware: With ‘state_invert: false’ a ‘high’ input level (open circuit) is logged as a ‘0’. A ‘low’ level (input connected to ground) is logged as a ‘1’. Remember that the pins have an internal pul-up to 3.3V, so open circuit results in a ‘high’ input level.

11.16 pulse_driver:

The power and relay ports are mostly controlled from sensor drivers. This is done using the “supply_port” or “relay_port” settings. However, if a port is not used by any of the sensors drivers, it can be controlled from below driver.

This driver can be useful for driving wipers, valves, pumps and so on. In most cases you will probably use either one power port or one relay port, but you can use both, as below example shows. The timing of the relay and the pulse port will be the same.

For a power_port, if:
`normally_off: True`, the port is normally off (no power) and is switched on for the pulse_duration. If `normally_off: False`, the port is normally on (power is applied) and is switched off for the pulse duration.

For a relay_port:
Similar to the power_port, where ‘no power’ corresponds to the relay state as is shown graphically on the front panel of the logger.

```
pulse_driver:
- id: wiper
  sample_interval: "0,20,40 * * * *"
  supply_pulse_port: 3
  relay_pulse_port: 2
  pulse_duration: 5
  normally_off: True
```

12 Generic drivers and reference tables

Generic drivers bring a lot of flexibility to the OMC-048. Of course, if there is a dedicated driver available for your particular sensor, it is always easier and better to use the dedicated driver. But it is impossible to make a dedicated driver for each sensor in the world. There are simply too many! And it is also useless to try to do so, because with generic drivers you can cover most of them anyway.

All generic drivers work in roughly the same way. Therefore I will explain only one of them completely. For the others I will only describe their specifics. So please read section 12.1, about ‘generic_serial_input:’ to get the general idea and then continue with the driver you need.

All generic input drivers accept parameters of an arbitrary type. Thus a parameter can be a number like ‘10’, or ‘-9.343’, but also a string like ‘NaN’ or whatever. The logger will not interpret this string, but simply write it to the data file ‘as-is’.

Note that in section 6.2 some ‘common settings for drivers’ were described. These are not repeated every time in the descriptions of the drivers.

12.1 generic_serial_input:

This driver is intended for sensors that provide a streaming output over RS232/422. This means that, after power on, the sensor automatically starts to produce data. The sensor should either produce a single data string (so take only one measurement after power on), or it should produce data periodically. All produced output strings should have an identical format with a fixed position for each parameter. Thus the string should not contain “name,value” pairs that can occur in arbitrary order.

The driver is not suitable for sensors that require a command before producing data. Communication for this driver is unidirectional: from sensor to logger.

To use a serial sensor with the generic_serial_input driver, you need to go through the following steps:

1. Configure and connect the sensor;
2. Add the driver to the config file;
3. Create a customized reference_tables file.

The steps are described in the following sub-sections.

12.1.1 Configure and connect the sensor

Some sensors may require no configuration at all, while others may offer you many choices in what data they produce and in which format or protocol. In any case:

- The sensor should produce the output parameters that you want, and preferably not much more. For example, if a sensor can produce a hundred different parameters, but you need only 5 of them, it is recommended that you configure the sensor to output only those 5.
- The sensor should produce its data automatically after power-on, either once or with a known (fixed) interval. Like data every second.
- The output of the sensor should be in this format:
  <start><value>{<separator><value>}<stop>   where {x} means 1 or more repetitions of x.
  For example $16.8,23.6\r\n
  where <start>=’$’, <separator>=’,’ and <stop>=’\r\n’. The two parameter values are 16.8 and 23.6.
  Another example: $,16.8,23.6\r\n
  where <start>=’$,’ and the rest is the same as above.

It is important that you know exactly what the output string looks like. You may need to consult the manual of the sensor. Once you configured the sensor, it is very wise to check the output string, to make sure it is exactly as you think it is, before going on with the next step. Manuals are not always as good as this one, so always check!

In order to check the output string, you can connect the sensor to the logger and use port redirection to make a terminal connection. Refer to section 14.2 for port redirection in general. Because the string may contain unprintable characters, you should put the terminal program into hexadecimal mode. Not all terminal program support this. Make sure you know what the <start> and <stop> characters (or strings) are, including possible unprintable characters. The <separator> is usually easier to spot, but sometimes there is more than one.

Important!
Make sure you save the configuration of the sensor on your computer. If, at some future time, you need to replace the sensor with a new one, you have to use the same configuration again. If you fail to do so, you either get the parameters mixed up (wrong results) or you have to figure everything out again! The sensor configuration MUST match the reference_table!

Once you know the exact format of the sensor's output string, and you have seen the data using port redirection, you can continue with the next steps.

12.1.2 Add the driver to the config file

To use this driver, include the below lines in the config file:

```
generic_serial_input:          # This means "serial_input_reference_tables.py" will be used
- id: gmx_a
  port: serial1
  reference_table: my_gmx      # This means that table 'my_gmx' will be used
  < text left out here>
```

You can chose any *id* you like, as long as it is unique. The *port* can be any of the 4 serial ports, plus the NMEA port. The *reference_table* is also a name that you can chose, but it must correspond to the name of the table in the Python file (explained next). It makes sense to name the *reference_table* after the type-name of the sensor. For example, 'EXO' or 'NEP5000'. And it makes sense to make the id uniquely identify the sensor. For example, if one sensor is used upstream from a dam and the other downstream. The id's could be 'up' and 'down' or 'upper' and 'lower'.

You can use several sensors with the same driver, each using its own (or the same) reference table, by simply making copies of the above text block (the '-id:' line and everything below it), with the necessary changes.

Don't use special characters like '+' and '-' in names that you choose. The underscore '_' can be used.

The generic_serial_input driver applies, obviously, to a serial port. Hence all that was described about serial ports in 'Common settings for drivers' (section 6.2) applies. Thus you can define a Power Port, a sample_interval, a baud rate, and so on. Make sure rxbuf is big enough to hold one complete output string from the sensor.

12.1.3 Create a customized reference_tables file

The generic_serial_input driver makes use of the file “serial_input_reference_tables.py”. You can create this file from a template file: ‘serial_input_reference_tables_*template*.py’. This file is located on the SD card, in the \script\modules folder.

Just copy ‘serial_input_reference_tables_*template*.py’ to the same folder, but rename it to ‘serial_input_reference_tables.py’ (remove the ‘_template’ at the end). Then customize ‘serial_input_reference_tables.py’ for your sensor.

Note that the ‘serial_input_reference_tables_*template*.py’ file is not used by the logger. It is there only for you, to help you create ‘serial_input_reference_tables.py’. If, later on, you copy a new release of the software over your current release, the ‘_template’ file will be overwritten, but the file you customized will not be overwritten. That’s the reason for having two files.

Notes:

- All generic drivers use a specific .py file containing reference tables.
- The files are located on the SD card in \script\modules.
- In a release, these files are called “xyz_*template*.py”. Where ‘xyz’ is the name of the driver.
- Always make a copy “xyz.py”, which you can customize.
- The logger application uses the customized “xyz.py” file, not the “xyz_*template*.py” file .
- This prevents overwriting the customized files when installing a new release.

You can edit the ‘serial_input_reference_tables.py’ file with a text editor like Notepad, but is it more convenient to use an editor that is ‘language sensitive’ (in other words, that ‘understands’ the Python language. You can, for example use Notepad++. It will use syntax coloring to help you prevent making typing errors.

The header (top) of the Python file contains a lot of information about the file and how you should use it.

The file contains several tables that you can use as a starting point for making your own tables. You can create as many new tables as you like. Each table should have a unique name (without special characters like ‘+’ and ‘-’ in the name). From the config file, you refer to the tables that you want to use.

Each table should describe to the logger application how the data string from the sensor is constructed and how it should be interpreted. In section 12.1.1 you determined the format of the sensor’s output. Now we will use this to create the table.

As an example, we use the same GMX sensor that was used in section 11.5, explaining the Gmx_501 driver. We now pretend this driver does not exist and therefore we use the generic driver.

In the manual we find this about the default output string:

- Format:
  Node, Relative Wind Direction, Relative Wind Speed, Corrected Wind Direction, Corrected Wind Speed, Pressure, Relative Humidity, Temperature, Dewpoint, Solar Radiation, GPS Location, Date and Time, Supply Voltage, Status, Checksum.
- Example:
  Q,310,000.04,033,000.59,1032.1,040,+020.6,+006.7,0001,+50.762988:-001.539893:-0.80,2015-06-09T09:24:34.9,+05.1,0004, 3D

Assume that, using a terminal program (and port redirection), we confirmed that we indeed receive the above string from the sensor. If the sensor would have been configured differently, we would have received another string! Using the terminal program in hexadecimal mode, we also discovered that each line actually starts with hexadecimal 02 and ends with a carriage return and a line feed. In Python, these are noted as '0x02' and either '\x0D\x0A' or '\r\n'.

We can now create this table (just an example):

```
my_gmx = {
    0:('\x02', ',', '\r\n'),
    2:("Wind Direction", "deg",  "DIR" ),
    3:("Wind Speed", "m/s",  "SPEED" ),
}
```

Explanation:

- The name 'my_gmx' is referred to from the config file.
- The table entry '0' defines <start>,<separator>,<stop>, in this case '\x02', ',', '\r\n'
- There is no table entry '1', so we ignore the first parameter from the sensor, which happens to be the 'Node' name, which is 'Q' in this example.
- Entries 2 and 3 define the parameters 2 and 3 from the sensor, which happen to be *Relative Wind Direction*, *Relative Wind Speed*, according to the sensor's manual. We log these parameters as 'Wind Direction' and 'Wind Speed'. We can choose any name we like. Make sure you enter the right units (here 'deg' and 'm/s' and give them a unique tag (here 'DEG' and 'SPEED'). The tags only need to be unique within this table, because the sensor id ('gmx_a' in this example) will be appended (e.g. "DEG_gmx_a").
- You can add more entries if you want to sample more parameters.

Once you finished this file and saved it on the SD card (\script\modules folder), you can start the logger application. Make sure you have 'sensor_data_print: True' to begin with. Carefully check if the values reported in the REPL window correspond to the correct parameters. You will see lines like this:

```
Sensor data: ('Wind Direction', 'm/s', 'SPEED_gmx_a') 008.02
```

Note that, if you miscount the position of a parameter in the sensor's output string, everything will seem to work fine, but you will get the value of the wrong parameter!

Final note:
When editing a Python file, you are actually sitting on a programmers chair. Unless you are a Python programmer, you cannot know all the implications. So carefully stay close to the examples in the _template files. And take note of the fact that Python is sensitive to indents (somewhat similar to the YAML language used in the config files). So make sure you use the right number of leading spaces.

12.2 generic_nmea:

NMEA is often used for meteorological sensors and in the marine world in general. The generic_nmea driver allows you to define as many NMEA sentences as you like. You can apply the driver to any of the 4 serial ports, and to the dedicated NMEA input. The ports will be used unidirectional (input only).

I assume that you have read section 12.1 to get the general idea of generic drivers. Here I will only mention the specifics of the steps involved:

1. Configure and connect the sensor;
2. Add the driver to the config file;
3. Create a customized reference_tables file.

The steps are described in the following sub-sections.

12.2.1 Configure and connect the sensor

Make sure the sensor outputs what you need, and preferable not too much more. That will just waste (processing) power. So try to switch off unused sentences.

Start, separator and stop symbols are defined by the NMEA standard, so you don't need to bother with them. The baud rate is standard 4800 Bd but you can use 9600 Bd or whatever you like, as long as you define it in the config file. NMEA standard uses RS422, but you can also use RS232 if you specify it in the you-know-where (yes, in the config file).

Note that the dedicated NMEA port is actually just an isolated RS422 port. It can be used for RS232 input signals as well (refer to chapter 1).

After connecting to the logger, you can use port redirection (section 14.2.3) to check the incoming data.

12.2.2 Add the driver to the config file

To use the driver, add this to the config file:

```
generic_nmea:                          # This means "nmea_reference_tables.py" will be used
- id: gnss_up
  port: nmea                           # Could also be serial1, serial2, …
  reference_table: VTG,GGA,RMC         # This means these tables/sentences will be used
  < text left out here>
```

As with the generic_serial_input_driver, you can use the same driver for multiple sensors (each with a unique id) and for each you can define which tables to use. Also, for each sensor you can define the baud rate, the sample_interval, the Power Port and so on.

12.2.3 Create a customized reference_tables file

Refer to 12.1.3 for a general description on reference tables.

Copy 'nmea_reference_tables_*template*.py' to 'nmea_reference_tables.py' and customize it. The file contains several example-tables, some of which are standardized tables. Just add the tables you need for your sensor and refer to them from the config file.

Note that, in each table, you only need to mention the parameters that you want to take out of the corresponding sentence. The examples show how you can comment-out (with '#') parameters that you don't need.

12.3 generic_sdi12:

SDI-12 is a widely used serial bus protocol. It allows one master device (in this case the OMC-048) to communicate bidirectionally with up to 64 sensors, each identified by a unique slave address.

Communication is relatively slow, with 1200 Bd, but for sensors that produce only a relatively small amount of data, this is still enough. To our experience, SDI-12 is one of the easiest to use interfaces. And it is widely supported. The logger has only one SDI-12 port, but since this is a bus, you can connect multiple sensors to it.

The reader is assumed to be somewhat familiar with SDI-12. Also, I assume that you have read section 12.1 to get the general idea of generic drivers. Here I will only mention the specifics of the steps involved:

1. Configure and connect the sensors;
2. Add the driver to the config file;

3. Create a customized reference_tables file.

The steps are described in the following sub-sections.

12.3.1 Configure and connect the sensors

First configure the sensors to your liking. In response to a 'Send Data Command', they should send the desired parameters in the desired units. Make sure each sensor has a unique slave address.

Make sure you have the **right parameters** in the **right order** and in the **right units** before you continue!

I recommend labelling each sensor (or each sensor cable) with the slave address and with the id that you are going to give the sensor in the config file and the reference_table (see below). Especially if the sensors are otherwise identical, it is easy to get them mixed up. For example, if you have two PT (Pressure & Temperature) sensors, you could give them slave address 1 and 2 and give them the id PT1 and PT2. If you also have two CT (Conductivity & Temperature) sensors, you could give them slave address 11 and 12 and id CT11 and CT12. Or CT1 and CT2.

Next you can connect the sensors to the SDI-12 port of the logger.

If you intend to switch power on/off to all of them simultaneously, you can connect all sensors to the same Power Port (keeping the current limit in mind). If you don't want to switch the power, you can connect them directly to a power supply.

Important!
You can connect multiple sensors to the same SDI-12 port, but only if all sensors support the Concurrent commands C! or CC!. These must also be specified in the reference tables. Obviously, the sensors should all have a unique slave address.

12.3.2 Add the driver to the config file

To explain this, two examples are given.

Example 1: Multiple sensors scheduled together

Add below text to the config file:

```
generic_sdi12:
- id: sdi12
  port: sdi12
  baudrate: 1200
  bits: 7
  parity: 0
  reference_table: pt12,pt2x
  sample_interval: "0,10,20,30,40,50 * * * *"
```

You can choose a single sensor_id that will be used for all referenced sensors. You can list multiple reference tables. You need one table per sensor and give it a unique name. In the logged data, the sensor will be identified by the table name and the sensor id. For example, with the above configuration, the tag 'Temp' will be named 'Temp_pt12_sdi12' respectively 'Temp_pt2x_sdi12'. If you have multiple identical sensors, for example two PT12's, they should all have their own table with a unique name. For example, the tables could be called PT_1 and PT_2 ('`reference_table: PT_1,PT_2`'). Table PT_1 would have the slave address of the sensor labelled PT_1 and the tag 'Temp' of this sensor would become 'Temp_PT_1_sdi12'

Make sure the tables you reference do exist in the reference_tables file (next sub-section). The other lines ('port' till 'parity'), shown in **bold** MUST be included as indicated.

If you use only a single sensor and hence refer to only a single table, you can use either C! or M! commands. Whatever your sensor supports. If you use multiple sensors, you can only use C! (or CC!).

Note that, in this example, all sensor will get their C! (or CC!) command on (practically) the same scheduled moment, hence all sensors are started (practically) simultaneously. (I write 'practically', because SDI-12 is a serial bus, so things always happen one after another. But the time difference is much less than a second, so there is 'practically' no time difference.)

Example 2: Multiple sensors scheduled independently

Add below text to the config file:

```
generic_sdi12:
- id: sdi1
  port: sdi12
  baudrate: 1200
  bits: 7
  parity: 0
  reference_table: pt1,pt2
  sample_interval: "0 0,10,20,30,40,50 * * *"
- id: sdi2
  port: sdi12
  baudrate: 1200
  bits: 7
  parity: 0
  reference_table: ct1,ct2
  sample_interval: "0 5,15,25,35,45,55 * * * *"
```

Now we have two sensor id's and 4 unique tables. So this example is for 4 sensors. The 4 tables should obviously exist in the reference_tables file (next sub-section).The 'Temp' tag of these sensors will be 'Temp_pt1_sdi1' or 'Temp_ct2_sdi2' and so on.

All sensors must support the C! (or CC!) command and have a unique slave address.

The sensors referenced under id sdi1 and those under sdi2 can be scheduled independently. However, because they are on the same physical bus, they should not interfere. The process for sdi1 should be finished before sdi2 starts and the other way around. It is the user's responsibility to make sure that all processes (in this case there are two) do not overlap. In this example, it means that sensors should be ready in less than 5 minutes.

In the generic_sdi12: driver it is not recommended to specify a response_timeout value, because the delay of a sensor is handled by the SDI-12 protocol.

12.3.3 Create a customized reference_tables file

Refer to 12.1.3 for a general description on reference tables.

Copy ‘sdi12_reference_tables_*template*.py’ to ‘sdi12_reference_tables.py’ and customize it.

This file already contains a large number of example tables. They *could be* ready for use, but -depending on your sensor configuration- you *may* need to modify the tables! In any case, you *must* check if what your sensor actually does, corresponds to what the tables specifies. For example:

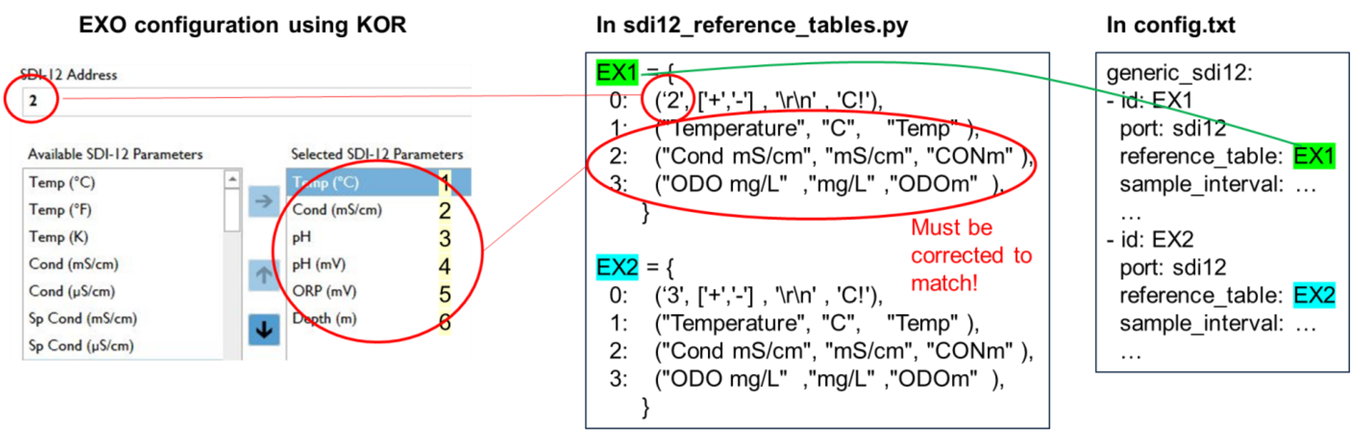
- The sensor (slave) address should match.
- The values that the sensor returns on a data command should match the table. Also make sure the units match! For example, a pressure sensor may be configured to output pressure in hPa or in PSI or more. This configuration should match the table.
- Use the concurrent command C! or CC! if your sensors support this, or use the M! or MC! commands.
- You don’t need to bother with the SEPARATOR and STOP characters, as they are standardized.

Note that you need a table (with a unique name) for each connected sensor. If, for example, you connect two EXO3 sondes, you could call the tables EX1 and EX2 and label your sondes accordingly.

Generally, you should configure your sensors to output only the parameters you need. But, if for some reason a sensor has more parameters than you need, you can leave the unwanted parameters out of the table. But you still should increment the line number, as is shown in the example below. In this example, the parameter with number 2 is commented out, so will not be read by the driver.

```
EX1 =   {
            0:   ('2', ['+','-'] , '\r\n' , 'M!'),
            1:   ("Temp",      "PSIG",   "Pres" ),
          # 2:   ("Cond",    "C",     "Temp" ),
            3:   ("PH",        "V",      "Volt" ),
            4:   ("PHmv",      "PSIG1",   "Pres1" ),
            5:   ("ORPmv",    "C1",     "Temp1" ),
            6:   ("Depth",       "V1",     "Volt1" ),
```

A note about using EXO sondes with SDI-12:
You can connect the EXO3 sonde directly to the OMC-048 by SDI-12, because the EXO3 has an SDI-12 interface. The EXO1 and EXO2 do not have such an interface. They require the use of an intermediate DCP adapter. You can connect by SDI-12 to this adapter, but probably the RS232 interface with the dedicated *EXO:* driver is more suitable in those cases. When using SDI-12, use the KOR software to select the desired SDI-12 output parameters. Make sure they are in the same order as in the reference table. See Figure 18, where the mismatch between the selected parameters in KOR, and the reference table still needs to be corrected.



```
EX1 = {
  0:   ('2', ['+','-'] , '\r\n' , 'C!'),
  1:   ("Temperature",  "C",    "Temp" ),
  2:   ("Cond mS/cm", "mS/cm", "CONm" ),
  3:   ("ODO mg/L"  ,"mg/L" ,"ODOm"  ),
     }

EX2 = {
  0:   ('3', ['+','-'] , '\r\n' , 'C!'),
  1:   ("Temperature",  "C",    "Temp" ),
  2:   ("Cond mS/cm", "mS/cm", "CONm" ),
  3:   ("ODO mg/L"  ,"mg/L" ,"ODOm"  ),
     }
```

```
generic_sdi12:
- id: EX1
  port: sdi12
  reference_table: EX1
  sample_interval: …
  …
- id: EX2
  port: sdi12
  reference_table: EX2
  sample_interval: …
  …
```

Figure 18. Relation between the EXO configuration, the reference_tables file, and the config file.

12.4 modbus:

ModBus is a widely used serial bus protocol. It allows one master device (in this case the OMC-048) to communicate bidirectionally with a large number of sensors (slaves), each identified by a unique slave address. The OMC-048 supports Modbus RTU over RS485. All devices on a single RS485 bus should be set to the same baud rate and parity, and must have a unique slave address.

The reader is assumed to be somewhat familiar with Modbus. Also, I assume that you have read section 12.1 to get the general idea of generic drivers. Here I will only mention the specifics involved.

The modbus driver is a generic driver for ModBus RTU. It supports the basic read and write actions from/to registers. Because many sensors require a specific sequence of read and write actions for normal operation, specific drivers may be needed for these sensors. These specific drivers then use the generic ModBus driver. For example:

- The 'modbus_seametrics' driver uses the generic 'modbus' driver to read the sensor data twice, to get rid of old data first. See section 11.9.
- The 'Modbus-opus' drivers uses the generic 'modbus' driver to issue commands, read status, wait the appropriate times, and finally to read data. So a complex sequence of actions, specific for the OPUS sensor, is executed. See section 11.8

If your sensors requires just the reading of registers, without issuing commands or anything else, all you need is this generic driver. In all other cases, a specific driver needs to be written (in Python code) 'on top' of the generic modbus driver.

To use the generic modbus driver or any of the specific modbus drivers, follow these steps:

1. Configure and connect the sensors;
2. Add the driver to the config file;
3. Create a customized reference_tables file.

The steps are described in the following sub-sections.

12.4.1 Configure and connect the sensors

First configure the sensors to your liking. Make sure each sensor has a unique slave address. Also make sure they are all set to the same baud rate.

I recommend labelling each sensor (or each sensor cable) with the slave address and with the id that you are going to give the sensor in the config file and the reference_table (see next sub-sections). Especially if the sensors are otherwise identical, it is easy to get them mixed up. For example, if you have two PT (Pressure & Temperature) sensors, you could give them slave address 1 and 2 and give them the id PT1 and PT2. If you also have two CT (Conductivity & Temperature) sensors, you could give them slave address 11 and 12 and id CT11 and CT12. Or CT1 and CT2.

Next you can connect the sensors to any of the 4 serial ports of the OMC-048. Note that, since modbus uses RS485, you need to connect to the A and B pins of the port. It is common to label the two RS485 signals by 'A' respectively 'B', or by '-' (=A) and '+' (=B). However, different manufacturers seem to have different opinions about which way around to label the signals, so in some cases you may have to swap the signals. You cannot use the NMEA port, because it is input-only.

Next to the two signal wires A and B, you also need to make sure the grounds are connected. Sometimes labelled 'C', but often 'GND'.

If you intend to switch power on/off to all sensors simultaneously, you can connect all sensors to the same Power Port (keeping the current limit in mind). If you don't want to switch the power, you can connect them directly to a power supply.

12.4.2 Add the driver to the config file

Add below text to the config file (example):

```
modbus:
- id: modbus_1
  protocol: RTU
  register_print: True
  port: serial1
  mode: RS485
  rxbuf: 1024
  baudrate: 9600
  reference_table: mt1,mt2
- id: modbus_2
  # If you want to independently schedule other sensors
  reference_table: mt1,mt3
```

Also add the usual lines for the sample interval, the power port and so on. Like with any other (serial) sensor.

Note the following settings:

protocol:	You only have one option here: RTU
register_print:	This will print the raw register values in 'unsigned short' notation. This is convenient for helping you set up the reference table correctly. Once you have done so, you can remove 'register_print' or set it to 'False'.
rxbuf:	If you read a large number of registers, it may be wise to increase rxbuf from its default (128). See section 6.2.

As you can see, you can list multiple tables after 'reference_table'. For each sensor you need a table. The sensors will share all settings (like sample_interval and other timing settings). This is similar to the SDI-12 driver, so please refer to 12.3.2 for details.

If you want the sensors to be scheduled independently, you can make an '-id:' section for each sensor (or group of sensors) that you want to schedule independently. Again, refer to 12.3.2 for details. The main difference with SDI-12, is that you can now use multiple ports, so you don't have to worry about conflicts.

Note that the default port setting uses no parity (parity: None).

12.4.3 Create a customized reference_tables file

Refer to 12.1.3 for a general description on reference tables.

Copy 'modbus_reference_tables_*template*.py' to 'modbus_reference_tables.py' and customize it.

This file already contains a large number of example tables. They *could be* ready for use, but -depending on your sensor configuration- you *may* need to modify the tables! In any case, you must check that the table matches your sensor. In particular, make sure that the slave address is correct.

In the config file, you can use 'register_print' to help you create the right table. Use 'sensor_data_print: True' to verify that you are reading the correct data from the registers. Once the table is correct, you can remover these settings from the config file, or set them to False.

12.5 Iridium:

This 'generic' driver is a bit different from the other generic drivers, because on one hand it is not really 'generic' but 'specific' for one type of Iridium modem, while on the other hand it uses a reference_table for customization, just as the 'generic' drivers do. This explains is why the driver is called 'Iridium:' instead of 'generic_iridium'.

I assume that you have read section 12.1 to get the general idea of generic drivers.

12.5.1 Iridium_modem:

Please refer to chapter 9 for an explanation on the use of Iridium. Don't forget to create a customize 'iridium_reference_tables.py' file, see section 12.5).

To declare the use of the Iridium modem, include this part in the config file:

```
Iridium_modem:
  id: iridium_modem
  port: serial1
  supply_port: 1
```

Obviously, you should make both port numbers correspond to your physical connections. The baud rate and so on are fixed by the driver.

12.5.2 Iridium:

Please refer to chapter 9 for an explanation on the use of Iridium.

```
Iridium:
- id: data_iridium
  system_id: Z
  transmit_interval: "0 0 * * *"
  add_timeout_field: True
```

Obviously, you need to choose your system_id and transmit_interval. The system_id is included in each transmission, so it is wise to use only a single character to identify your logger.

The ‘`add_timeout_field`’ is default (if you leave the line out) set to False. This means that, if a sensor fails to respond within its response_timeout interval, its parameters will not be logged and hence will not appear in the data file at all. This saves data (cost).

If ‘`add_timeout_field: True`’, a sensor timeout will result in an empty field in the data file. This can help the interpretation of the file, because you can now count the fields. Refer to sub-section 18.2.2 for more details on the file format.

12.5.3 Iridium_file:

Please refer to chapter 9 for an explanation on the use of Iridium.

```
Iridium_file:
- id: iridium
  create_interval: "55 59 * * *"
```

12.5.4 Create a customized reference_tables file

In chapter 9 it is explained how to connect the Iridium modem to the logger and how to create a sensible timing schedule. It was also explained that you need to include a three text blocks in the config file (Iridium_modem:, Iridium:, Iridium_file:).

In contrast to the other generic drivers, the *Iridium:* block does not include an explicit reference to a reference table. Instead, the reference is implicit and fixed to the table called ‘iridium_standard’ in the ‘iridium_reference_tables.py’ file.

Copy the ‘iridium_reference_tables_*template*.py’ to ‘iridium_reference_tables.py’ and customize it to your needs. The header (top part) of the file gives you some information on its use.

To customize the table, you can follow the same procedure as described for the ‘generic_serial_output’ driver: First ignore the fact that you want to use Iridium. Create a config file for everything else that you want to use in your final application. Thus leave out the blocks related to Iridium. Connect and configure all the sensors and so on. Make sure you have ‘sensor_data_print: True’ and start the application while watching the REPL. You will see all sampled data printed in ‘(name, unit, tag)’ lines like this:

```
Sensor data: ('Int. Temperature', 'C', 'ITEMP') 22.58428
Sensor data: ('Cp', '%', 'C1') -25.0
Sensor data: ('pH', 'pH', 'PH_EX1') 17.41
```

Decide which parameters you want to include in the Iridium transmissions. In contrast to the other reference tables, you now only need the <u>tags</u> of the parameters. In the above example, the tags are ‘ITEMP’, ‘C1’ and ‘PH-EX1’. The tags of the required parameters should be copied to the left-most column of the reference table.

This is the table from the iridium_reference_tables_*template*.py file, adapted a little for readability:

```
iridium_standard =  {
#Sensor TAG  #Iridium TAG  #Nr of decimals  #Char clip (includes +/-/,)
'IVBAT' :      ('V',               1,                 4,       ),
'ITEMP' :      ('T',               2,                 5,       ),
'IHUM'  :      ('H',               3,                 8,       ),
'LAT'   :      ('L',               None,              None,    ),
'LON'   :      ('l',               None,              None,    ),
                     }
```

In the left-most column you can fill-in the tags of the parameters you want. The '#Iridium tag' defines the tag that the parameter will get in the Iridium file. To save on data transmission cost, it is wise to use a single character. The software that is used to receive and interpret the data files (like OMC-Data-Online or Blue2Cast) uses the 'Iridium TAG' tag to identify the parameter. (The 'sensor TAG' is not transmitted.)

The '#Nr of decimals' column defines how many decimals (behind the decimal '.') should be transmitted. For example, '5.1' contains 1 decimal value ('1').

The '#Char clip' defines the maximum number of characters used for the whole parameter value. This allows you to control the maximum numbers of character in the file. The .py file shows you some examples. Due to restrictions on the Iridium SBD messages, the total file size should not exceed 338 bytes.

Once you completed the reference table, you can complete the config file. Next, you can do a test without actually transmitting data, by either using a modem that has not yet been registered by your Iridium provider, or with a modem that is indoors and cannot 'see' a satellite. Then your transmissions will fail (avoiding data-cost), but you will still get the Iridium files on your SD card (see chapter 9). You can use these to check that:

- Your files contain all the parameter you need with the correct tag and precision;
- The total file size is within the 338 limit.

12.6 generic_serial_output:

This is, as you could have guessed by the name, an *output* driver. It produces serial output on any of the four serial ports. Some application examples are:

- Sending data from the OMC-048 to a PC (COM port) or PLC over RS232/422.
- Sending data from the OMC-048 to a radio or other modem.
- Sending data from the OMC-048 to an alphanumerical display.

This output driver can work in parallel -and independent of- the cellular and Iridium modems.

The generic output will output the sampled values immediately after they are sampled. Thus there is no intermediate storage and hence you cannot define a transmit_interval (refer to 12.7 for an alternative that does use a configurable interval)

You can specify precisely which of all sampled parameters you want to output. So you can make a selection.

To use the driver:

1. Configure the logger for obtaining the required samples;
2. Add the driver to the config file;
3. Create a customized reference_tables file.

12.6.1 Configure the logger for obtaining the required samples

First ignore the fact that you want to use the generic_serial_output. Create a config file for everything else that you want to use in your final application. Connect and configure all the sensors and so on. Make sure you have 'sensor_data_print: True' and start the application while watching the REPL. You will see all sampled data printed in '(name, unit, tag)' lines like this:

```
Sensor data: ('Int. Temperature', 'C', 'ITEMP') 22.58428
Sensor data: ('Cp', '%', 'C1') -25.0
Sensor data: ('pH', 'pH', 'PH_EX1') 17.41
```

You can now copy-paste the colored parts (between the round brackets (…) including the brackets) of the required parameters to a temporary text file. Make sure you identified and copied all parameters that you want to be output by the generic_serial_output driver..

For advanced users:
For some sensors you can also find the '(name, unit, tag)' parts in the driver file. For example, in EXO.py (located in \script\modules) you can find the line "`18: ("pH" ,"pH" ,"PH")`". So, if you know that you want to include this parameter from the EXO sonde, then you know which string to use, even before connecting the sonde. This 'trick' allows advanced users to customize the serial_output_reference _tables.py' file without even having an EXO at hand. But take care that the tag "PH" (as shown in ***ex.py****) is extended with _<id>, where <id> is the id of the EXO (defined in the exo: driver). The reason is explained in section 11.1.3. (To refresh your memory: It is to make the tags unique if you have multiple EXO sondes.) This also explains why you have to be an advanced user to use this trick! In the reference_table, you need to use 'PH_EX1', if you want to output the parameter with tag 'PH' from the EXO with id 'EX1'.*

You need the list of the desired '(name, unit, tag)' lines when customizing the reference_tables file.

12.6.2 Add the driver to the config file

To use the generic_serial_output driver, add this to the config file:

```
generic_serial_output:
 - id: gs_out
   port: serial1
   reference_table: sensorA,OMC2900
   baudrate: 9600
   mode: RS232
   < text left out here>
```

As mentioned before, this is not a sensor driver, so things like sample_interval do not apply. Also note that this is just an example. You can choose your own table names.

Each time data from a particular sensor is received, the driver will 'look-up' the received '(name, unit, tag)' in all of the listed reference tables. If, for example, a match is found in table 'sensorA', the driver will output a string with the received sensor data in the format defined in the 'sensorA' table. Thus you can write different tables for the different sensors you have (even if you don't use them in this particular application), and refer only to the tables of the sensors that are actually connected.

Finally, note that you can generate multiple outputs on different ports with the same or different tables using this driver. Just copy & edit the '-id:' line with everything below it. And assign a different port to each of them.

12.6.3 Create a customized reference_tables file

Copy 'serial_output_reference_tables_*template*.py' to 'serial_output_reference_tables.py' and customize it. The file contains several example-tables. Just add or modify the tables for your output. And make sure the names of the tables correspond to the names used in the config file.

The Python file also contains a header with some explanation on what to do. You can also add a checksum to the output string, or you can completely reformat the string. You need to have programming experience to understand all options. You can ask Observator for more examples.

12.7 Data_serial_out:

Like generic_serial_output (12.6), this is also an output driver and not a sensor driver. It operates very similar to the cellular modem, except that the data file is not transmitted over the cellular network, but serialized over one of the serial ports of the OMC-048.

The data_serial_out: operates independently from the cellular output and uses its own (temporary) file. Hence, they can be used simultaneously. You can use only one data_serial_out driver.

Please refer to chapter 8 to refresh your memory on how scheduling for the data_file and the cellular modem works. Refer to chapter 18 for a description of the data_file for cellular transmission, as the format is the same as for data_serial_out. The principle of closing and opening a new data file at each specified moment is also the same.

The Data_serial_out driver can be used either with or without a reference table. If used:
- Without table: the output string is the same as the 'OMC-045' file format.
- With table: the output string is formatted as indicated in the table. Similar to the generic_serial_output driver.

Both options are described in different paragraphs below.

12.7.1 Without a table

This is explained by an example.

Suppose this is the only sensor driver in the config file:

```
Onboard:
- id: onboard
  sample_interval: "* * * * *"
```

Also assume that system_id: RBE (in the block Omc048:)

In config.txt, add these lines

```
Data_serial_out:
- id: data_serial
  create_interval: "0,10,20,30,40,50 * * * *"
  port: serial1
  baudrate: 115200
  mode: RS232
  txbuf: 1024
```

This will create and output a file every 10 seconds. The onboard sensors are sampled every second. The output could look like this on an ASCII terminal:

```
OMC-045_rbe_048000125
T;25;26;27;
Timestamp;Int. Temperature;Int. Humidity;Int. Coin cell voltage;
yy/mm/dd hh:mm:ss;C;rh;V;
Time;ITEMP;IHUM;IVBAT;
D;00/01/01 00:09:12;25;27.11028;0;26;40.66138;0;27;0.3771429;0;
D;00/01/01 00:09:13;25;27.11028;0;26;40.67664;0;27;0.3771429;0;
D;00/01/01 00:09:14;25;27.11028;0;26;40.66901;0;27;0.3674725;0;
D;00/01/01 00:09:15;25;27.11028;0;26;40.67664;0;27;0.3835897;0;
D;00/01/01 00:09:16;25;27.11028;0;26;40.66138;0;27;0.3739194;0;
```

```
D;00/01/01 00:09:17;25;27.121;0;26;40.65375;0;27;0.3803663;0;
D;00/01/01 00:09:18;25;27.11028;0;26;40.66138;0;27;0.3674725;0;
D;00/01/01 00:09:19;25;27.11028;0;26;40.64612;0;27;0.3900366;0;
D37B
```

In the REPL it would show like this (irrelevant text is left out)

```
Started logging in 'data/OMC-045_rbe_tmp_000101_000911.txt'
Sensor [onboard] data successfully obtained
  Sensor data: ('Int. Temperature', 'C', 'ITEMP') 27.11028
  Sensor data: ('Int. Humidity', 'rh', 'IHUM') 40.66138
  Sensor data: ('Int. Coin cell voltage', 'V', 'IVBAT') 0.3771429
```
--- repeated lines left out
```
Sensor [onboard] data successfully obtained
  Sensor data: ('Int. Temperature', 'C', 'ITEMP') 27.11028
  Sensor data: ('Int. Humidity', 'rh', 'IHUM') 40.64612
  Sensor data: ('Int. Coin cell voltage', 'V', 'IVBAT') 0.3900366
Closed file data/OMC-045_rbe_tmp_000101_000911.txt
Started logging in 'data/OMC-045_rbe_tmp_000101_000920.txt'
Starting serial transmission of file: 'data/OMC-045_rbe_tmp_000101_000911.txt'
Transmission over serial completed, removed file:'data/OMC-
045_rbe_tmp_000101_000911.txt'
```
… and so on

The differences with cellular transmission:

- On data_serial_out, the output starts with the <02> followed by the filename. Then follows the file contents. Finally, a checksum (CRC16) is appended followed by <03>. Every text line, including the file name and the checksum with <03>, is terminated with <0D><0A>. Thus:
  ```
  <02><filename><0D><0A
  <first line><0D><0A>
   …
  <last line><0D><0A>
  <CRC><03><0D><0A>
  ```
- The file name is constructed as below example shows.
- After transmission (which always succeeds, because there is no feedback), the file is always deleted. (Not moved to a 'transmitted' folder). You can use data_file: in parallel with serial_data_out: to keep data organized in files on SD.

All sensor data is stored in the temporary file and is serialized.

Make sure that baud rate is set sufficiently high to output all data in time. Also make sure that the output buffer size is big enough (e.g. txbuf: 1024).

12.7.2 With a table

This is explained by an example.

Suppose this is the only sensor driver in the config file:

```
Onboard:
- id: onboard
  sample_interval: "0 * * * *"
```

Now add the driver to the config file. Note that the last line specifies a table.

```
Data_serial_out:
- id: data_serial
  create_interval: "0 * * * *"
  port: serial1
  baudrate: 9600
  mode: RS232
  txbuf: 1024
  reference_table: MyTable
```

Now, because the driver specifies a table, a customized ‘serial_output_reference_tables.py’ file is needed. For example, add below table to the file:

```
MyTable = {
    0: ('', ' ', '\r\n'),
    1: (('Int. Coin cell voltage', 'V', 'IVBAT'), "{:.5f}"),
    2: (('Int. Temperature', 'C', 'ITEMP'),       "{:.5f}"),
    3: (('Int. Humidity', 'rh', 'IHUM'),          "{:.5f}")
          }
```

This will produce below output. One line per minute, on second 0.

```
2.95912 24.15350 49.32327
2.94300 24.15350 49.34921
2.94945 24.12413 49.34463
```

Note that this is very similar to generic_serial_output. The main difference being, that generic_serial_output generates a line each time a sensor delivers data that matches a table. In contrast, Data_serial_out first collects data from all sensors in a temporary file, and serializes this data on the time points specified by ‘create_interval’. You can format the strings to include a time stamp, a CRC or a tag (name) for each value.

Also note that, if the temporary file contains multiple samples of the same parameter, only the last one is output. For example, if in the above example the onboard sensors would be sampled every second, and the ‘create_interval’ would still be 10 seconds, the data file would contain 10 samples of each parameter. Only the last (most recent) would be serialized (output).

Finally, make sure that baud rate is set sufficiently high to output all data in time. Also make sure that the output buffer size is big enough (e.g. txbuf: 1024).

13 Controlling wipers, flow-cell pumps and other devices

The OMC-048 is not intended to be a *controller*, like a PLC. However, some simple control is possible and is supported. In more complex cases, you will need an external controller.

Some control functions supported by the OMC-048:

- **Activating a wiper** for cleaning a sensor. In some cases, like the EXO sondes (section 11.1) the wiper is integrated in the sensor (sonde) and the specific sensor driver handles the wiper control. In other cases, like the Eureka sondes, no control is needed, other than switching sensor power on and off. However, some sensors may use external wipers that require some control. Examples are the wipers from Zebra Tech, Aqualabo, TriOS or s::can.
- **Activating a pressure valve** for cleaning a sensor. Some sensors have a cleaning mechanism using pressurized air. This has to be activated periodically.
- **Activating a UV LED** for anti-fouling. Some sensors use an UV-LED to prevent bio-fouling. If this LED is controlled from the sensor (like for the MoSens CT sensor) no control is needed. But in some cases it may be needed to switch the LED on and off periodically.
- **Activating a pump** for a flow cell. If a sensor is mounted in a flow cell, it may be needed to switch a pump on before a measurement, and off after the measurement. This saves power compared to continuous pumping.

In section 13.1 I will describe the general principles for controlling an external device, regardless of what that device is. After that, I will give some specific examples.

Advise:
Once you made the connections (wires) between the external device and the OMC-048, it is very convenient to use some commands in the REPL to switch power and relays on/off for testing. In the REPL, first use ‘import omc048’ and then commands like, for example:

- p = omc048.power_supply(1); p.on(); p.off(); and p.toggle()
- r = omc048.relay(1); r.on(); r.off(); and r.toggle()

See section 14.2 for details.

13.1 Controlling external devices

To control an external device there are basically two ways:

1. switch the power on/off;
2. give a trigger pulse or a ‘on’-signal.

In both cases, the device can either return some status signal (like ‘failure’ or ‘ready’) or not.

Above two case only differ in the way the device is wired to the logger. But the configuration (programming) of the logger is the same in both cases. Therefore I will first describe the two wiring options. Then I will describe the configuration which can be with or without status return signal.

If you are able to program Python, you can make more complex control by writing your own driver. That is beyond the scope of this manual.

13.1.1 Switching power on/off

Figure 19 shows two ways to switch power to a device on/off. This is exactly the same as how you switch power to a sensor. If you need to switch high voltages or currents, you can use the shown schematics to drive a (large) external relay.

The left schematic has the following properties:

- The Power Ports are used. This means you have 12 V (and 5 V for loggers with SN>xx200). These voltages are made from the power supply of the logger. If the supply voltage is above 12 V (for example: 24 V), these outputs stay at their nominal level. Only when the supply voltage drops below 12V, the Port voltage also drops below 12 V. Thus you can use 12 V (or 5V) devices in a system that is powered at, say 24 V.
- The output currents are limited to what the Power Ports can handle (refer to the hardware manual or the data sheet).
- If power to the logger is switched off, the power at the Power Port will obviously be gone. If the logger is switched on again, the Port will stay off till the logger application switches it on again.
- If the application program crashes, or is interrupted by Control-C in the REPL, the Power Port stays in its current state (on or off).

The right schematic has the following properties:

- The internal relays are used. This means you can use an external power supply, independent of the power supply of the data logger. Or you can use the same power supply as the logger. The relay contacts are electrically isolated from the rest of the logger.
- Because you are not going through the internal DC/DC converters, you can go for higher currents and you don't have power conversion losses.
- The relays are bi-stable. This means they retain their last position if the logger application is stopped for whatever reason (crash, control-C, power-down). This means that, if you have a separate power supply for the external device, it may remain powered even if power to the logger is lost. However, if the *input_power_monitor* (section 10.5) is used, the relay will be switched off when the voltage drops below the threshold.

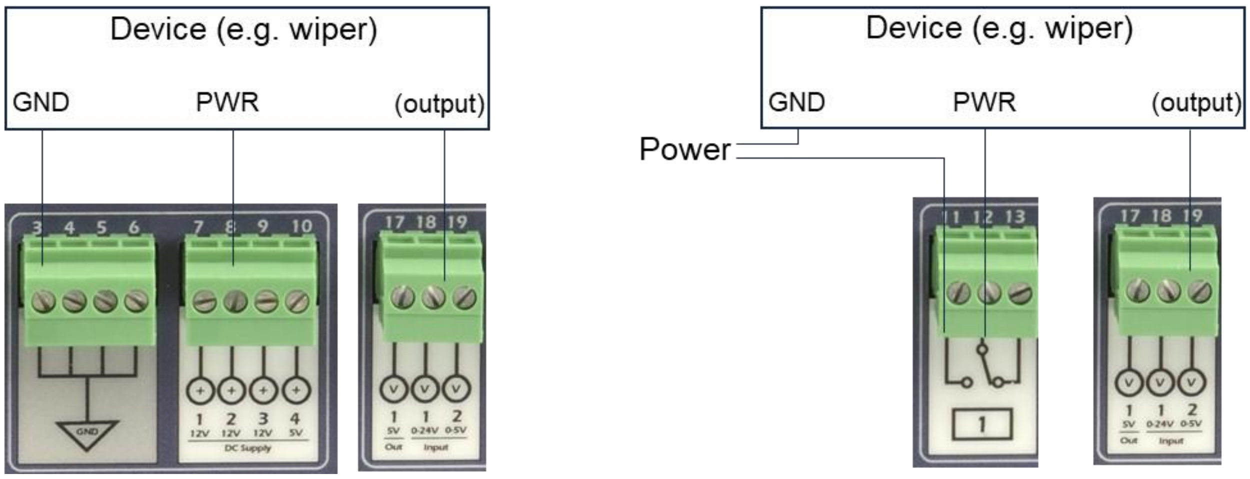


Figure 19. Switching power on/off using a power port (left) or using an internal relay (right). Optionally with a return (output) signal

For the configuration, see 13.1.3 or 13.1.4

13.1.2 Trigger pulse

If a device needs to receive a trigger pulse while it is powered, you can use the same schematics as shown in Figure 19. For ‘PWR’ you can read ‘Trigger’. This way you can issue trigger pulses of 12V or 5 V from the power ports, or any available voltage through the relays.

For the configuration, see 13.1.3 or 13.1.4

13.1.3 Configuration without return signal

As an example, let’s assume that the external device is connected to Power Port 1. Whether this is the power to the device, or the trigger input, is not relevant for the configuration. Let’s assume that the device has no return signal and needs power (or trigger) for 5 seconds every whole minute.. In that case, the logger only needs to generate an output pulse. So we can simply use the pulse_driver.

In the config file, add the following text for this example:

```
pulse_driver:
- id: wiper
  sample_interval: "0 * * * *"
  supply_pulse_port: 1
  pulse_duration: 5
  normally_off: True
```

13.1.4 Configuration with a return signal

As an example, let’s assume that the external device is connected to Power Port 1. Whether this is the power to the device, or the trigger input, is not relevant for the configuration. Let’s assume that the device also has a status output (voltage) that we connect to the analog voltage port 1. In the config file, add the following text for this example:

```
Analog_voltage:
- id: some_device
  port: 1
  sample_interval: "50 * * * *"
  min_in: 0
  max_in: 24000
  min_out: 0
  max_out: 24
  supply_port: 1
  supply_port_always_on: False
  startup_time: 3
  log_name: Status
  log_unit: V
  log_tag: S
```

This configuration will power the device for 3 seconds, starting at second 50 of every minute. At the end of the 3 seconds, the analog voltage (Status) is read and power is switched off.

You can add a *cooldown_time*, if you want the power to stay on a while after sampling the voltage.

If you want to use the relay instead of the power port, simply replace

```
  supply_port: 1
  supply_port_always_on: False
```

By:

```
  relay_port: 1
  relay_port_always_on: False
```

13.2 Controlling wipers (examples)

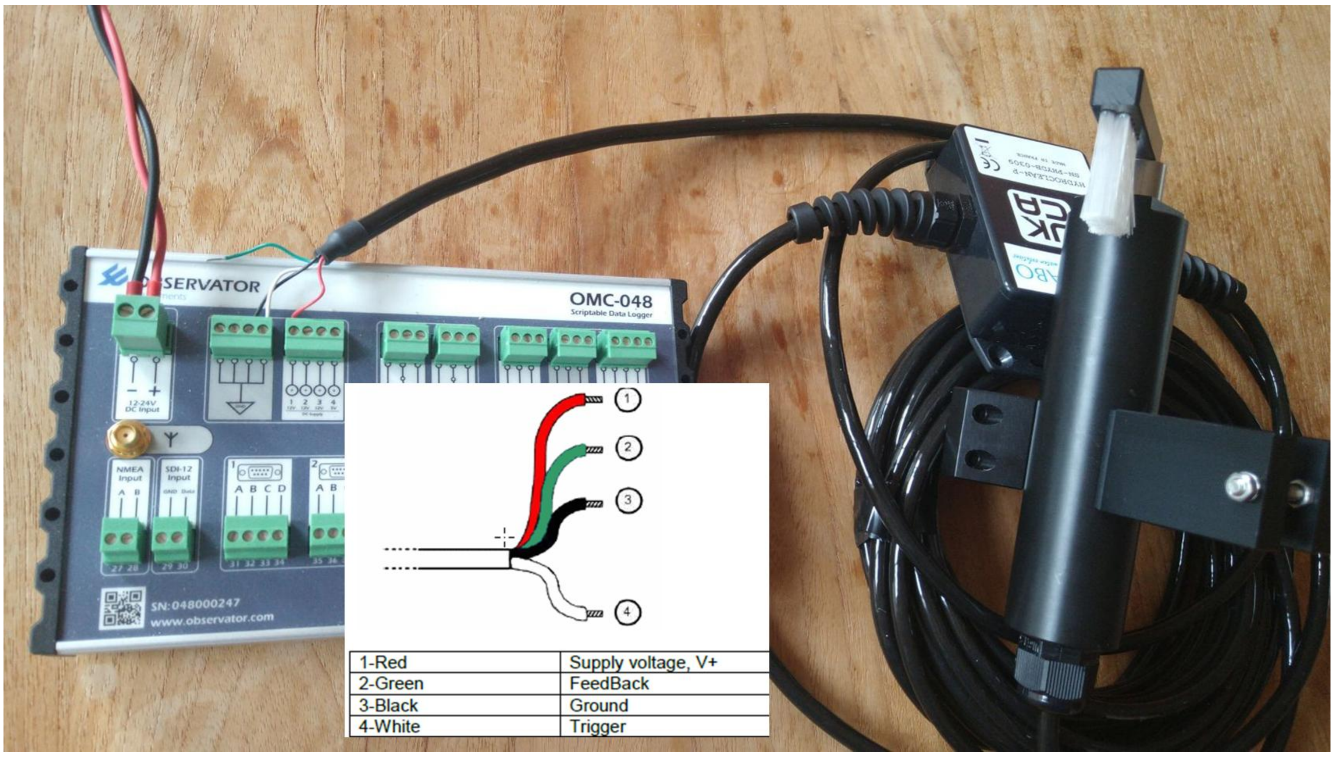
Below a few examples with different wipers are shown. Note that these are just examples; as usual in life, many roads lead to the same destination (often Rome).

All examples have the wiping scheduled to start at second 50 of every minute. This is just for demonstration; not for practical use. For a correct timing in practical applications, consider the following:

- If you wipe too often, the brush wears unnecessarily fast. And you waste power.
- If you wipe too little, the sensor may get too dirty for good measurements.
- If the sample interval is relatively long, it is probably wise to wipe before each measurement (sample). But if you sample often, this conflicts with point 1, so you need to wipe only every now and then.
- During a wipe, you should generally not read data from the sensor, because the brush will disturb the measurement. Also, immediately after the wipe, the removed dirt may still ‘hang around’ and affect the measurement. (This depends on the sensor.) So you may want to leave some time between the wipe and the sample moment.
- Note that some sensors measure (or average) over a significant time. In that case keep enough time between the wiper action and the sample moment.

13.2.1 Aqualabo ‘HYDROCLEAN_P’

Figure 20 shows a test setup with the HydroClean_P connected to the OMC-048. The inset shows the wires of the wiper. Check this with the documentation from Aqualabo for your particular wiper. If you like to use the feedback from the wiper, you can connect the green wire to pin 18 (the 0-24V analog voltage input) of the logger.



1-Red	Supply voltage, V+
2-Green	FeedBack
3-Black	Ground
4-White	Trigger

Figure 20. HydroClean_P connected to the OMC-048.

Use below text block in the config file (also if you are not interested in the feedback signal, see 13.1.3):

```
# The 'feedback' from the sensor is connected the voltage input.
# - A 'low' (0V) should be read if the wipers is okay.
# - A 'high' voltage (12V) indicates failure of the wiper.
Analog_voltage:
- id: wiper
  port: 1
  sample_interval: "50 * * * *"
  min_in: 0
  max_in: 24000
  min_out: 0
  max_out: 24
  supply_port: 1
  supply_port_always_on: False
  startup_time: 3
  log_name: WiperFailure
  log_unit: B
  log_tag: Fail
```

Note that:

- The ‘Trigger’ wire is tied to ground. However, when the power is switched on, the wiper will anyhow perform a single wipe action (move back and forth).
- A wipe action takes about 2 s, so the startup time is set to 3 s.
- It would also be possible to power the wiper directly from the power supply, and use the power port just to trigger a wipe action. However, this wastes a little power (see below).
- With the ‘Trigger’ fixed ‘low’, the wiper only wipes after power on. With the ‘Trigger’ fixed ‘high’, the wiper wipes every 5 s.

We measured a power consumption of 10 mA (at 12V) when idle (powered, but not wiping). During a wipe we measured an average current of around 100 mA during about 2s. When obstructed, current can peak to 200 mA. The power ports can handle this, so no relay is needed.

13.2.2 Trios W55

Figure 21 shows a test setup with the W55 wiper connected to the OMC-048. Check the meaning of the wire colors for your particular sensors. Here the ‘trigger-’ and ‘trigger+’ wires are tied together with the power ‘ground’ and ’12-24V’ respectively.

Note that:

- The trigger signal will be high as long as the power is on. However, the wiper will only wipe once (immediately after power-on).
- A single wipe action consists of the wiper brush moving back and forth twice.
- A wipe action takes 2-3 seconds, so the pulse_duration is set to 4 s.

Since no feedback signal is given by the wiper, the pulse_driver: can be used.

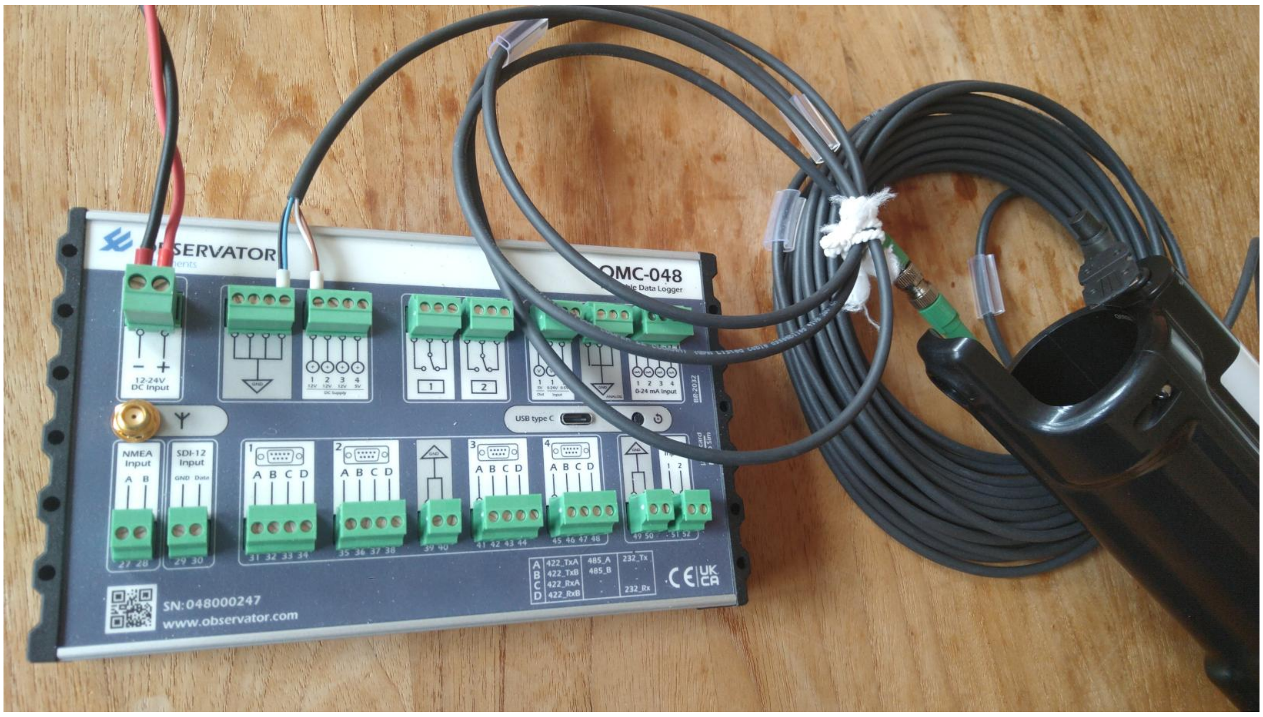


Figure 21. TriOS W55 connected to the OMC-048.

```
pulse_driver:
- id: w55
  sample_interval: "0 0 * * *"
  supply_pulse_port: 1
  pulse_duration: 4
  normally_off: true
```

We measured a power consumption of 20 mA (at 12V) when idle (powered, but not wiping). During a wipe we measured an average current of around 50 mA during about 3s. Currents can peak to 150 mA. The power ports can handle this, so no relay is needed.

13.2.3 Zebra Tech

Figure 22 shows a test setup with the Zebra Tech wiper connected to the OMC-048. Note that this requires the use of a small interface box containing a micro controller. This is delivered by Zebra Tech, together with the wiper. The inset clearly shows this box. It interfaces to the logger with three wires: ground, power and ‘control’.

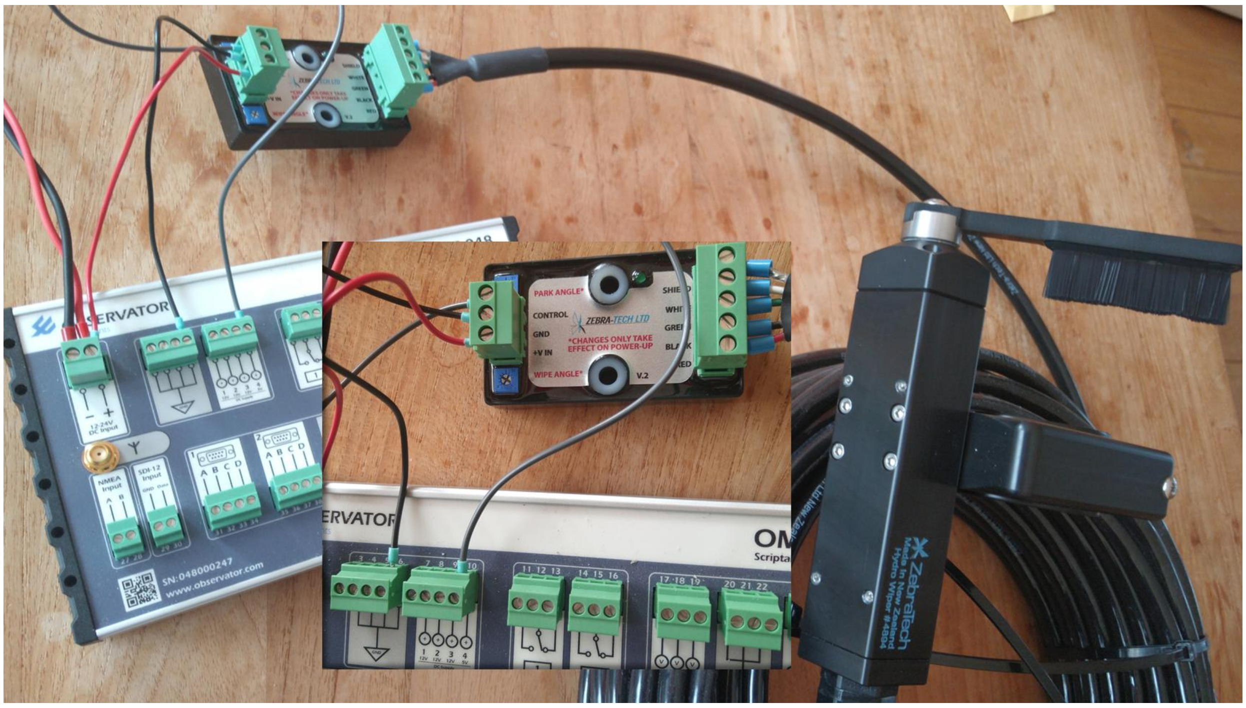


Figure 22. Zebra Tech wiper connected to the OMC-048.

The documentation from Zebra Tech states that the control input needs a signal of 5V, so we use the power port 4 from the logger. (I don't know if the input can also handle 12V). The power pins are connected directly to the main power supply. So we don't switch power.

```
pulse_driver:
- id: zebra
  sample_interval: "0 0 * * *"
  supply_pulse_port: 4
  pulse_duration: 1
  normally_off: true
```

Actually, the pulse_duration can be 2 or more seconds as well. It doesn't matter, but 1 s is more than enough to trigger the wiper, so why make it longer? Note that the wiper may take a second or so before it actually starts. The whole process may take 7 to 10 s, so allow for some time between starting the wipe and taking a sample from the wiped sensor.

The idle current of this wiper is very low (according to the spec, around 0.06 mA), so having constant power doesn't waste much power. During a wipe of around 6 s the average current is around 70 mA. Higher peaks of up to 500 mA can occur according to the manual.

At power-on the wiper will perform one wipe anyway, without receiving a trigger ('control') pulse.

Alternative using the logger's internal relays:
Instead of triggering with a power port, we can also switch the power through a relay. For this, connect the ground and 'control' wires of the Zebra controller to the logger's ground. Connect the '+Vin' from the Zebra controller to pin 12 (middle relay contact). Connect pin 11 (relay 'on' contact) to the power supply of the logger. In the config file, use:

```
pulse_driver:
```

```
- id: zebra
  sample_interval: "0 0 * * *"
  relay_pulse_port: 1
  pulse_duration: 15
  normally_off: true
```

Note:

- The pulse_duration is chosen relatively large, because we want to make sure the wiper receives power during its entire cycle, which may take 10 s.
- No trigger (‘control’) pulse is given, but the wiper will anyway wipe once after power on.
- You can also leave the control wire floating, because the controller has an internal pull-down resistor.

13.2.4 S::can ruck::sack

The ruck::sack wiper from s::can can be connected either to a power output (12 V), or via a relay to a power supply.

The red and purple wires connect to the power +, the black wire to ground.The wiper wipes two cycles in about 6 seconds. Current at 12V is about 100 to 160 mA. Average over the 6 seconds is 140 mA. Stand-by current is about 30 mA. Because the stand-by current is quite large, it is wise to power the whole wiper on and off, rather than using a trigger pulse.

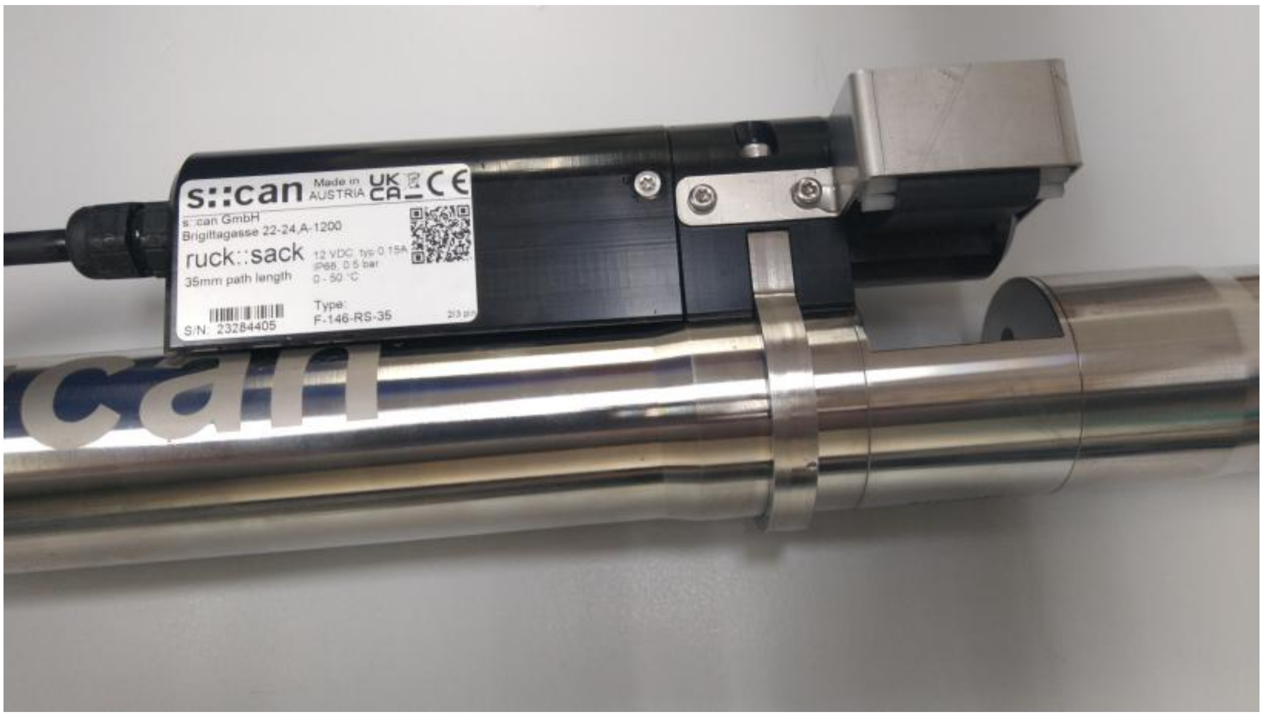


Figure 23. The ruck::sack mounted on a spectroyser (both from s::can).

For configuration, use, for example:

```
pulse_driver:
- id: ruck
  sample_interval: "0 0 * * *"
  supply_pulse_port: 3
  pulse_duration: 10
  normally_off: true
```

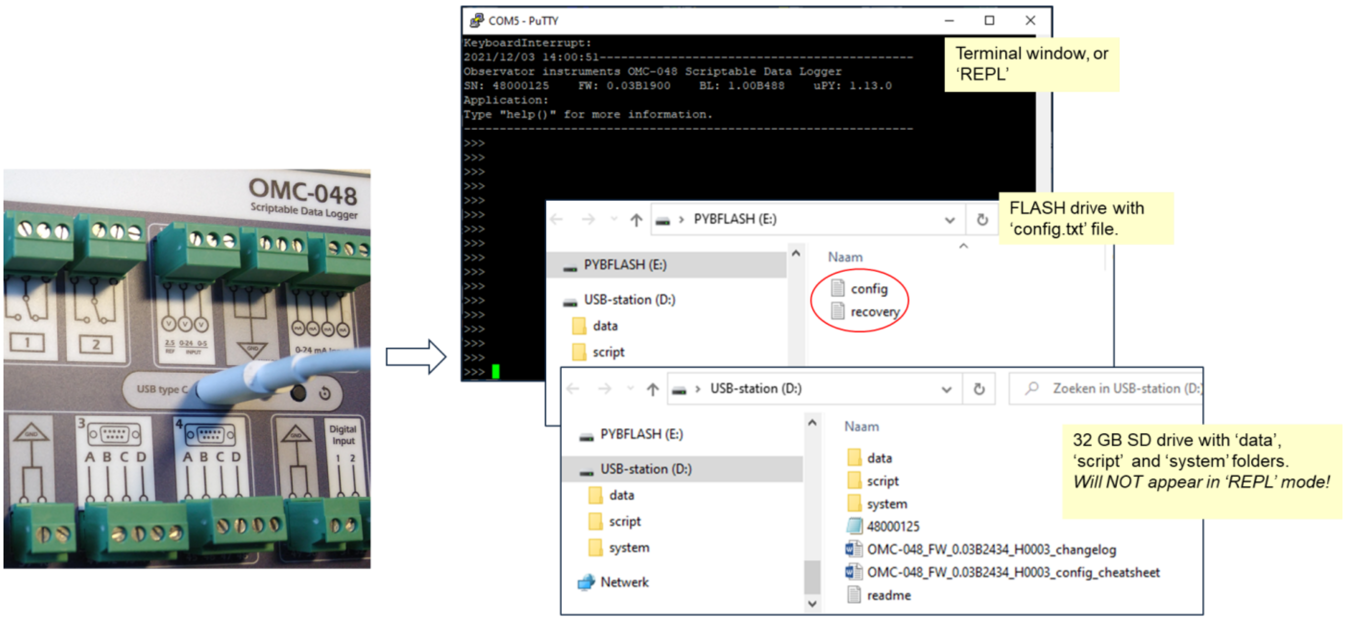
The pulse duration of 10 s is chosen to give some margin on the needed 6 s (in case the wiper is slowed down because of an obstruction). But not too long, to prevent wasting power.

14 Testing and debugging

The OMC-048 offers many advanced features for checking your complete system, from sensor to server. I will first give you a brief overview and then we will go into more detail in the next sections.

In this chapter, I assume you have physical access to the logger. In plain words: you can put your hands on the logger. So you can plug-in a USB cable. But most of the things described here are also possible with remote control, as described in chapter 16.

You still remember what happens if you connect a USB cable to the logger? Have look at Figure 24 (repeated from Figure 2). With USB connected, you can open a terminal window (or REPL) to the logger. Also, the flash drive with the config.txt file will show up in your windows file explorer. Maybe the SD card will also show up (refer to section 3 on USB modes).



```
KeyboardInterrupt:
2021/12/03 14:00:51---------------------------------------------------
Observator instruments OMC-048 Scriptable Data Logger
SN: 48000125    FW: 0.03B1900    BL: 1.00B488    uPY: 1.13.0
Application:
Type "help()" for more information.
---------------------------------------------------------------
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```

OMC-048
Scriptable Data Logger
USB type C
Digital Input
COM5 - PuTTY
Terminal window, or 'REPL'
PYBFLASH (E:)
USB-station (D:)
data
script
Naam
config
recovery
FLASH drive with 'config.txt' file.
USB-station (D:)
Zoeken in USB-station (D:)
PYBFLASH (E:)
USB-station (D:)
data
script
system
Netwerk
Naam
data
script
system
48000125
OMC-048_FW_0.03B2434_H0003_changelog
OMC-048_FW_0.03B2434_H0003_config_cheatsheet
readme
32 GB SD drive with 'data', 'script' and 'system' folders.
Will NOT appear in 'REPL' mode!

Figure 24. A USB connection opens up a COM port for the REPL, as well as one or two (disk) drives.

The three main tools for testing and debugging are briefly described below and are detailed in the next sections.

Remember that you need to apply external power to the logger if you want to run the logger application. USB power alone will not do.

The terminal window (REPL)
In the terminal window, you can see what the logger is doing at that moment. So it gives you a 'live view' on what's going on. You can also stop the logger application and start specific debugging actions, including direct serial communications with connected sensors.

The system log files
The disadvantage of the 'live view' is, well, it is 'live'. What happened a while ago is no longer visible. And if the configured intervals are large (like an hour or so), you may need to wait a (very) long time before something happens. For this, you have the system log files on the SD card. Don't confuse these with the files containing the logged data. The system log files contain information similar to what shows up in the

terminal window, but you can find past information per day. So you can look back what happened ‘5 days ago around 15:00 hours’.

The startup self-test
Another useful file is the ‘self test report file’. Each time the logger application is started, the logger performs a self-test (if configured to do so). This test typically includes communication with the sensors and a test FTP or TCP transmission. The file is stored on the SD card and is transmitted by FTP or TCP (if possible).

14.1 The REPL with a running application.

When the application is running, all you can do in the REPL is: sit back, relax, watch and enjoy. The application *may* print all kind of messages on the screen for you to watch. Whether it actually *does* print messages depends on the situation.

In the “Omc048:” section in the config.txt file, is the line:
```
repl_log_level: info      #debug/info/warn/error/fatal
```

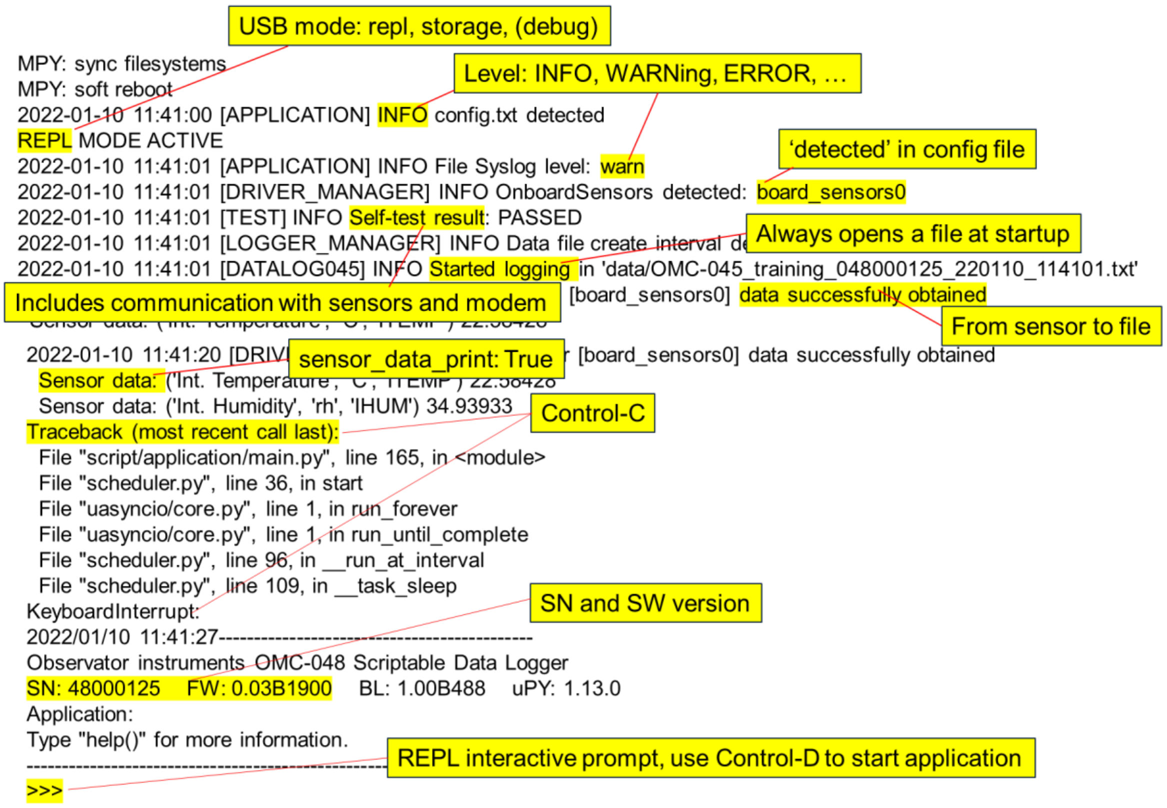
The ‘repl-log-level’ determines the ‘level’ that messages should minimally have to be printed in the screen. ‘Fatal’ is the highest level and these messages are always printed. The five possible levels have the following meaning (from lowest to highest level):

1. **Debug**: Print all of the below, plus information that may be relevant for **debugging** the software. Not recommended for normal use (too much information).
2. **Info**: Print all of the below, plus **information** about what the application is doing. This is the recommended setting if you want to keep a close look of what is going on.
3. **Warning**: Print all of the below, plus **warnings**. This is recommended if you want to keep the terminal on for a long time and you want to see only potential problems (warnings) or worse.
4. **Error**: Print all of the below, plus **errors**. This is recommended if you want to keep the terminal on for a long time and you want to see only real problems (errors) or worse.
5. **Fatal**: Print only **fatal** error. If you get these, you are in big trouble.

Note that you can independently set a ‘log level’ for the log files in the same way. See section 14.3.

Generally, I would recommend the level ‘info’. That level is also used in the example below (Figure 25). The first few lines of the example are typical for starting the logger. Once the logger application is running, most lines are of the form:
<timestamp> [<software routine>] <level> <message text>

When the logger is interrupted with Control-C, you will see some Python traceback lines (only relevant for programmers), followed by some text with the logger’s serial number and FW version number and finally by the typical REPL interactive prompt.



USB mode: repl, storage, (debug)
Level: INFO, WARNing, ERROR, …
'detected' in config file
Always opens a file at startup
Includes communication with sensors and modem
From sensor to file
sensor_data_print: True
Control-C
SN and SW version
REPL interactive prompt, use Control-D to start application

```
MPY: sync filesystems
MPY: soft reboot
2022-01-10 11:41:00 [APPLICATION] INFO config.txt detected
REPL MODE ACTIVE
2022-01-10 11:41:01 [APPLICATION] INFO File Syslog level: warn
2022-01-10 11:41:01 [DRIVER_MANAGER] INFO OnboardSensors detected: board_sensors0
2022-01-10 11:41:01 [TEST] INFO Self-test result: PASSED
2022-01-10 11:41:01 [LOGGER_MANAGER] INFO Data file create interval de
2022-01-10 11:41:01 [DATALOG045] INFO Started logging in 'data/OMC-045_training_048000125_220110_114101.txt'
                                                             [board_sensors0] data successfully obtained
  Sensor data: ('Int. Temperature', 'C', 'ITEMP') 22.58428

 2022-01-10 11:41:20 [DRIV                            r [board_sensors0] data successfully obtained
   Sensor data: ('Int. Temperature', 'C', 'ITEMP') 22.58428
   Sensor data: ('Int. Humidity', 'rh', 'IHUM') 34.93933
 Traceback (most recent call last):
   File "script/application/main.py", line 165, in <module>
   File "scheduler.py", line 36, in start
   File "uasyncio/core.py", line 1, in run_forever
   File "uasyncio/core.py", line 1, in run_until_complete
   File "scheduler.py", line 96, in __run_at_interval
   File "scheduler.py", line 109, in __task_sleep
 KeyboardInterrupt:
 2022/01/10 11:41:27-------------------------------------------
 Observator instruments OMC-048 Scriptable Data Logger
 SN: 48000125   FW: 0.03B1900    BL: 1.00B488    uPY: 1.13.0
 Application:
 Type "help()" for more information.
 ---------------------------------------------------------
 >>>
```

Figure 25. Annotated example of REPL output of a logger that is just started and then interrupted.

By studying the text in the REPL window, you can learn a lot about what is going on. If you have set '`self_test: True`' (as is generally recommended), always look for the line 'Self-test result: PASSED' (or FAILED if it is not your day). Pay particular attention to the lines in the first minute or so after starting the logger. I usually find it convenient to capture the text from the terminal window and copy-paste it to Notepad, so I can carefully read the lines without being bothered by the constant scrolling of the live window.

14.2 The REPL in interactive mode.

When in interactive mode, you will see the typical REPL prompt '>>>' waiting for you. You can now enter Python commands and even write (small) programs. This manual is not aimed at Python programmers. Therefore I will just describe a few very handy and useful things that you can do by simply copying the given text lines.

When typing in the REPL, you can hit the 'up-arrow' key on your keyboard to get back the previous line you entered. Hit the key several times to go back several lines.

I recommend you copy the Python lines from the following two sub-sections into a .TXT file on your computer. Then you can easily copy-paste what you need into the REPL. If you use them a lot, also have a look at chapter 17 on programming.

14.2.1 Controlling the Power Ports

You can control (switch on and off) any of the four power ports with the below lines:

```
import omc048                 # Import the omc048 module
p = omc048.power_supply(1)    # Make ‘p’ refer to Power Port 1

p.on()                      # Switch Power Port on
p.off()                     # Switch Power Port off
p.toggle()                  # Toggle the Power Port on/off
```

- You have to do ‘`import omc048`’ only once per interactive session (thus until you hit Control-D).
- Once you entered the first two lines, you can use ‘`p.on()`’, ‘`p.off()`’ and ‘`p.toggle()`’ as many times as you like.
- You can do the same for the other power ports, by just replacing the 1 by 2, 3 or 4. If you want to use multiple ports at the same time, use p1, p2, …. instead of just ‘p’.
- Note that the Power Ports can only supply power if the power supply to the logger is switched on. They cannot be powered from USB power..

14.2.2 Controlling the Relay Ports

You can control the relay ports in a similar way as the power ports.

```
import omc048               # Import the omc048 module
r = omc048.relay(1)         # Make ‘r’ refer to relay 1

r.on()                      # Switch relay on
r.off()                     # Switch relay off
r.toggle()                  # Toggle the relay on/off
```

Note that a relay port will retain its last position after the logger is switched off.

14.2.3 Serial port redirection (tunneling)

You can create a tunnel from your terminal program to any of the serial ports with the below lines. If you have connected a sensor (or other device) to a serial port, the tunnel allows you to enter commands that go straight to the sensor. And the sensor response will show up on your screen immediately. As if the sensor was directly connected to a COM port of your computer.

If you have specific (configuration) software that uses a COM port to communicate with the sensor, you can use this software together with the tunnel. But this software should not issue a Control-C character to the sensor, because this will close the tunnel (see below).

You may be familiar with USB-to-RS232 (or USB-to-RS422) adapters. They also create a COM port on your computer. The OMC-048 can do the same. This can be particularly convenient if you don’t have such an adapter (or ‘converter cable’) at hand. But also to communicate with a sensor (or other device) that is already connected to the OMC-048. You just need to stop the application (Control-C) and enter the below lines.

Is your sensor powered from a Power Port? Then switch on the Power Port first and wait till the sensor is awake before trying to communicate.

```
import omc048             # Import the omc048 module
s = omc048.serial(1)      # Make ‘s’ refer to Serial 1
s.init(9600, s.RS232)     # Initialize the port with 9600Bd and RS232
```

```
s.redirect()            # Start the redirection (open the tunnel)
```

- You have to do '`import omc048`' only once per interactive session (thus until you hit Control-D).
- Once you entered the first three lines, you can use '`s.redirect()`' to start the redirection and **Control-C** to close the redirection repeatedly.
- Once the tunnel is open, it is possible to close the terminal program on your PC and to start a specific configuration program for the connected sensor. Once finished with this program, you can close it and open the terminal program again. Then you can use Control-C to close the tunnel.
- You can do the same for the other serial ports, by just replacing the 1 by 2, 3 or 4. It is even possible to create a tunnel to:
    - nmea port (5). Use baud rate 4800.
    - sdi12 port (6): Use `s6.init(1200, s.RS232, bits=7). This is not a useful redirection, because there is no 'tool' to issue SDI-12 commands like 'break'.`
    - internal cellular modem (7). Use baud rate 115200. To power the modem, use:
      ```
      m = omc048.modem()
      m.power(1)      # 1=on, 0=off
      ```
      You can use this if you are familiar with AT commands, or have a tool for this.
- If you want to change the baud rate or the mode (from RS232 to RS422, for example), first close the tunnel (Control-C), then enter for example '`s.init(4800, s.RS422)`' and then reopen the tunnel with '`s.redirect()`'.

14.3 The system log files

The 'system log files' should not be confused with the data files. The system log files are located in the /system folder on the SD card. Each day a new file is created. They contain information that is similar to what you see passing by in the REPL, when the logger application is running.

The major difference with the REPL itself, is that the log files allow you to see messages that were generated in the past. So you can look for an event that happened on a certain date & time. The REPL is 'live'.

As described in sub-section 14.1, you can specify a threshold for the level of messages that you want to see in the REPL. Similarly, you can define a threshold for the messages that should be printed to the log files. You do this in the 'Omc048:' block in the config file: '`file_log_level: info`'. Refer to 14.1 for a description of the possible levels.

I recommend to use level 'info' for debugging of for systems that you want to keep a close eye on. Once you have confidence in your system, I can recommend increasing the level to 'warn', so the log files don't get unnecessarily big.

14.4 The startup self-test

Each time the logger application is started, the logger performs a self-test (if configured to do so). This test typically includes communication with the sensors and a test FTP or TCP transmission. The file is stored on the SD card and is transmitted by FTP or TCP (if possible).

You enable the self-test in the 'Omc048:' block in the config file: '`self_test: True`'. Obviously, this delays the start-up of the application, because the tests may take some time. Particularly if you have sensors that may be slow to respond (like GPS/GNSS) and cellular communication is included. But once the application is up and running in the field, it is highly recommended to have the self-test on.

Each time a service engineer has done maintenance to the sensors and starts the logger again, the self-test will show if everything is working again. You don't have to wait for a scheduled sample or transmission to see if the sensor and the modem are working well, because they will be tested during the self-test.

You can find the self-test result files in the /data folder on the SD card. If the cellular modem is properly configured and is able to communicate with a server, it will also transmit the file to the server.

A simple self-test result file may look like this:

```
{
  "SerialNumber": "048000125",in
  "DateTime": "2022-06-20T15:46:52.729Z",
  "Results": {
    "FTP": "PASSED",
    "SSL": "PASSED",
    "board_sensors": "PASSED",
  },
  "SelfTest": "PASSED"
}
```

I reformatted the file a little for readability. Actually, the file format is called 'JSON' and is suitable for interpretation by software like Blue2Cast.

15 Updating the software and firmware

15.1 When to update?

We recommend updating to the latest software (SW) and firmware (FW) whenever you start a new application/project. You may also update if the new release includes new features or improvements that you want to make use of in a running application/project.

But if you have an application that is running well, you should normally not update the software (SW) and firmware (FW).

15.2 When to format the SD card?

If you have been testing & debugging using the USB 'debug' mode, it is wise to format the SD card before going to a real deployment.

Otherwise, there is no need to format the SD card. However, if you want a complete 'fresh' start for a new project, you may want to start with a complete empty and clean SD card. In that case formatting is just as easy, and a bit more thorough than deleting everything.

Before formatting, make sure there is no valuable data on the card that you did not yet save somewhere! Also make sure you have saved your reference table files and other customized files.

To format the SD card, simply plug in in the USB cable and make sure the USB mode is set to 'storage'. Then select 'format', as shown in Figure 26.

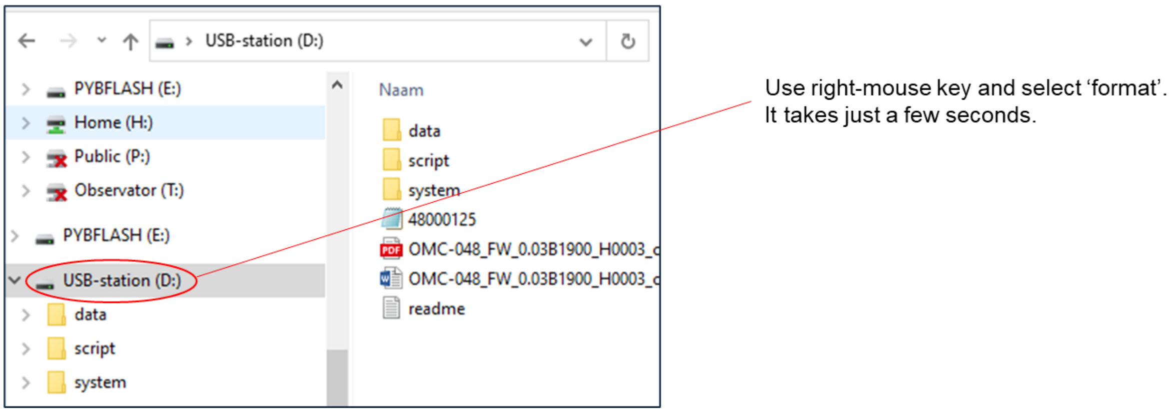


Figure 26. Formatting the SD card.

15.3 How to update?

Basically, all you need to do to update the SW & FW, is to copy a new 'script' folder (SW) and a new .bin file (FW) to the root (top) of the SD card. That's all. It is shown in Figure 27. But there are a few more details that may be relevant.

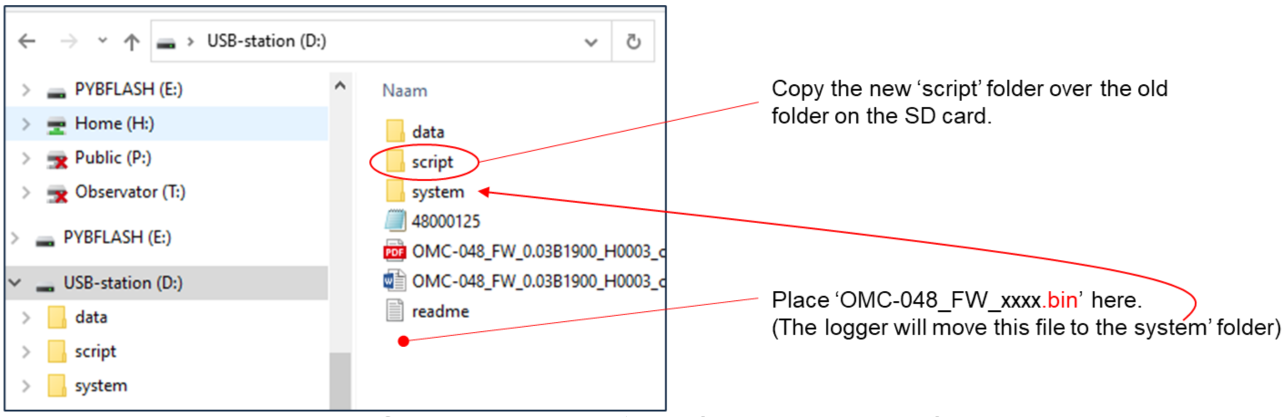

USB-station (D:)
PYBFLASH (E:)
Home (H:)
Public (P:)
Observator (T:)
PYBFLASH (E:)
USB-station (D:)
data
script
system
Naam
data
script
system
48000125
OMC-048_FW_0.03B1900_H0003_c
OMC-048_FW_0.03B1900_H0003_c
readme
Copy the new 'script' folder over the old folder on the SD card.
Place 'OMC-048_FW_xxxx.bin' here.
(The logger will move this file to the system' folder)

Figure 27. Installing new FW and SW is just copying the 'script' folder and the xxx.bin file.

First of all, to be able to do anything with the SD card, you need to have access to the SD card. This means the logger must be in 'storage' mode. So you may need the change the USB mode in the config file to 'storage' and restart the logger (Control-C to stop, Control-D to start), just to get access to the SD card.

Once you have access to the SD card, you can copy the script folder from the release 'over' the script folder on the SD card. Then copy the new xxx.bin file to the root (top) of the SD card (see Figure 27). You can now reset the logger in three ways:

1. Type 'obs.hard_reset()' in the REPL
2. Hard reset by pushing the small reset button on the logger.
3. Switch power on and off (unplug USB + external power and plug-in again).

The logger will now take a bit more time in starting-up than it normally does, because it is installing the new FW first. In doing so, it will move the xxx.bin file to the 'system' folder. So, in this folder you can always find previously installed FW. In the REPL, the logger will report its new version number.

Note that the release contains a few more files than just the script folder and the xxx.bin file. For example, there may be release notes (or a 'changelog'), documentation, example config.txt files or a 'config-cheatsheet', and so on. If you find it convenient to have these on the logger, you can copy them to SD as well, but personally I prefer to have them just on my computer for reference and to keep the SD clean. It's your choice.

15.4 What about the 'generic_tables'?

If you used generic_tables, remember that you once made a copy of a 'xxx_reference_tables_*template*' file, and renamed it to 'xxx_reference_tables'. This later file (without the '_*template*' ending) was the one you customized and is the one that is used by the logger.

The new release may include an updated template file. So make a copy of this new template and copy-paste the tables from your old customized file to the new one.

By the way, it is always wise to have copies of your customized files on your computer. This allows you to modify them on your computer and then to copy them to all your loggers.

15.5 What about the 'config.txt' file?

A new release will mostly mean 'additions', so your old config files may still work on the new release. However, sometimes certain blocks in the config file may need to be changed. So please always check the documentation of the new release.

If you are starting a new project/application, it is wise to create a new config from scratch, using the 'config_template.doc' file that you can download from the OMC-048 product page.

16 Remote control

16.1 Overview

The data logger can be remotely controlled with our Blue2Cast software. This is a web-based application that runs on a server. If configured to do so, the logger periodically connects to this server to see if there are users waiting to take control. Users can connect to the server by visiting its webpage. They can see the available loggers and indicate which logger they want to take control of.

Once a connection is established between the user and the logger (actually, the server is always in between), the user can copy, edit and delete files, update the firmware, modify the config file and get access to the REPL. Access to the REPL includes using port redirection (tunneling).

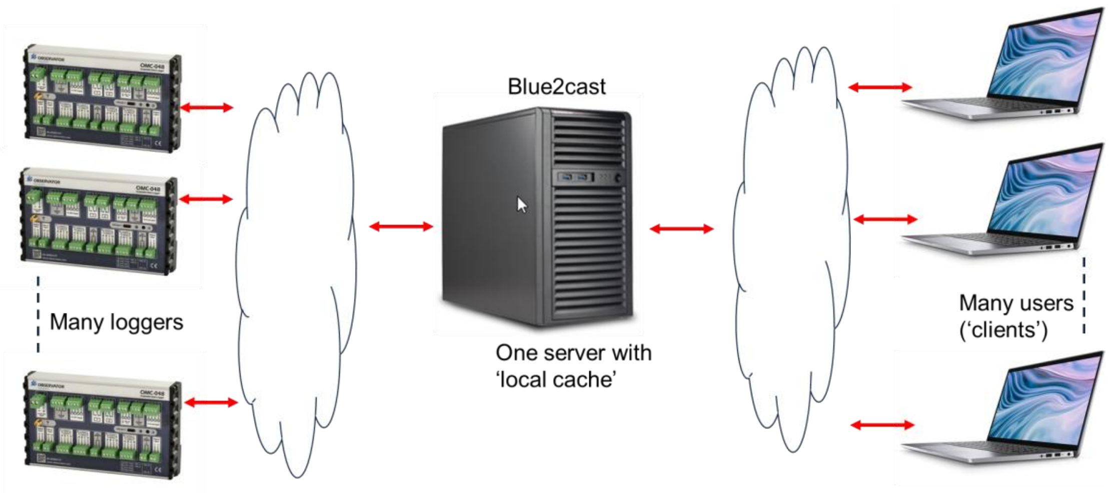


Figure 28. Overview of Blue2Cast Remote Control

All communication is secure (encrypted) and password protected. Remote control is only available with cellular communication.

At this moment, Blue2Cast is only available as a hosting service by Observator. It is being developed into a tool that can be licensed to run on customers' servers as well.

The Blue2Cast manual is still to be written.

16.2 Preparation for remote control

To prepare the logger the first time for remote control, you need to take the following one-time-only steps:

1. Make sure the logger is properly configured for cellular communication. Thus the SIM card is installed, the modem and the connection settings are correct, the antenna is connected and so on. So, everything just like your normal application, without remote control.
2. To enable remote control, set "webserver_access: True" in the block 'Connection_settings:' in the config file. The relevant lines should look like this:

```
webserver_access: True
webserver_url: blue2cast.com
webserver_port: 5000
```

```
        webserver_password: <OBSERVATOR GENERATED HASH>
```

3. Create a 'recovery.txt' file on the FLASH, next to the config.txt file. An example of a recovery.txt file is shown below. Copy the SIM details (in the 'modem:' block) from the config file to the recovery file.
4. Now (re-)start the logger application (with external power applied). The logger will try to contact the Blue2Cast server immediately. But, as you can see in the config file, the logger does not yet have a valid password. So the Blue2Cast server will deny access every time. Leave the logger running.
5. Contact Observator with the serial number of your logger. We will tell Blue2Cast to accept the next contact request from this particular logger for initial setup. As soon as the logger tries to contact again, Blue2Cast will take control of the logger and will perform the following actions:
    a. The `webserver_password` is set in the config file (thus the config file will be changed)
    b. The same is done for the 'recovery.txt' file.
    c. The recovery.txt file is copied to internal memory.
    d. The logger application is restarted with the new config file.

If you have the REPL open while this all happens, you can see it happening.

After that, the logger is ready for remote control. Each time the modem is switched on (for a regular data transmission), the logger will also contact Blue2Cast to check if someone wants to take control.

This is an example of a recovery.txt file. You need to fill-in the SIM details in the 'modem:' block. You can change the transmit_interval if you like.

```
# ----System---- #
Omc048:
  usb_mode: debug
  system_id: RECOVERY
# ----Modem-Settings---- #
Modem:
  id: onboard_modem
  port: modem
  sim_username:
  sim_password:
  apn:
# ---- External connection settings ---- #
Connection_settings:
  webserver_access: True
  webserver_url: blue2cast.com
  webserver_port: 5000
  webserver_password:  <TO_BE_GENERATED>
  transmit_interval: "0 0,5,10,15,20,25,30,35,40,45,50,55 * * *"
```

The purpose and use of the recovery.txt file is explained in the next section.

16.3 Recovery

Suppose the logger is somewhere on a remote island. But within cellular reach and remote control enabled. Further suppose that you have made a change to the config file using remote control, but you also accidently made a mistake in the file. After you restart the logger, the application crashes, because of the mistake. Now the logger will not make contact again with the Blue2Cast server, because the application is not running (it crashed). So now you have to go to the remote island to fix the config file by USB. Depending on the island, this may or may not make you happy.

To avoid losing the connection in case of an error (a 'crash' of the logger application, for whatever reason), we introduced the *recovery.txt* file. This file is located on the FLASH drive, next to the config.txt file. If the application crashes with the config.txt file, it will restart using the recovery.txt file. The recovery file is similar to the config file, but it is wise to just include the minimal number of lines needed for the logger to make connection to Blue2Cast. The more you put in, the more change you introduce something that causes an error.

If you accidently removed the recovery file from FLASH, or something is wrong with it, the application will try to recover from the recovery.txt file stored in *internal memory*. This memory is not accessible to you and you cannot see anything of it, except for the recovery file. So normally you will be unaware of its existence. During preparation for remote control, Blue2Cast will copy the recovery.txt file from Flash to the internal memory. You can also do this manually in the REPL:

- First make sure the recovery.txt on FLASH is correct.
- In the REPL, in interactive mode (you may need control-C to stop the running application), type:
  - `file = open('/flash/recovery.txt')`
  - `obs.recovery(file.read())`
- You may verify the contents of the file in internal memory by typing `obs.recovery()`.

As mentioned before, Blue2Cast will do this for you during preparation. So you only have to do it manually if you want to change the file, for example because you have the change the 'modem:' settings (new SIM card or new APN, for example).

So, to summarize:

- The application will try to run with the following files in this order:
  1. First it will try the config.txt file (on Flash). If this fails,
  2. it will try the recovery.txt file (on flash). If this fails,
  3. it will try the recovery.txt file (on internal memory).
- During 'preparation for remote control', Blue2Cast will fill-in the password in files the config file and in the recovery file, and it will copy the recovery file to internal memory.
- You can edit the recovery.txt file on Flash, and you can manually copy it to internal memory.

Make sure you have the recovery files correct while the logger is on your desk and before you ship it to some remote uninhabited island!

16.4 Removing remote control

Why would you ever want to remove the remote control option? Reasons could be:

- A very sensitive application, where you don't want to risk that any of your colleagues accidently changes something.
- You switch from cellular communication to Iridium or radio communication. So remote control is no longer possible and you don't want the logger to waste time and power on uselessly trying to make a cellular connection.

Basically, all you have to do to disable remote control, is to set webserver_access to False in the config file: `webserver_access: False`
Or, if you don't use cellular at all, you can completely remove the 'Connection_settings:' block from the config file. In either case, the logger will not try to connect to the webserver for remote control.

16.5 Removing the recovery files

It is highly recommended to remove the recovery files if you change to a configuration without cellular communication. For example, when using Iridium or just local storage on SD.

The reason is this:
If the logger application crashes for whatever reason, it will try to restart and reconnect to the Blue2Cast server, using the recovery files as described before. Without cellular communication this will fail, so the logger will keep trying forever. Thereby wasting a lot of (battery) power.

To avoid this, you need to remove both recovery files:

- Recovery.txt can be removed from FLASH by simply deleting it, or by renaming it.
- Recovery.txt on the internal memory can be cleared (emptied) by typing `obs.recovery("")` in the REPL.

In case you may want to restore remote control later on, make sure to save the webserver_password somewhere. If you lose it, you need to go through the 'preparation for remote control' procedure again.

17 Programming the OMC-048

This manual is about configuring the logger and not about programming it. Programming is for Python programmers. But the logger allows you to write your own Python programs. This chapter is just an appetizer for programmers. I will first give some general information on programming and then I will give three small examples. The first one is to change your OMC-048 into an USB-to-RS232 (or 422) converter!

17.1 General info on programming

As mentioned in the section ‘What is the REPL?’, the OMC-048 is a small computer with a lot of input-output capabilities. Just like your computer, it can run many different application programs (apps).

Whenever the OMC-048 is started, it runs the main.py program located in the \script\ application folder. It is this program that turns the OMC-048 into a data logger. This main.py program uses the .py files located in \script\modules, reads the config.txt file and writes the data files and the system log files and so on.

If you replace the main.py file by ‘something else’, then the OMC-048 will be that ‘something else’. The examples in this chapter are simple main programs that will not read the config file nor need any of the .py files in \script\modules. You could delete all of these, if you just want to run the examples. In that case, you can forget practically everything else in this manual, because it all applies to the logger application.

Before using any of the following examples, first rename the current *main.py* into *main_logger.py* or something. So you can change it back later, when you want your logger back.

If you want to make serious programming effort, I recommend using a language sensitive text editor. You could start with Notepad++. More information can be found in the online ‘*Software manual OMC-048*’ on our website.

17.2 USB to RS232/422 converter

Suppose you need a USB to RS232 or 422 converter, but you don’t have one at hand. But you do have an OMC-048. We already saw in sub-section 14.2 which commands you need to open a tunnel. Well, if you simply put those commands in a file called main.py and place it in the \script\application folder, you will get a tunnel every time the logger starts!

After you saved a copy of the original main.py file, copy-paste the below text in a file called main.py and place it in the \script\application folder.

```
import omc048
p = omc048.power_supply(1)
p.on()
s = omc048.serial(1)
s.init(9600, s.RS232)
s.redirect()
```

Now hit Control-D to start the application. Each time you power-on the logger this application will run automatically. The last line will open the tunnel and the program will stay at that line till the tunnel closes. When you hit Control-C, the tunnel closes and the program finishes and you will be in the REPL. In the REPL, you can type `p.off()` to switch the power off and `s.redirect()` to reopen the tunnel.

Note that the logger needs power (not just USB power) to power its Power Ports.

17.3 A small flashing light

Suppose you need a small blinking LED, but you don't have one at hand. Just suppose. But you do have an OMC-048.

After you saved a copy of the original main.py file, copy-paste the below text in a file called main.py and place it in the \script\application folder.

```
import omc048
import obs

led = omc048.LED()
led.colour('magenta')

while 1:
  led.toggle();
  obs.delay(250)
```

Make sure you don't change the two leading spaces in the last two lines!

Now hit Control-D to start the application. Each time you power-on the logger this application will run automatically. And you will have your blinking LED! Note that the program has an infinite 'while' loop, so it will never end. You have to type Control-C to interrupt the program. Then you will be in the REPL, where you can keep typing '`led.toggle()`' forever, to manually keep the LED blinking.

17.4 A level controller or thermostat

This slightly more complex example can be used when you have an analog 4-20 mA sensor and you want to switch some device on/off depending on the level detected by the sensor. For example: The sensor is a water level sensor and you want to switch a pump on/off, to keep the water level around a preset value. Or it is a temperature sensor and you want to switch a heater (or a fan) on/off to keep the temperature within limits.

After you saved a copy of the original main.py file, copy-paste the below text in a file called main.py and place it in the \script\application folder. Again, don't change the number of leading spaces/tabs.

```
import omc048
import obs
from power_monitor import Hysteresis

adc = omc048.analog_current(1)
hys = Hysteresis(adc, 9, 11)

led = omc048.LED()
led.colour('blue')
led.on()

p = omc048.power_supply(1)
p.on()

omc048.relay(1).off()
state = False
```

```
while 1:
  x = hys.check()
  if x:
    if state == False:
       led.colour('red')
       omc048.relay(1).on()
       state = True
  else:
    if state == True:
       led.colour('blue')
       omc048.relay(1).off()
       state = False
  obs.delay(500)
```

This example uses analog current port 1 (pin 23), Power Port 1 (pin 7) and switches relay 1 (pins 11, 12, 13). The analog level will cause the relay to switch when it goes below 9 mA and to switch back when it goes above 11 mA. The LED on the logger switches together with the relay, so it indicates the state of the relay.

When you start this program (with Control-D or after power on), it will switch on the Power Port, switch the LED on in blue, and put the relay in its ‘off’ state. Then it will go in an infinite loop, where it checks the level every 500 ms and switches the relay + LED accordingly.

When you hit Control-C, you will enter the REPL and you can manually control the relay by typing, for example ‘`omc048.relay(1).off()`’.

18 Appendix 1: Data files

18.1 Data files created with ‘Data_file:’

These files are used for storing (logging) all sensor data for applications that either use cellular communication or no communication at all. For Iridium communication, refer to the next section.

If FTP (over cellular communication) is used, the file that is stored on the SD will be transferred to the FTP server ‘as is’. If TCP is used, the contents of the file is transferred to the Blue2Cast data base, but you cannot find the file itself on the server, like you can with FTP.

18.1.1 The file name

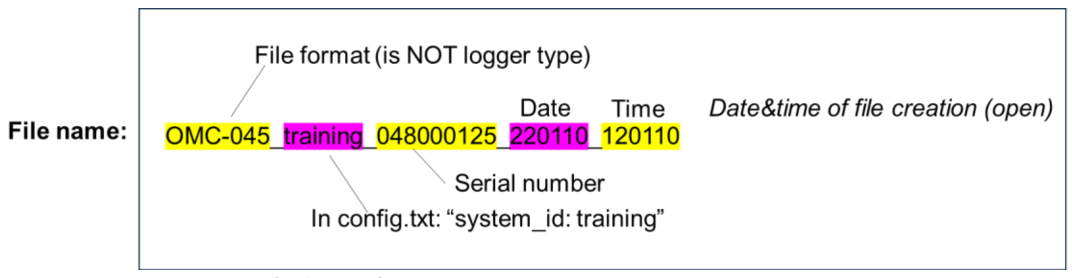
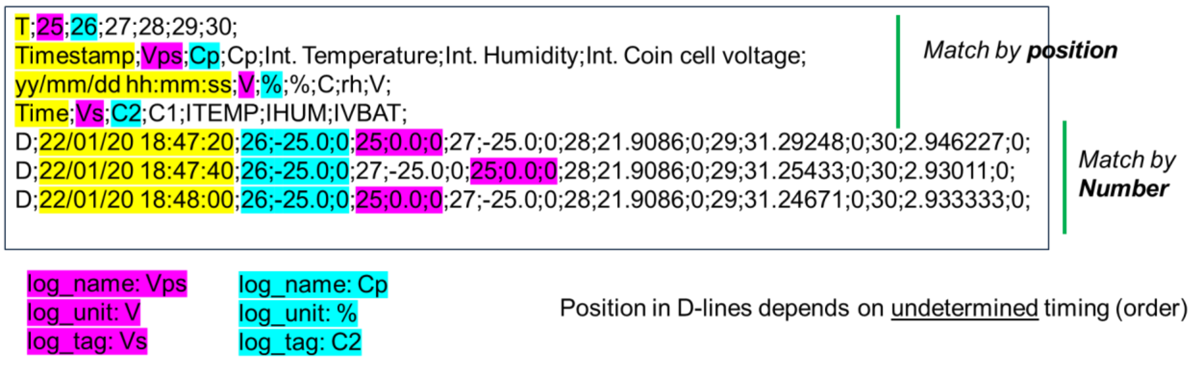

File name:
File format (is NOT logger type)
Date
Time
Date&time of file creation (open)
OMC-045_training_048000125_220110_120110
Serial number
In config.txt: “system_id: training”

Figure 29. File name of a ‘Data_file’

18.1.2 The file contents

Normally you should not need to ‘read and interpret’ a data file. Blue2Cast or other tools will read the data and present it to you in a ‘human-friendly’ format. But for debugging (and for programmers) it may be useful to understand the so called ‘OMC-045’ format (named after the logger for which it was developed).

Please have a look at Figure 30 for the details.



```
T;25;26;27;28;29;30;
Timestamp;Vps;Cp;Cp;Int. Temperature;Int. Humidity;Int. Coin cell voltage;
yy/mm/dd hh:mm:ss;V;%;%;C;rh;V;
Time;Vs;C2;C1;ITEMP;IHUM;IVBAT;
D;22/01/20 18:47:20;26;-25.0;0;25;0.0;0;27;-25.0;0;28;21.9086;0;29;31.29248;0;30;2.946227;0;
D;22/01/20 18:47:40;26;-25.0;0;27;-25.0;0;25;0.0;0;28;21.9086;0;29;31.25433;0;30;2.93011;0;
D;22/01/20 18:48:00;26;-25.0;0;25;0.0;0;27;-25.0;0;28;21.9086;0;29;31.24671;0;30;2.933333;0;
```

```
log_name: Vps      log_name: Cp
log_unit: V        log_unit: %
log_tag: Vs        log_tag: C2
```

Figure 30. The ‘OMC-045’ file format.

18.2 Data files created with ‘Iridium_file:’

These data files are used to store (a selection of) the data from the sensors for subsequent transmission by the external Iridium modem. The format is much more compact than the format described in the preceding section, to save on the data transmission cost.

The current Iridium file format is identical to the format used with the (now retired) OMC-045-3 logger and is compatible with OMC-Data-Online.

18.2.1 The file name

The file name is constructed as shown in Figure 31. This the file name on the SD card. Once transmitted over the Iridium network, it depends on your provider how you will receive the file.

File name:

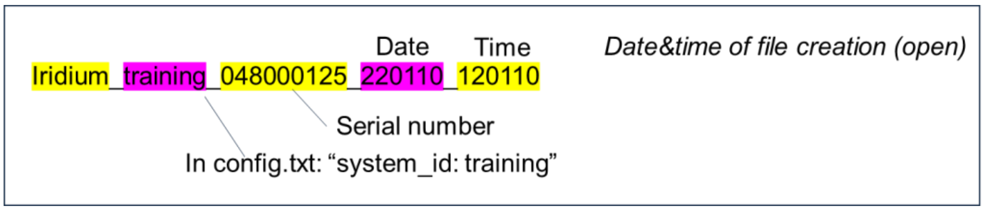

Date
Time
Date&time of file creation (open)
Iridium_training_048000125_220110_120110
Serial number
In config.txt: “system_id: training”

Figure 31. File name of an Iridium_file.

18.2.2 The file contents

The following line shows an example of the contents of an Iridium_file.

*A;210416093848;T;10;H;14;V;13.1;a;1.8;b;0.0;L;51.87522;l;4.619341

- A is the system_id you defined in the ‘Iridium:’ block.
- 210416093848 is the date & time stamp yymmddhhmmss (here 2021-04-16 09:38:48)
- T;10 is the first <tag;value> pair. The ‘tag’ is the ‘Iridium TAG’ you defined in the reference table, see 12.5.

A tool interpreting this file should read the <tag;value> pairs and not assume that all tags are always present, nor that they always appear in the same order. Only if ‘add_timeout_field: True’ and special attention has been given to getting the parameters (tags) in a fixed order, you can identify the parameters by their position. In this case, a missing parameter (due to a response_timeout) will still result in a <tag;value> pair, but the ‘value’ will be empty (like T;;H;14 for a missing value for tag ‘T’).

19 Appendix 2: Tested sensors

Below is a list of serial (digital) sensors that Observator has used and tested with the OMC-048. Some have dedicated drivers, others use generic drivers. Many more sensors have been used by our customers and more will be tested over time.

Many analog sensors have also been used (tested), but are not listed, because interfacing analog sensors is usually trivial.

If you are in doubt if and how your sensor can be supported, please contact us.

Note that the generic drivers will support most sensors with these standard interfaces:

- SDI-12;
- Streaming ASCII data on RS232/422/485;
- NMEA on RS232/422;
- Analog 4-20 mA, 0-5V and 0-24V.
- ModBus

Serial (digital) sensors that have been tested with the OMC-048 by Observator at this moment. Many more have been used by customers.

Sensor/brand	Driver, table	Comments
EXO	• EXO: • generic_sdi12:	Multiparameter water quality sondes from Xylem/YSI
Manta	• manta:	Multiparameter water quality sondes from Eureka
NEP-5000	• NEP5000: • generic_sdi12:	Turbidity sensor from Observator
Navilock NL-8022MP	• navilock_md6: (alias gps:)	Multi-GNSS receiver ('GPS')
GMX501	• Gmx501:	Compact Weather station from GILL instruments, including GPS support and time sync.
Maximet (GMX)	• generic_serial_input:	Compact Weather stations from GILL instruments
AWAC, Aquadopp	• nortek_cp:	Current profilers from Nortek. Binary format for current profile only (not waves)
SVS603	• generic_nmea: (table PSVSW)	Wave sensor from Seaview
YSI 6-series	• ysi_6_series:	Multiparameter water quality sondes from Xylem/YSI (obsolete)
PT12,PT2X,CT2X, Smart sensors	• generic_sdi12: • modbus_seametrics:	Smart Sensors (pressure, temperature, conductivity and more) from Seametrics
Q-series PAR	• generic_serial_input:	Digital PAR sensors from Biospherical
Orinoco Solo	• generic_nmea: (table PTID)	Tide gauge from Seabed
Ponsel/Aqualabo	• generic_sdi12:	Several CT, pH and DO sensors from Aqualabo

TriOS NICO, OPUS	• generic_serial_input: (NICO) • modbus_opus: (OPUS)	UV spectral absorption sensors for Nitrate and more from TriOS
NKE MoSens	• modbus:	CT sensor with UV-LED from NKE Instrumentation
NKE WiMO	• modbus_wimo	Multiparameter sondes from NKE Instrumentation
spectro::lyser	• modbus_spectrolyser	UV spectral absorption sensors for Nitrate and more. From s::can (Badger Meter)

Note:
Some sensor that use a generic driver may already have a reference table ready for use. Others may need customization, depending on your sensor settings.

20 Appendix 3: Example config files

Sometimes it is easier just to look at a few examples to see how things work, than to read all the theory! So in this chapter I will give a few examples to get you started. It is best to read them in sequence, because they go from simple to more complex. If you want to make your own config file, it is best to start with an empty file and copy the relevant text blocks from the ‘config_template’ document.

But note that these examples cover just a tiny little bit of what is possible. If you want more, you will have to beat your way through the theory first!

20.1 Just to switch USB mode

This must be the smallest possible config file! It doesn’t do much, but you can use it to switch the usb_mode. Otherwise it is pretty useless.

```
# ----System---- #
Omc048:
  usb_mode: storage #debug/storage/repl
  system_id: training
  application: whatever
```

20.2 Logging (without transmission) of simple sensors

Below config file instructs the logger to take samples from a few sensors and to store them in a file. Data is not transmitted.

```
# ----System---- #
Omc048:
  usb_mode: repl #debug/storage/repl
  system_id: training
  application: whatever
  file_log_level: warn
  repl_log_level: info
  utc_time_offset_hours: +1
  utc_time_offset_minutes: +0
  sensor_data_print: True
  self_test: True

# ----Data-log-settings---- #
Data_file:
- id: data
  create_interval: "10 * * * *"

# ----Sensor-Settings---- #
Onboard:
- id: board_sensors0
  sample_interval: "0,20,40 * * * *"

Analog_voltage:
- id: AV
  port: 1
  sample_interval: "0,20,40 * * * *"
  min_in: 0
  max_in: 24000
  min_out: 0
  max_out: 24
  log_name: Vps
  log_unit: V
  log_tag: Vs

Analog_current:
- id: AC1
  port: 1
  sample_interval: "0,20,40 * * * *"
  min_in: 0
  max_in: 24
  min_out: 0
  max_out: 24
  log_name: Cp
  log_unit: mA
  log_tag: C1
  supply_port: 3
  supply_port_always_on: False
  startup_time: 5
- id: AC2
  port: 2
  sample_interval: "0,20,40 * * * *"
  min_in: 4
  max_in: 20
  min_out: 0
  max_out: 100
  log_name: Cp
  log_unit: "%"
  log_tag: C2
```

The used sensors are:

1. The 'Onboard' sensors (temperature, humidity and coin cell voltage)
2. The 'Analog_voltage' input (0 to 24000 mV, scaled to 0-24V)
3. Two 'Analog-current ' inputs 0 to 24 mA (AC1 is scaled 1:1 to 0 to 24 mA; for AC2, 4-20 mA is scaled to 0 to 100%.

Note that all sensors have an identical sample_interval, but AC1 has a startup_time of 5 s, so will be sampled 5 s later. It also uses a power port, that will be switched on at the sample moment. It will be switched off immediately after the sample has been taken.

Also note that the "`log_unit: "%"`" has the percent sign in-between quotes. The logger itself does not care about the quotes, but the remote control function of Blue2Cast does require these quotes. So better put them in.

The sensors are sampled every 20 seconds and are stored in a file that is created every minute.

Because '`sensor_data_print: True`', you will see REPL output like this (just showing the sensor data and leaving everything else out):

```
  Sensor data: ('Cp', '%', 'C2') -25.0
  Sensor data: ('Int. Temperature', 'C', 'ITEMP') 25.86617
  Sensor data: ('Int. Humidity', 'rh', 'IHUM') 40.88263
  Sensor data: ('Int. Coin cell voltage', 'V', 'IVBAT') 2.981685
  Sensor data: ('Cp', 'mA', 'C1') 0.0
```

A data file will look like this: (Sorry about the small print, but I don’t want to break the lines)

```
   T;25;26;27;28;29;30;
   Timestamp;Vps;Cp;Cp;Int. Temperature;Int. Humidity;Int. Coin cell voltage;
   yy/mm/dd hh:mm:ss;V;%;mA;C;rh;V;
   Time;Vs;C2;C1;ITEMP;IHUM;IVBAT;
   D;22/05/23 13:55:20;25;0.6479201;0;26;-25.0;0;28;25.88761;0;29;40.86737;0;30;2.975238;0;
   D;22/05/23 13:55:25;27;0.0;0;
   D;22/05/23 13:55:40;25;0.6479201;0;26;-25.0;0;28;25.86617;0;29;40.875;0;30;2.978461;0;
   D;22/05/23 13:55:45;27;0.0;0;
   D;22/05/23 13:56:00;25;0.6479201;0;26;-25.0;0;28;25.86617;0;29;40.88263;0;30;2.981685;0;
   D;22/05/23 13:56:05;27;0.0;0;
```

Note that sensor AC1, with tag C1, is sampled 5 s later than the other sensors, because of its startup_time.

20.3 Logging and transmitting with onboard sensors

Below example samples the onboard sensors on second 0, 20 and 40 of every minute. It saves them to a file that is created every 5 minutes (on second 10) and is also transmitted every 5 minutes. Every transmission, the relevant cellular diagnostic data is also sampled (`cellular_diagnostics: True`)

Data transmission is done by FTP. (The alternative would be to send to the Blue2Cast server by setting ‘`data_protocol: SSL`’). Remote control is enabled (‘`webserver_access: True`’), but for this you need to have a password from Observator. Else this should be set to `False`.

```
# ----System---- #
Omc048:
  usb_mode: repl #debug/storage/repl
  system_id: training
  application: whatever
  file_log_level: warn
  repl_log_level: info
  utc_time_offset_hours: +1
  utc_time_offset_minutes: +0
  sensor_data_print: True
  self_test: True

# ----Modem-Settings---- #
Modem:
  id: onboard_modem
  port: modem
  sim_username: KPN
  sim_password: gprs
  apn: internet
  cellular_diagnostics: True

# ---- External connection settings ---- #
Connection_settings:
  data_protocol: FTP
  ftp_url: ftp.omc-data-online.com
  ftp_port: 21
  ftp_username: OMC-test
  ftp_password: omc-test
  transmit_interval: "10 0,5,10,15,20,25,30,35,40,45,50,55 * * *"
  utc_time_sync: True
  utc_time_server: europe.pool.ntp.org
  utc_time_server_port: 123
  webserver_access: True
  webserver_url: blue2cast.com
  webserver_port: 5000
  webserver_password: <ask Observator>

# ----Data-log-settings---- #
Data_file:
- id: data
  create_interval: "10 0,5,10,15,20,25,30,35,40,45,50,55 * * *"

# ----Sensor-Settings---- #
Onboard:
- id: board_sensors
  sample_interval: "0,20,40 * * * *"

# ----Log only selected parameters
Log_parameters:
  onboard_modem:
  - Signal strength
  board_sensors:
  - Int. Humidity
```

Note that the 'Log_parameters' section causes only two parameters to be logged. The REPL output now looks like this (just showing the sensor data and leaving everything else out):

```
  Sensor data: ('Signal strength', 'dBm', 'SSTR') -59
  Sensor data: ('Int. Humidity', 'rh', 'IHUM') 42.19489
  Sensor data: ('Int. Humidity', 'rh', 'IHUM') 42.08045
  <similar lines left out>
  Sensor data: ('Signal strength', 'dBm', 'SSTR') -58
  Sensor data: ('Int. Humidity', 'rh', 'IHUM') 42.19489
  Sensor data: ('Int. Humidity', 'rh', 'IHUM') 42.08045
```

The signal strength is measured at each transmission and is stored in the file that will be send in the next transmission.

And this is the resulting data file:

```
  T;25;26;
  Timestamp;Int. Humidity;Signal strength;
  yy/mm/dd hh:mm:ss;rh;dBm;
  Time;IHUM;SSTR;
  D;22/05/23 15:15:20;25;42.2254;0;
  D;22/05/23 15:15:25;26;-61;0;
  D;22/05/23 15:15:40;25;42.20252;0;
  D;22/05/23 15:16:00;25;42.02704;0;
```

```
<Similar lines left out>
D;22/05/23 15:19:40;25;42.172;0;
D;22/05/23 15:20:00;25;42.20252;0;
```

If you are reading this line, you are very persistent! My compliments 😊