# Implementing the Backseat Driver Architecture

## on a G3s Slocum Glider: A Manual for Operators

Sophie Scopazzi

Version Date: April 2024

# Contents

# Preface

This manual is a starting point for anyone who wants to implement the Slocum backseat driver architecture on a G3S Slocum glider. It also may be regarded as everything required to replicate, follow along, and/or continue with my thesis work as this manual is complementary to my Master's thesis *Sensor and Behavior Frameworks for Implementing Backseat Driver on a Slocum Glider*.

All the files used throughout this document are available on Sophie Scopazzi's Github:
**https://github.com/sscopazzi/Slocum_Backseat_Driver_Manual**

---

## 0.1. Intended Audience

To the best of my ability, I've tried to approach the writing of this manual assuming no prior knowledge; however, prior knowledge of Slocum glider operation and piloting, along with experience with Python functional programming would be extremely helpful. If that sounds scary, please don't let that dissuade you from working through these topics. They can be difficult to work through, especially at first, but it'll be okay!

## 0.2. How to Use This Manual

This manual has been written so one can work sequentially through this document and the concepts within as they progressively get more advanced. If you feel you already know something, please feel free to use the Contents page to skip to the area you want to know more about!

- **Chapters 1-2:** Background about what the backseat driver architecture is, what an external controller is, how it is used and the type used in this manual, then how to implement the external controller onboard a glider or a glider simulator.
- **Chapters 3-6:** Learning glider piloting and simulation, then performing a simple backseat driver mission.
- **Chapters 7-9:** A more advanced backseat driver application, then reading and manipulating science data onboard the external controller. Sending a file processed on the external controller to the glider, for ability to retrieve via SFMC mid-mission.

Lastly, I highly recommend reading through Teledyne Marine's G3S Backseat Driver Software Guide prior to using the examples in this manual. It will provide more of the software theory behind the examples gone over in this manual. At the time of writing I:

- Referenced version M315400-NFC, Rev A of the software guide
- Used glider software version 11.00 SW311356-NFC-RA

## 0.3. Acronyms Used

**Table 1**

| Acronym | Explanation | Description |
|---|---|---|
| BSDA | Backseat Driver Architecture | The overall system concept of adding an external controller (EC) into the glider and the hardware/software required to do so |
| BSD | Backseat Driver | Typically conflated as either the EC or BSDA, thus not used in this manual |
| FC | F-CPU, Flight Computer | Handles all glider hardware control, navigation, and related subsystems |
| SC | S-CPU, Science Computer | Handles all scientific measurements |
| EC | External Controller | Additional computer inside the glider connected to the SC |
| GoM | Gulf of Mexico | South of the United States |
| LC | Loop Current | |
| LCE | Loop Current Eddy | |
| LMC | Local Mission Coordinates | Coordinate system glider's use for navigation based upon mission start location |
| RP | Raspberry Pi 4 Version B | A ubiquitous small single-board computer |
| SFMC | Slocum Fleet Mission Control | Online webapp for controlling and piloting gliders |
| YC | Yucatan Current | East of the Yucatan Peninsula |

# List of Figures

# List of Files

# 1

# What is Backseat Driver...

# ...and how it relates to G3S Slocum Gliders

**In analogous terms**, a Backseat Driver (BSD) is like when you and your partner drive to visit a friend in a new-to-you town. It's your car so you drive with your partner in the passenger seat, and the local friend in the back seat. The local friend is scootched up and leaning forward through the front seats pointing across your face telling you to turn left so you don't miss the exit. Your friend, in the backseat, is seeing what's outside the car and making decisions that you, in the driver's seat, blindly follow. Maybe they even disagree with what the maps on your phone is saying because they know something better based on prior knowledge or what they see outside.

In this analogy, the driver is the flight computer, the passenger is the science computer, and the local friend is the BSD. The flight computer and science computer are already part of the glider system (you and your partner's car), hence the term "external controller" for the external controller (friend visiting your car). Conceptually, the external controller is given the name "external" because it resides outside the glider's normal system (Figure 1.1, red box to right). It's important to note that it is not outside or external to the glider housing, just the glider's standard control loop.

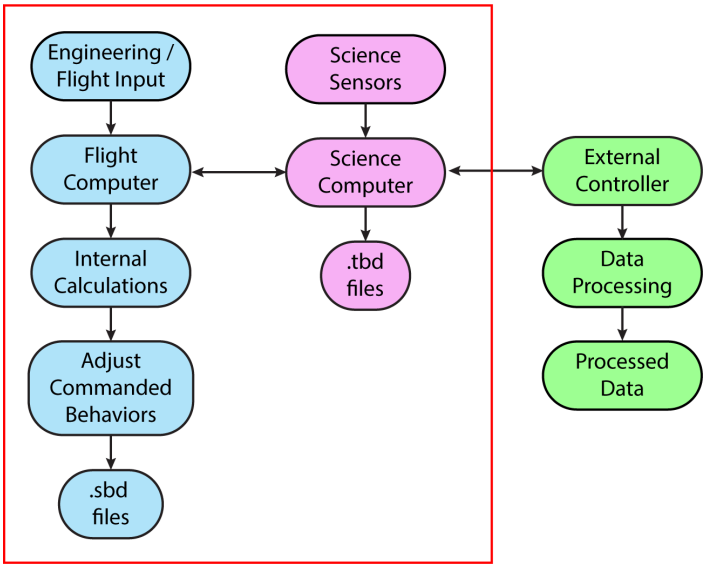


**Figure 1.1: Backseat Driver Conceptual Layout (basic):** Red box contains standard commercial-off-the-shelf glider configuration, with the addition of the external controller in green

An important nomenclature note that can be confusing while researching this topic is that the phrase external controller and BSD *can* often be equated to mean the same thing, so long as we are not talking about backseat driver as a concept. For clarity, in this manual I will:

- Use external controller when the computer could be conceptually generic
- Use RP when talking about specific use-case examples in this manual, using the Raspberry Pi
- Use BSD to refer to the backseat driver concept

## 1.1. What is an External Controller?

The external controller could technically be almost any kind of computer capable of receiving and sending serial data, but it also must exist within the size and energy consumption constraints that come from being housed internal to the Slocum glider during long-duration missions; i.e., it essentially must be a small form factor and energy efficient single-board computer. This has led to adoption of external controllers such as a BeagleBone Black or a Raspberry Pi 4, with other exciting newer possibilities like the NVIDIA *Jetson Nano*. However implemented, an external controller onboard the Slocum glider allows for greater decision making, control fidelity, and onboard data processing than possible without.

At a practical level The external controller runs a python script, so think of it more as user directed autonomous control. It does think for itself, but within constraints placed upon it by us, the creators of the Python script. Thus, what the external controller does is:

1. Receive glider values and/or a sensor's data file specified by the user
2. Processes and/or manipulates the data
3. Sends commands to the glider to alter its behavior and/or process a sensor's datafile into a smaller file and transfer it back to the science computer for sending ashore (process an onboard AD2CP file into a smaller file)

## 1.2. How the External Controller and Science Computer Communicate

The external controller and science computer communicate by configuring the external controller as a sensor, which is connected to a science computer's open port. The physical connection is made using a custom cable detailed in a later chapter. The external controller and science computer use the standard RS-232 communication protocol, allowing for two way communication. Character combinations specify the type of message and delimiters break up the values therein. There are two typical ways the Slocum backseat driver architecture is organized:

**Table 1.1:** Two Typical Approaches to Using the EC

| Always Powered Example: Adaptive Control | Intermittent Power Example: File Processing |
|---|---|
| 1. The external controller subscribes to sensor values that are already onboard the glider. The values can be from either the flight computer (heading, etc.) or science computer (temperature, salinity, or other variable from science sensors)<br>2. Every glider cycle, the science computer sends each updated subscribed sensor value to the external controller.<br>3. The EC's Python script takes the incoming string and breaks it apart using delimiters built into the message (first (*), then (,), then (:)) to get the value from the science computer into a usable format on the EC<br>4. Using the same Python script, the external controller manipulates or otherwise uses the incoming values to make decisions not possible onboard the flight computer or science computer in real-time<br>5. The external controller then updates flight sensor values (b_args) on the flight computer via communicating back to the science computer. | 1. The science computer powers the external controller on as defined in the typical glider sample behavior (15 choices). Once powered on, the Python script runs on external controller startup, requesting the most recent file from the glider. This can be any file on the science computer, but usually from a sensor whose file is too large to send ashore without onboard processing, like an AD2CP<br>2. The file is transferred to the EC, then processed, usually into a smaller file size.<br>3. The external controller sends the smaller file back to the science computer for sending ashore via Iridium |

All the different message formats are descried in the following tables. Some are ”background” messages not directly controlled by the user, while others must be specified in the python script ran onboard the external controller.

**Table 1.2:** What the science computer expects from the external controller (output messages)

| Message | Format | Description |
|---|---|---|
| **$SW,num:value,num:value,...** | **num:** Integer value corresponding to the index (0–63) in the extctl.ini. **value:** Float point number of the value to which the mission param or sensor shall be set | In a SW message, multiple sensors or mission parameters can be set in a single message. |
| **$MD,mask,value** | mask (decimal), value (decimal) | This type of message sets the mission mode. Only bits in mask are modified, which allows multiple independent groups of behaviors. Both mask and value are decimal numbers. You probably won't need to use this one. |
| **$FR,filename,directory** | **filename:** string, 8.3 format, the name of the file to be read. **directory:** string, 8.3 format, the directory in which to find the file. This argument is optional; if no directory argument is provided, the default directory is `config` | Command to specify a file on the science computer to read;, i.e., send a file from the science computer to the external controller. |
| **$FW,filename,directory** | **filename:** string, 8.3 format, the name of the file in which to write incoming $FO data (see next message explanation below. **directory:** string, 8.3 format, the directory in which to write the file. This argument is optional; if no directory argument is provided, the default directory is `logs` | Command to specify a file on the science computer to write;, i.e., send a file from the external controller to the science computer |
| **$FO,base64_data** | **base64_data:** string, data to write, encoded in base 64 format | Output data to file specified by $FW |
| **$FC** | N/A | Closes output file |
| **$GO** | N/A | This message is expected by the EXTCTL proglet after it has sent file data in a $FI message. It is an acknowledgment the file data has been received. The EXTCTL does not send the next packet of data until it has received a $GO message |
| **$LS,directory,wildcard** | directory (string), wildcard (string) | Commands the EXTCTL proglet to start listing file names in the given directory matching the given wildcard. The glider begins pushing back DIR messages containing the names of the files found |
| **$TXT,message** | **message:** value | Must be a float, integer, or a decimal value (not a string). Commands the EXTCTL proglet to print the given message to the glider terminal |

**Table 1.3:** What the external controller expects from the science computer (input messages)

| Message | Format | Description |
|---|---|---|
| **$HI** | N/A | This message is sent by the EXTCTL proglet after initialization. Indicates proglet is ready to receive messages. Done automatically without need for user action |
| **$BY** | N/A | Sent by proglet before shutdown. Indicates the proglet is no longer ready to receive messages. Done automatically without need for user action |
| **$SD,num:value,...** | **num:** an integer value corresponding to the index (0–63) from extctl.ini, the ordering of when that sensor appears in extctl.ini determines the index (start counting at 0) **value:** Float point number representing the new sensor values | The requested sensors in the extctl.ini file's "is" section. The listed sensors will be included in an $SD message only when **that sensor is updated**. Multiple sensors are included in the same message if they are updated at the same time, and are separated by the field delimiter (,) |
| **$FI,base64_data** | **base64_data:** A string containing data to write, encoded in base 64 format | These messages are file input data. After receipt of an $FR message, the proglet will output $FI messages until the end of the input file is reached. At the end, and additional zero-length $FI message will be sent automatically |
| **$ER,code,directory** | code (integer), directory (string)<br>1. File error<br>2. Incorrect number of fields in message<br>3. Sequence error, file write before file open<br>4. Could not decode data field<br>5. Unknown message type | An error message that indicates the proglet was unable to process the most recently received message.<br>1. Bad value or could not parse number<br>2. Bad index, index is greater than the max index in extctl.ini<br>3. $FR or $FW directory argument is not an existing directory (bin, state, config, logs)<br>4. Error occurred when lowering UART power bit of the EXTCTL proglet |
| **$NEXT** | N/A | This message is sent from the EXTCTL proglet after it has finished processing a $FO message from the Backseat Driver and is ready for the next $FO message to be sent. Useful as a trigger for the external controller script so that data can be transferred at a controlled rate |
| **$DIR,file1,file2,file3,...,file10** | List of filenames (strings) | Sent after receiving a $LS message from the external controller. It contains up to 10 file names that match the parameters sent in the $LS message. The proglet continues to send $DIR messages until all files found matching the parameters in the $LS message have been sent |

## 1.3. Using External Controller for Adaptive Control (u_mission_params)

**Section under construction.**

# 2

# External Controller Implementation

**Chapter Overview and Concepts:**

1. Download the RasPi operating system and flash it to a microSD card.
2. Set up SSH access on the RasPi for remote access.
3. Remotely connect to the RasPi via SSH using WinSCP.
4. Automate the operation of our script using the RP's crontab.
5. Review handy troubleshooting commands that can be handy to have.
6. Make the physical connection between the RasPi and the glider's SC.

## 2.1. Raspberry Pi Specifications

In this manual's case, the external controller is a Raspberry Pi. A RasPi is a low-power single-board computer that's ubiquitous in classes that teach basic hands-on computer science.

## 2.2. Setup the Raspberry Pi

Before implementing the BSD capabilities, we will first download the operating system for the RP. The steps to do so are as follows:

1. Download Raspberry Pi Imager V 1.7.5 for your operating system.
   (a) Insert the SD card into your laptop or card reader.
   (b) Click "Choose OS" – choose the Raspberry Pi OS (64-bit) option. If it isn't listed on the main page, select Raspberry Pi OS (other).
   (c) Click "Choose Storage" – select the appropriate storage for your project.
   (d) Click "Write" – this will install the options you have selected above.
2. Insert SD card into slot in the Raspberry Pi and power on
3. Set up a username (rutgers) and password (rutgers)
4. Create a folder for your project in the '/home/rutgers/' directory. Something like a folder called `backseat_driver`. Make sure the folder has no spaces, hence the _.
5. In this folder, you will have your python scripts for your project and a shell script that executes those scripts, along with anything else needed for the mission by the RasPi (we will do this near the end).

## 2.3. SSH access

Access to the RasPi via SSH, secure shell, is required for ability to remotely send and receive files while the RasPi is inside a glider. SSH is a communication "protocol...for securely sending commands to a computer over an unsecured network." (*CloudFlare: What is SSH*). If your university has a VPN to connect to the universities network, the additional benefit of connecting the RasPi to an internet connected network (a universities, or departments) is the ability to remotely SSH in from home while running tests.

1. Enable SSH access by either, 1) From the RP's desktop GUI: Preferences -> Raspberry Pi Configuration -> Interfaces -> Enable SSH, or 2) From the RP's terminal: `sudo raspi-config -> interface options -> SSH`
2. Once SSH has been enabled, reboot the RasPi from the desktop GUI or from a terminal with the command: `sudo reboot`
3. Connect the RasPi to the same network as your computer (wireless network or Ethernet). Connecting to the same wifi network on the RasPi and your computer is probably the easiest option.
4. Get the RP's ip address it was assigned on the network in the RasPi terminal with command: `hostname -I`
5. Verify connectivity between your computer and the RasPi via SSH. From a command prompt or terminal window: `ssh rutgers@[RP's ip address].` It probably will ask if you trust the source, so type `yes` then the password you setup earlier. When typing passwords into a terminal window it will not look like anything is happening (it doesn't show ***** characters). This is normal behavior. Trust your keypresses then hit enter. If unsure halfway through, press delete a lot or hit enter so it asks you to retype.

```
hostname -I                     # from the terminal shows RasPi ip address
sudo reboot                     # from the terminal, reboots the RasPi
ssh rutgers@[RP's ip address]   # from windows/mac ssh into the RP
```

Now that the RasPi can be accessed via SSH, you can send and receive files over the shared network. This can be done via the command line from the Windows or Mac computer; however, it's far easier to use a program like WinSCP to send files.

### 2.3.1. Connecting to the Raspberry Pi via WinSCP

**Section under construction.**
(WinSCP Website). A Mac program one can use is Cyberduck.

## 2.4. Automating the Script Using Crontab

To automate the script on the RP, we require a two files:

1. **launcher.sh:** A launcher file that runs the python script (.py).
2. **python_script.py:** The script we will run is written in python and saved as a .py file. The program the RasPi runs makes decisions, runs calculations, etc. This script will be in our /backseat_driver folder on the RP, and is provided later in this manual for each chapter.

### 2.4.1. Making the .py Launcher

The launcher is required to avoid errors and to ensure the log outputs properly. This is the entirety of the launcher.sh file in the project's home directory `home/rutgers/backseat_driver`.

```
# launcher.sh file on the RP
cd /                                # make sure we are in the home directory
cd /home/rutgers/backseat_driver    # move to directory where the .py script is
sudo python3 [script name].py       # run the script
cd /                                # go back to home directory
```

**File 2.1: Backseat Driver Implementation:** launcher.sh

### 2.4.2. Setting up the Crontab and Automating the .py Script

1. Send command `sudo crontab -e` to open the crontab on the RP. This opens a file on the RP, *but only edit via this command*
2. Enter the lines below to run the script in the x time period you want.

```
# to open crontab thru the RasPi terminal (only open via terminal, not the file on the RP)
sudo crontab -e

# at end of file input these commands with the path to launcher
* * * * * sleep 5: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1
* * * * * sleep 10: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1
* * * * * sleep 15: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1
* * * * * sleep 20: sh /home/rutgers/backseat_driver/launcher.sh >>
                        ... etc ...
* * * * * sleep 55: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1
* * * * * sleep 60: sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1

# to run once every minute
* * * * * sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1

# to display updates written to cronlog while crontab is running
# tail -f cronlog
```

**File 2.2: Backseat Driver Implementation:** Setup the the Crontab

The crontab is the RP's way of running a script on reboot or a schedule. The shortest cronjob time period is every minute, but we can get around this limitation by using the sleep command, and running a cronjob every x seconds. The end result is a script that runs every x seconds. The end of each line above (2-11) is how the output of the script is saved in the cronlog folder (important for troubleshooting and viewing script running).

### 2.4.3. Viewing the output of the Raspberry Pi .py in the Cronlog

Now that we specified a location to log the output of the script, we can view the log as the RasPi runs the script. To display a static version of the entire cronlog first navigate to the cronlog file location descriped above using `cd [directory name]`. Once there, use `nano cronlog` or `vim cronlog`; however, to view the end of the file as new running scripts update the file, use `tail -f cronlog`. This prints the results of the script as it's run, as it's showing the updates to the tail (the end) of the cronlog. To get out of the `tail -f` command, use `control-c`.

## 2.5. Troubleshooting Python Scripts on the Raspberry Pi

```
# list all running python scripts
ps -aef | grep python

# if need to kill ALL scripts
sudo killall python3

# if need to kill specific scripts
sudo killall -e python3 sendHeadDir.py

To run on boot use this command in the crontab (for reference)
@reboot python /home/rutgers/bsd/script.py
```

If the RaspPi script runs fast you will probably not see it using the grep command, but if the .py is run in a program like Thonny it *should* show up as Thonny 'suspends' instead of fully ending the script when ran. This also depends on the script, as an 'if' statement will end, but a 'while' statement could hang and wait for values (we'll get into this later).

## 2.6. Connecting the External Controller to the Glider

The external controller needs to communicate bidirectionally with the glider to read variables and/or files from the glider and to provide resultant control and/or files back to the glider. This necessitates a physical connection from the RasPi to the glider's science computer. This is done by using a custom cable and utilizing a UART communication method.

### 2.6.1. Making the Connector

The photo below (Figure 2.1) shows the parts as ran for simulations: top left is the RasPi; top right is the RS-232 and DC power adapter; bottom center are the shoebox simulator's science port. The parts used:

1. **10-pin Male Molex:** Words
2. **Weird small connector thingies and their tool:** Words
3. **DC Adapter:** Words
4. **RS-232 Adapter:** Words
5. **placeholder because I forget:** Words

**Figure 2.1: Science Computer and External Controller Connector:** Image of Parts During Simulation

Figure 2.3 (next page) shows each component part for the external controller with corresponding label. Figure 2.4 (next next page) is the wiring diagram for making the external controller's connector.

Additionally, if you will be simulating a science sensor I'd recommend making the connector for that now (do it all at once). The photo below (Figure 2.2) shows the simCTD USB to RS232 to Molex connector.

**How to Wiring Resources:**

1. SparkFun resource for *how to read a wiring schematic*
2. KiCad Schematic Editor help documentation

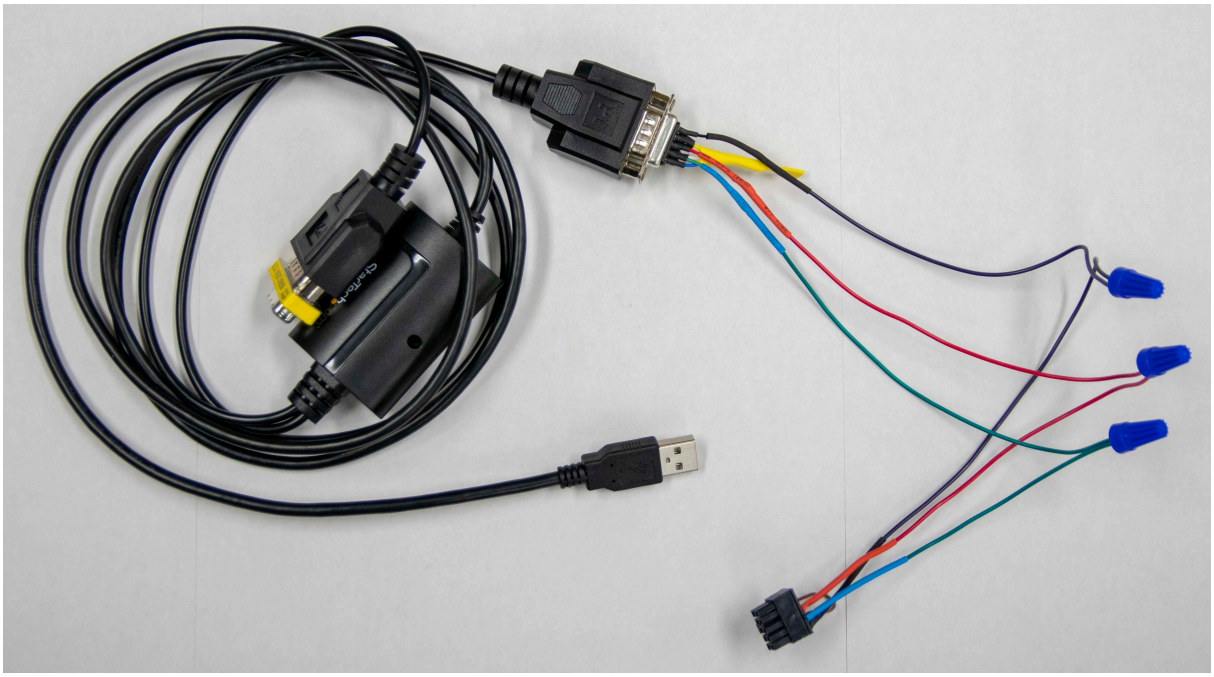

**Figure 2.2: Science Computer and Laptop Connector:** simCTD RS-232 cable. Blue wire nuts for increased modularity

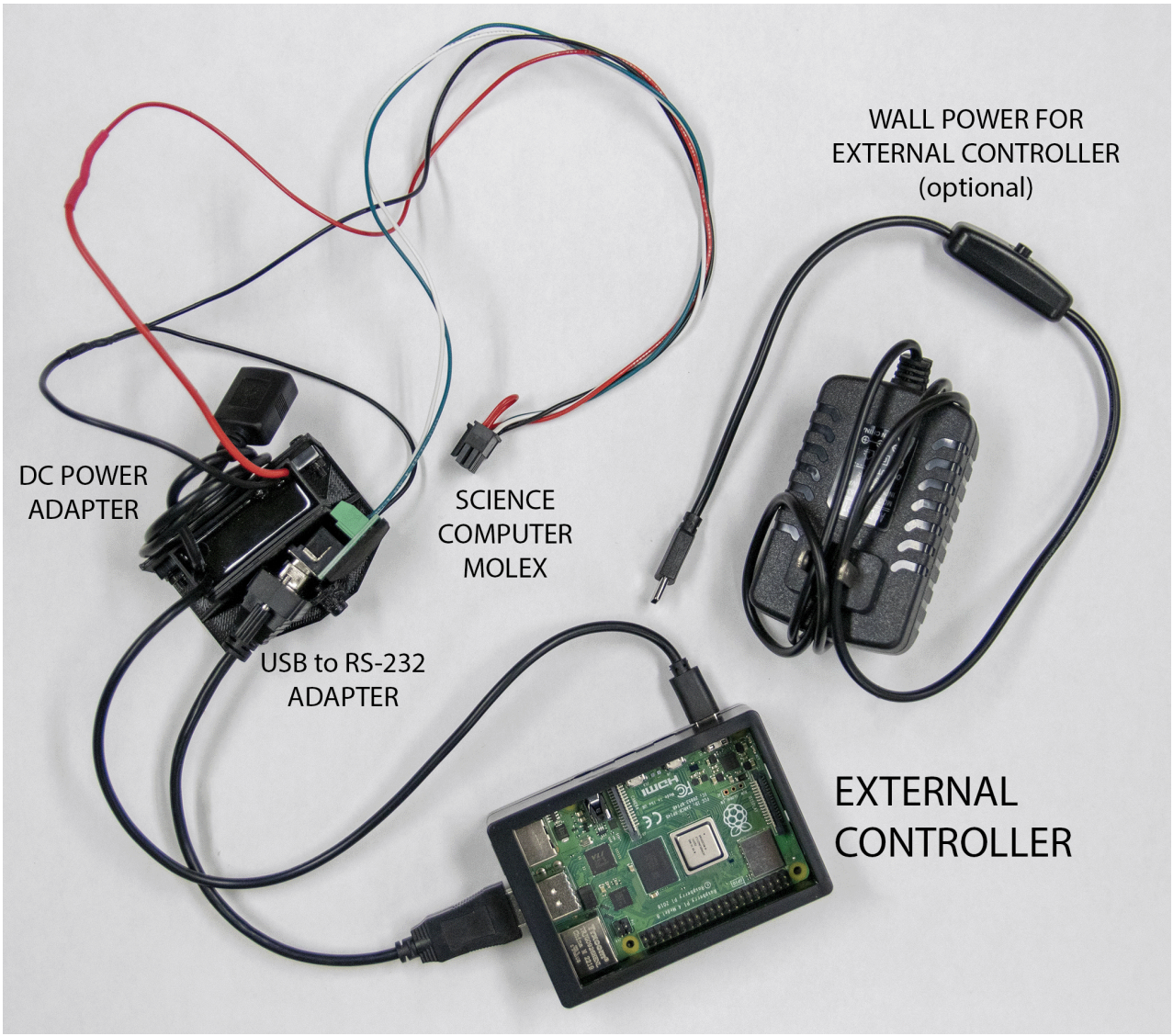


**Figure 2.3: Science Computer and External Controller Connector:** Individual parts labeled with words

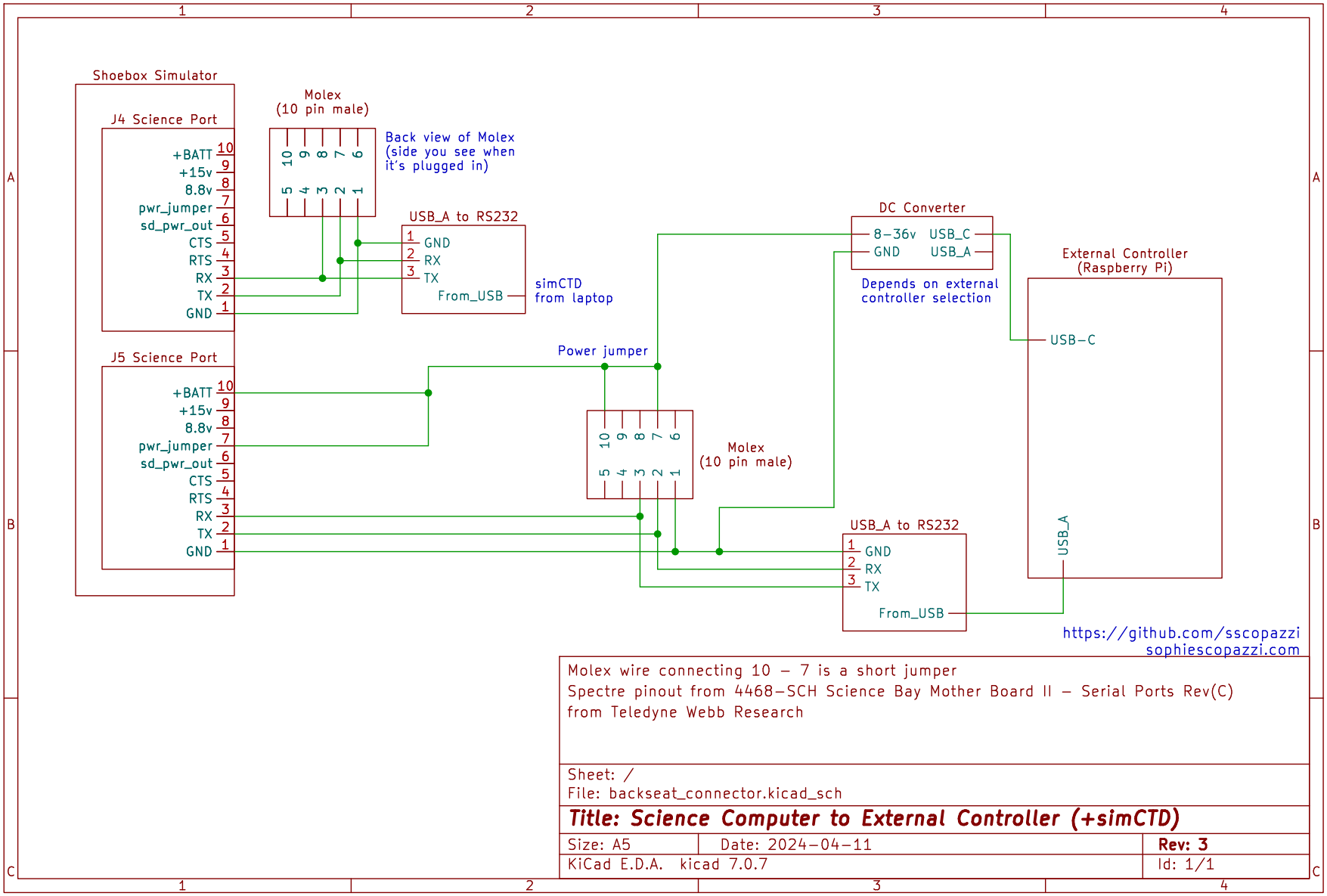


**Figure 2.4: Science Computer and External Controller Connector:** Wire Diagram

## 2.7. Which Files Go Where: Glider Pathways

The list of files we will reference or use for each chapter is large, and where they go on the glider is vitally important. In most cases, the file extension explains where the file goes; `.mi` for the `/missions/` folder and `.ma` for the `/mafiles/` on the FC. Where it can get confusing is with `.dat`, `.ini`, etc. When dropping files into the "To Glider" folder via SFMC and sending them to the glider using `dockzr *.mi *.ma` etc, the glider puts the files in the correct folder. Highlighted files are required for using the BSDA.

**Table 2.1:** Reference Documents

| Filename | What it's for |
|---|---|
| masterdata.txt | words |

**Table 2.2:** Flight Computer Files

| Filename | What it does | Where it goes |
|---|---|---|
| autoexec.mi | Glider specific file containing everything that makes the glider that specific glider (calibrations for onboard equipment, etc.) | `/config/` |
| sbdlist.dat | Specify which FC values to save and how to save them (decimation) | `/config/` |
| simul.sim | Tells the glider that we are simulating, and how (electronics only, e tc.) | `/config/` |
| bs_cft.mi | The mission file that sets b_args (values) and calls the `.ma` files | `/missions/` |
| loadsim.mi | Ran before starting the mission, this initializes the simulated variables | `/missions/` |
| sample90.ma | Controls the sample behavior of the extctl proglet (the external controller) | `/mafiles/` |
| yo99.ma | Controls the yoing of the glider - dives and climbs | `/mafiles/` |
| surfac10.ma | Controls surface timeouts and other parameters | `/mafiles/` |
| surfac40.ma | Each number relates to a different type of surfacing reason | `/mafiles/` |
| surfac42.ma | For example, this is mission end | `/mafiles/` |

**Table 2.3:** Science Computer Files

| Filename | What it does | Where it goes |
|---|---|---|
| extctl.ini | Controls and specifies which SC values are sent to the BSD | `/config/` |
| proglets.dat | words | `/config/` |
| tbdlist.dat | Specify which science values to save in the .tbd files and how often to save them (decimation). See comments in file for more detail | `/config/` |

**Table 2.4:** Raspberry Pi Files

| Filename | What it does | Where it goes |
|---|---|---|
| current_field_test.py | the BSD script ran on the RasPi | `/backseat_driver` |
| launcher.sh | Specified in the crontab, this runs the .py | `/backseat_driver` |
| crontab | Runs the .py on a schedule | Accessed with command `sudo crontab -e` |

### 2.7.1. extctl.ini

```
# EXTCTL.INI
# Maximum number of variables one can be subscribed to is 64 by default

# MISSION PARAMS SECTION - u_mission_param[a-l]
mp
                            # INDEX NUMBER FOR RasPiPYTHON
u_mission_param_a           # 0
# c_heading in 7099.ma

# OUTPUT SENSORS SECTION
# there needs to be one sci_generic defined here per mission param used
os
sci_generic_a nodim         # 1

is # INPUT SENSORS FROM GLIDER SECTION

# FLIGHT COMPUTER - example variables
m_water_vel_dir rad         # 2

# SCIENCE COMPUTER - example variables
sci_water_cond S/M          # 3
sci_water_temp degC         # 4
sci_water_pressure bar      # 5

baud        # Baudrate of UART
9600        # default value

packet_size # specify packet_size of files transferred
600         # default value
```

**File 2.3: Current Field Test:** extctl.ini - science computer's external processor settings file

Both the baud and packet_size can be higher for faster file transfers, but a baud over 15200 is not recommended for stability. The max packet size is 730. If baud and packet_size arguments are left out of the extctl.ini, the SC will use the default values. I leave them in all files used for clarity throughout this manual.

**IMPORTANT**: During each glider cycle, the message send to the EC contains only the **updated** values specified in the extctl.ini, not **every** value asked for on **every** cycle. To restate, **if the sensor value has not been updated on that specific cycle of the glider, the value will not be sent during that cycle's message**. Teledyne Marine knows this is a desirable toggle-able setting and it may be included in a future BSD release.

### 2.7.2. proglets.dat

```
proglet = extctl            # define name of proflet, corresponds with sample90.ma on the FC
    uart       = j5         # science port it's plugged in (could also be j4)
    bit        = 0          # power bit control, ability to turn the RasPion/off
    start_snsr = c_extctl_on(sec)
```

**File 2.4: File Explanation:** proglets.dat

### 2.7.3. sample90.ma

```
behavior_name=sample

<start:b_arg>
    b_arg: sensor_type(enum)                    90 # c_extctl_on
    b_arg: sample_time_after_state_change(s)    0  # start sampling right away
    # Sampling Arguments
        b_arg: intersample_time(sec)        0
        b_arg: state_to_sample(enum)        15
        b_arg: nth_yo_to_sample(nodim)      1
<end:b_arg>
```

**File 2.5: File Explanation:** sample90.ma

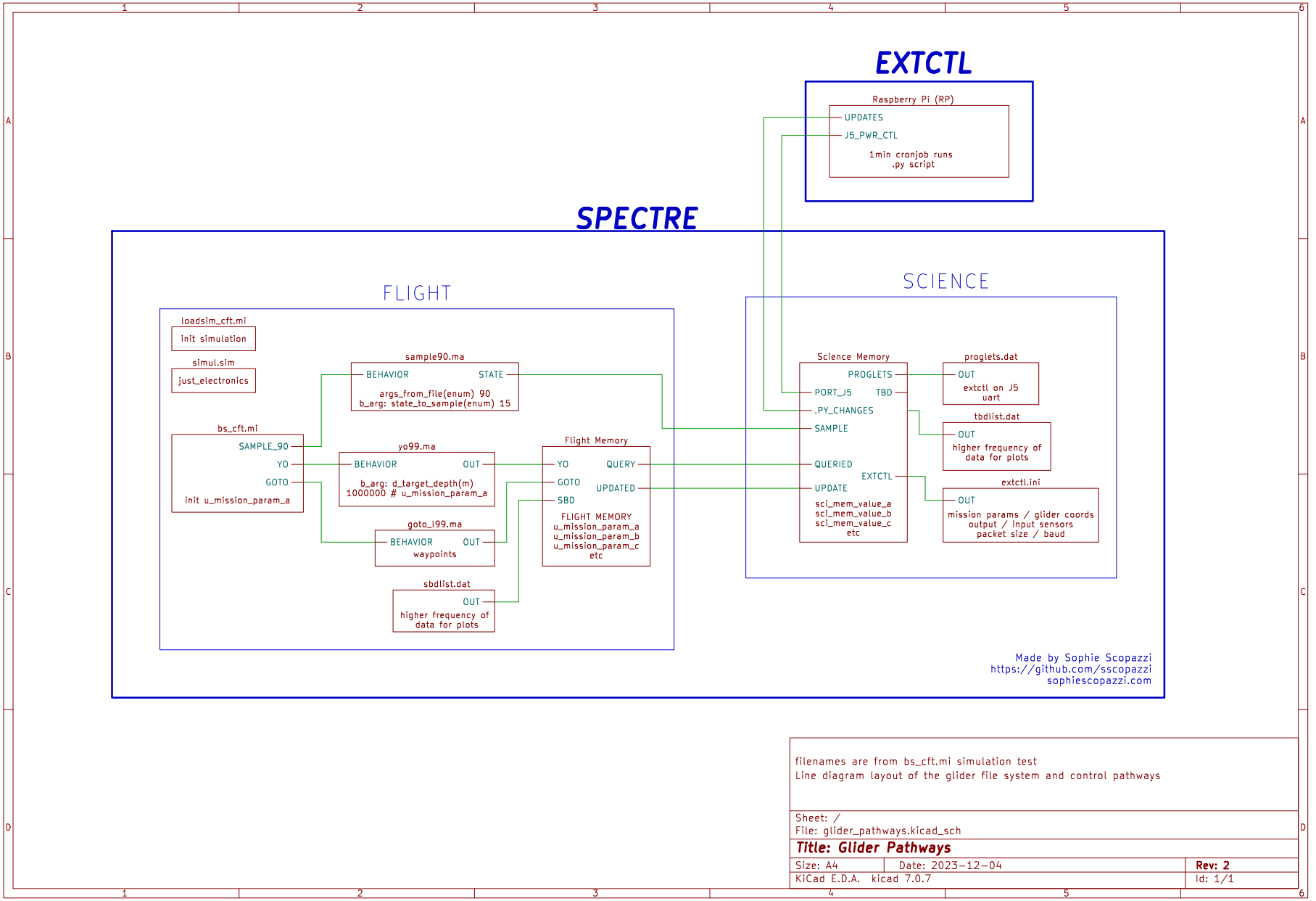


**Figure 2.5: Internal Glider File Pathways:** Which Files Go Where

# 3

# Running a Glider Mission in a Simulated Current Field

**Chapter concepts:**

1. Review every file required for piloting a glider in the simulator
2. Simulate a basic ocean current using the glider simulator's built-in fast speed zone
3. Control m_depth using the external controller

Now that we have a fully configured and ready external controller, our RasPi, and have created our wired connection to the glider science computer, we can get into using the glider simulator. This chapter familiarizes us with how to pilot a glider and all the files required and what they contain. We will also implement an extremely basic type of BSD control where we change the commanded depth of the glider after it dives. This chapter's information and files are the building blocks for all future chapters and glider missions.

## 3.1. Using Slocum Fleet Mission Control (SFMC)

SFMC is the web interface used to control gliders. Through SFMC we can send commands, send and receive files from the glider, and select SFMC .xml scripts (which we go over in future chapters) that control the glider automatically without any user input. There are many other features of SFMC, but we will only go over what is necessary for this manual here, starting with the typical glider terminal commands needed for this work.

**Table 3.1:** SFMC Terminal Commands

| Command | What it does |
|---|---|
| {enter key} | hows ready for commands |
| dir | list files in the current directory |
| cd [directory name] | Changes terminal to that directory |
| type [filename] | print filename to glider terminal |
| dockzr [file] | Send files from SFMC to the glider |
| zs [file] | Send files from the glider to SFMC |
| *.[file extension] | The * acts as a wildcard, representing every file with that extension |
| sample*.ma | The same idea as above, but represents every file starting with sample |
| ver | Displays glider name and software version<br>Ex: Vehicle Name: spectre, Glider: Version 11.00 SW311356-NFC-RA (0, 1) |
| consci | Transfer from the flight computer to the science computer |
| quit | Return from the science computer to the flight computer |
| control-C | Quits the current mission control-R |
| Resumes the mission (glider dives) exit reset | Restart the flight computer (will boot back up) |
| exit | Turns off glider (does not boot back up) |
| help | Shows a list of commands (far more than shown here) |

### 3.1.1. Sending Files to the Simulator

On the top right is a blue box called ”to-glider”. Files in this folder can be sent to the glider. Below the blue box is the ”To Glider File Drop Zone”, the place to drop files that will be sent to the glider. However, unlike a standard glider which utilizes the ”to-science” folder mid-mission, we will only be using the ”to-flight” folder and that drop-zone.

An example of steps to send two files, `bs_cft.mi` and `sample90.ma`, to the glider is below:

1. Create (or download from Github) both files. To follow the Rutgers Glider Team’s convention, make a duplicate of both files and add a `.cur` file extension. This is done to have an easily accessible record of what’s actually on the glider, because once the `.ma` file has been sent it disappears from the folder. Thus, drop four files: `bs_cft.mi`, `bs_cft.mi.cur`, `sample90.ma`, `sample90.ma.cur`.
2. In the home directory of the flight computer when not in a mission the command to send files is `dockzr *.ma *.mi`. This will send all files ending with both extensions to the glider. One file can be sent by specifying the file (`dockzr sample90.ma`). When in a mission, all commands are began with an exclamation mark, so the command would be `!dockzr *.ma *.mi`.


**Figure 3.1: SFMC:** Top right folders - to/from-glider, file drop zones, logs

### 3.1.2. Getting Files off the Simulator

To send files off the glider, you must be in the directory the file is within, then:

1. `zs [filename]`. This will send the specified file. Remember the use of wildcards (*), as they allow to batch send all `.sbd` files in one commmand, instead of one at a time.
2. Once sent, in the topmost right click the ”from-glider” folder and download the files sent.

### 3.1.3. Logs, Scripts, Viewing Input Commands

**Logs** are on the lower right of the blue buttons. They are text files of the glider terminal’s text and extremely useful for going back and checking what happened if you can’t scroll up and find it. Even days later they can be downloaded and searched through using control-F: very handy!

**Scripts** are selected below the terminals enter-command box. The name of the selected script is on the left while the script state is on the right. Script states are gone over more in Chapter 4 and 5.


**Figure 3.2: SFMC:** Selecting scripts and viewing their current state

Below the script states is where to view recently submitted commands. If scripts are running uncheck ”Show human user commands only”.

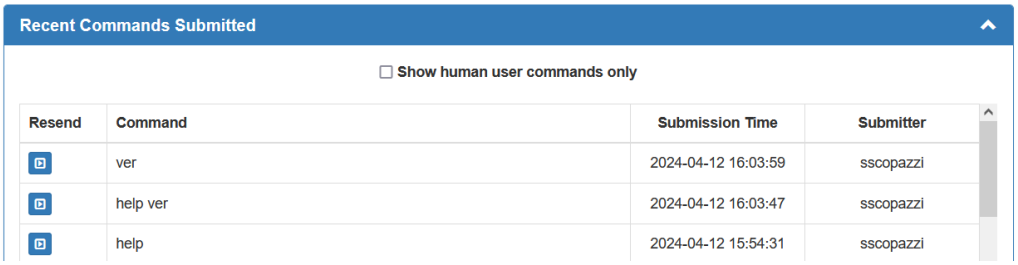


**Figure 3.3: SFMC:** Viewing recently submitted commands

## 3.2. Files Used in Test

All files for simulations in this manual are on GitHub. As the basic principle for the files are similar in all missions, I'll only explain the pertinent changed details in future chapters.

### 3.2.1. bs_cft.mi

```
### Onboard Processing Specific Parameters ###
sensor: u_allowable_cycle_overrun(msec) 3000 # how large x_cycle_overrun_in_ms can be
sensor: u_mission_param_a(nodim)        7    # z, initalize
sensor: u_reqd_depth_at_surface(m)      3    # in, depths less than this considered "at surface
    "
#   - yo_l99.ma        #
#   - goto_l99.ma      #
#   - loadsim_cft.mi   #
# GOTO_LIST : Waypoint list
behavior: goto_list
    b_arg: args_from_file(enum)     99  # mafiles/goto_l99.ma
    b_arg: start_when(enum)          0  # 0-immediately
# YO : Flight params
behavior: yo
    b_arg: args_from_file(enum)     99  # mafiles/yo99.ma
    b_arg: start_when(enum)          2  # 2-depth idle
    b_arg: end_action(enum)          2  # 2 resume
# SAMPLE : EXTCTL
behavior: sample
    b_arg: args_from_file(enum)     90 # 90 v11 (89 v10.8)
```

**File 3.1: Current Field Test:** bs_cft.mi - mission file

For each behavior the args_from_file number calls the relevant file; i.e., 99 in the yo behavior calls `yo99.ma` in `c:/mafiles` on the flight computer. For this test I differentiated the goto and yo files from the norm of ending in 10 by ending them in 99.

### 3.2.2. proglets.dat

```
proglet = extctl
    uart       = j5         # science port it's plugged in
    bit        = 0          # power, battery, ability to turn the RasPi on/off
    start_snsr = c_extctl_on(sec)
```

**File 3.2: Current Field Test:** proglets.dat - science computer's sensor file

Ensure the RasPi is plugged into the specified J port on the SC (it must be J4 or J5).

### 3.2.3. extctl.ini

```
                           # INDEX NUMBER FOR RASPI PYTHON
u_mission_param_a          # 0

os # OUTPUT SENSORS SECTION
sci_generic_a nodim        # 1

is # INPUT SENSORS SECTION
# Glider coords
m_present_time timestamp   # 2

# Specify the baudrate of the uart, don't technically have to specify but to not forget
baud
9600 # default value

packet_size # don't technically have to specify but to not forget
600 # default value
```

**File 3.3: Current Field Test:** extctl.ini - used to control m_depth

words

### 3.2.4. tbdlist.dat

```
1 # Basic Data
2 sci_m_present_time
```

**File 3.4: Current Field Test:** tbdlist.dat - science computer's data file

As there are no scientific sensors, the only line called in this file is `sci_m_present_time`. There are no numbers after it because this runs every glider cycle.

### 3.2.5. simul.sim

```
1 just_electronics
```

**File 3.5: Current Field Test:** simul.sim - must be on glider simulator for simulating

This file must be in the `c:/config` directory of the FC and needs only have the above line. This tells the simulator to purely use electronics; i.e., it's not an actual glider with a pump, etc.

### 3.2.6. sbdlist.dat

```
1  # INTERVAL              Seconds since last stored value, 0 is store every value
2  # STATE                 State of glider (dive, hover, climb), 15 is always store
3  # HALFYOS               # of dive/climbs in this segment to record for, -1 store for all
4  # YO_DUTY_CYCLE         Store data every n'th yo in this segment, -1 is every dive/climb
5
6  # ------------------------------------------------------------------
7  # SENSOR NAME           INTERVAL  STATE     HALFYOS   YO_DUTY_CYCLE
8  # (Defaults)            0         15        -1        -1
9  # ------------------------------------------------------------------
10 m_tot_num_inflections   30      15      -1
11 m_lat                   30      15      -1
12 m_lon                   30      15      -1
13 m_x_lmc                 30      15      -1
14 m_y_lmc                 30      15      -1
15 c_heading               30      15      -1
16 m_heading               30      15      -1
17 m_depth                 30      15      -1
```

**File 3.6: Current Field Test:** sbdlist.dat - flight computer's data file

To have enough data to plot and thus see what transpired during the simulation over smaller timescales, for example if only running four yos to a depth of $\tilde{4}$0m for $\tilde{3}$0 min, we need data at a higher resolution. There are a lot of other values in the sbdlist.dat, but the above are the only ones changed. To cause less headaches later when processing and plotting the .sbd files put the interval and state to the same values.

### 3.2.7. sample90.ma

```
1 b_arg: sensor_type(enum)        90  # c_extctl_on
2 b_arg: state_to_sample(enum)    15      # all states
```

**File 3.7: Current Field Test:** sample90.ma - flight computer's sensor specific file

The big list of numbers in the file is only for reference, the number at the top (`sensor_type(enum)`) is what to change. This is where you change where it samples; i.e., when to turn the RasPi on and off. For this test it has been set to always be on so it doesn't turn off in the middle of the simulation.

### 3.2.8. yo99.ma

```
1 b_arg:  d_target_depth(m)       1000000  # u_mission_param_a
2 b_arg:  c_target_depth(m)        5      ### visual flag ###
```

**File 3.8: Current Field Test:** yo99.ma - flight computer's control of dive and climbs (yos)

Nothing else in this file was changed from default values.

## 3.3. Local Mission Coordinates: Setting Location and Direction of Fast Current Zone

The fast zone is described by `s_water_speed_fast_zone_x_min/max`. The values input to these lines are Local Mission Coordinates (LMCs). They are set to zero upon mission start (`run bs_cft.mi`) and are x and y values with units of meters. Positive Y in the direction of magnetic north at mission start. Positive X 90 degrees to the right from positive Y. Having the LMCs 1:1 with latitude and longitude makes simulating an ocean environment and plotting easier (this is why `s_mag_var` is set to 0).

The glider always uses `xs_water_speed` as the simulated ocean current velocity. Thus, in the simulation file there need not be a `xs_water_speed` set to anything, only the `s_water_speed` because that's the base water speed in the sim. When `s_water_speed_fast` is enabled (not set to -1) it overwrites the s_water_speed based upon the `s_water_speed_fast_zone_x_min/max`.

### 3.3.1. loadsim_cft.mi

```
1  sensor: u_max_time_in_gliderdos(sec) -1.0 # disable run "sequence" after this much time in
       gliderdos without receiving a keystroke
2
3  # initial direction before it is changed using SFMC script
4  sensor: s_water_direction(rad)  0.00            # due north (direction current is going TO)
5
6  # to simulate a fast current between LMC coordinates
7  # fast zone start < measured x lmc coordinate < fast zone end
8  sensor: s_water_speed_fast(m/s) 1.5             # in, set to -1 to disable
9  sensor: s_water_speed_fast_zone_x_min(m) 200    # in
10 sensor: s_water_speed_fast_zone_x_max(m) 1500   # in
11
12 # current has a simulated location of higher speed between LMC coordinates. It looks like this:
13 #           |                                        |
14 #   slow    |                   fast                 |       slow
15 #           |                                        |
16 #   s_water_speed_fast_zone_x_min                    |
17 #                                    s_water_speed_fast_zone_x_max
```

**File 3.9: Current Field Test:** loadsim_cft.mi - setting simulated variables

This file is used to initialize the starting simulated glider and ocean environment by calling `loadmission loadsim_cft.mi` before running the mission. This file:

1. Sets the wind forcing to zero (do not want wind effecting the glider while on the surface)
2. The magnetic variation to zero (want LMC to equal latitude and longitude for easier plotting)
3. The ocean to be infinitely deep (1337m) with no bottom wavelength (don't want the glider to touch the bottom at varying depths; need consistency)
4. Initial glider location (this file sets it to east of the Yucatan Peninsula)
5. Sets the current field location using LMC (sets the *initial* current direction (in radians) to due north)

## 3.4. Three ways to make a goto file

Gliders use coordinates in longitude and latitude order ordered by [degrees][minutes].[decimal minutes]; i.e., -8440.0 2100.0. There are a few ways one can make a goto file:

### 3.4.1. Manually

Using Google Maps online, Google Earth Pro, OpenCPN, or other method to find GPS coordinates, manually create the goto file. If there are not many coordinates (1-4 or so) this is usually the easiest method. Because we have one waypoint, it's easy to make these goto files by hand.

### 3.4.2. kml2goto.py

Using Google Earth Pro, plan a route and export as a kml file, then use the .py. There are instructions in the .py file. Easier when there are many (5+) waypoints.

### 3.4.3. opencpn2goto.py

Using OpenCPN, plan a route, copy from the route planner into Excel, export as a .csv, then use the .py. There are instructions in the .py file. Easier when there are many (5+) waypoints.

### 3.4.4. goto_l99.ma

```
1  bbehavior_name=goto_list
2  # --- goto_l99.ma
3  <start:b_arg>
4      b_arg: num_legs_to_run(nodim)    -1    # loop through waypoints
5      b_arg: start_when(enum)           0    # BAW_IMMEDIATELY
6      b_arg: list_stop_when(enum)       7    # BAW_WHEN_WPT_DIST
7          b_arg: initial_wpt(enum)      0    # 0 to n-1, -1 first after last, -2 closest
8        b_arg: initial_wpt(enum)        0 # first (only) wpt in list
9        b_arg: num_waypoints(nodim)     1 # number of waypoints in list
10 <end:b_arg>
11
12 <start:waypoints>
13 #      LON      LAT             name
14 #    -8630.0     2100.0     |  location of simulation start
15     -2522.0     2100.0     |  far far away (westernmost point of Africa)
16
17 <end:waypoints>
18 # head due east of Yucatan
```

**File 3.10: Current Field Test:** goto_l99.ma - commands the glider to go to a waypoint

WPT Due East (right) shows the location of the waypoint.

The practice of putting a waypoint far away while crossing a location of strong currents is a standard practice for glider pilots. Thus, the waypoint is placed far away from the glider because we want the heading to remain as close as possible to 090 degrees while the glider is deflected northward when traversing the current field.

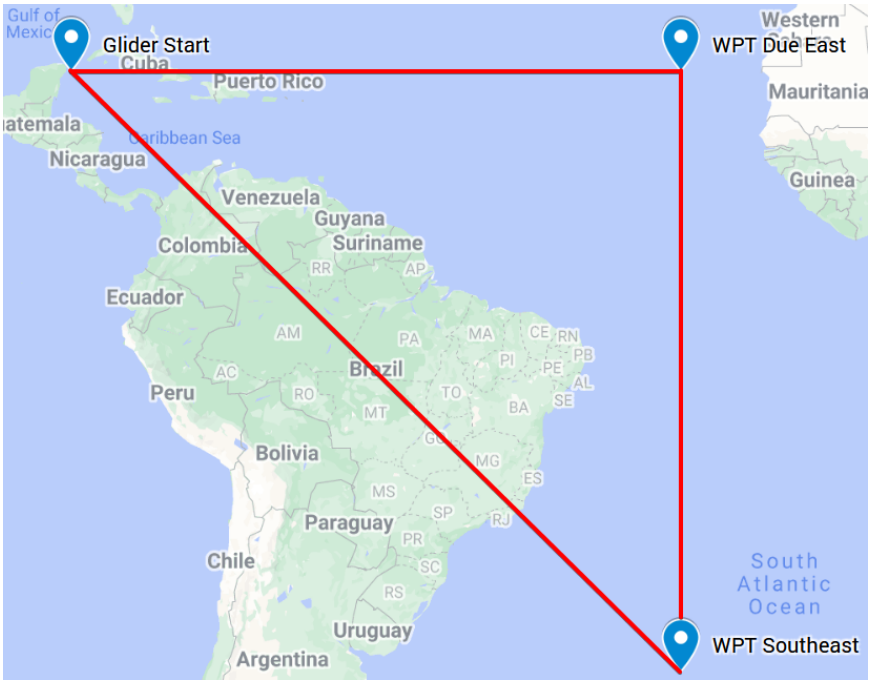


**Figure 3.4:** Waypoint Locations

## 3.5. RasPi script: depth_print.py

```python
import serial

uart_baud = 9600                # Desired baudrate for uart comms
send_chunk_size = 600           # amount of bytes per chunk when writing file to proglet
port_address = '/dev/ttyUSB0'   # Define comms port address

class extctl(object):
    def __init__(self, tty):
        self.port = serial.Serial(tty, baudrate=uart_baud, xonxoff=False, rtscts=False,
        dsrdtr=False, stopbits=serial.STOPBITS_ONE, parity=serial.PARITY_NONE,
        bytesize=serial.EIGHTBITS, timeout=None)

    def send(self, s):
        csum = 0
        # bs = s.encode('ascii')
        bs = s
        for c in bs:
            csum ^= c
        nmea = b'$' + bs + (b'*%02X\r\n' % csum)
        print(nmea)
        self.port.write(nmea)

    def write_SW(self, index, value):
        self.send(b'SW,%d:%f' % (index, value))

    def write_TXT(self,value):
        self.send(b'TXT,%f' % (value))

################################################################

serial.Serial(port_address, baudrate=uart_baud).close()
x = extctl(port_address)
port = serial.Serial(port_address, baudrate=uart_baud)

x.write_SW(0, 42)   # update mission_param_a one time
x.write_TXT(42)     # print value to glider terminal

print('d_target_depth␣updated,␣message␣sent')
```

**File 3.11: Current Field Test:** Raspberry Pi Script to change d_target_depth one time

## 3.6. Starting the Simulated Mission

Once the RasPi is hooked up and all the files are on the simulator we can begin the simulation! To do so, in the main direcrory of the flight computer type:

1. **`loadmission loadsim_cft.mi`:** This will initilaize the simulated ocean variables
2. **`report ++ m_depth`**: Displays m_depth, if this goes over 7m `u_mission_param_a` was updated
3. **`run bs_cft.mi`:** Run the mission
4. Once the glider starts diving, run the Python script to update the depth and print the value to the glider terminal

To run the script on the RasPi:

1. Open the RasPi's terminal and navigate to the folder containing the .py script using `cd [folder]` (following the manual's example, in the backseat_driver folder)
2. Once the glider has started diving, `sudo python3 depth_print.py` which will run the script. You should see the script's final print message in the terminal.

Once the script has been run, inspect the glider terminal for the message 42, and watch to see if m_depth goes > 7. If it does, `control-C` to cancel the mission. Restart the mission as before with the simulator and the RasPi, but don't use the `report` command. Once you update `u_mission_param_a` as before, let the simulation run for approximately 2.5 hours.

# 3.7. Plotting results using Python (current_field_test_simulation.ipynb/.py)

Before plotting we need to get the data files off the glider. This can be done a few ways (taking the simulator apart to get the CF card), but I recommend the following.

## 3.7.1. Getting the .sbd & .tbd files from the simulator

1. From the glider terminal change to the `c:/logs` folder using the command `cd logs` to get to the `c:/logs` directory.
2. `dir` to display all files in the directory.
3. The files at the bottom will have the most recent date. Inspect the ending number (for example, `2870012.sbd`)
4. `zs 28700*.sbd` uses a wildcard to send all the files from the mission.

## 3.7.2. cac Files

cac files are located in `c:/state/cache` on both the flight and science computer (shown right). In basic terms, these files hold the keys for processing the .sbd and .tbd files (the cac files to process the .tbd files are in the same folder on the science computer).

```
Directory of c:/state/cache/

./
../
06a36d4e.cac 116841     2024-01-22 21:46:10
aad86c9b.cac 116841     2024-01-22 21:46:10
b3d61675.cac 116841     2024-01-22 21:46:12
b9bd3e49.cac 118098     2024-01-24 19:41:10
65acaf6c.cac 118098     2024-01-24 19:41:12
9b83f4ec.cac 118098     2024-01-24 19:41:12
bb978bca.cac 116841     2024-01-28 23:49:42
5a250d8f.cac 118098     2024-01-29 00:02:38
12ed5f92.cac 118098     2024-02-21 00:00:00
11 files
```

**Figure 3.5: Current Field Test:** Cac files as seen from the glider terminal

## 3.7.3. Plots

**Left:** The blue line is the path of the glider through the current field boundaries, denoted by the vertical red dashed lines.

**Right:** The yo profile of the glider (blue line), with the max depth denoted by a red dashed line.

Python code to create these plots, along with some others not known, is on *Sophie's GitHub*.

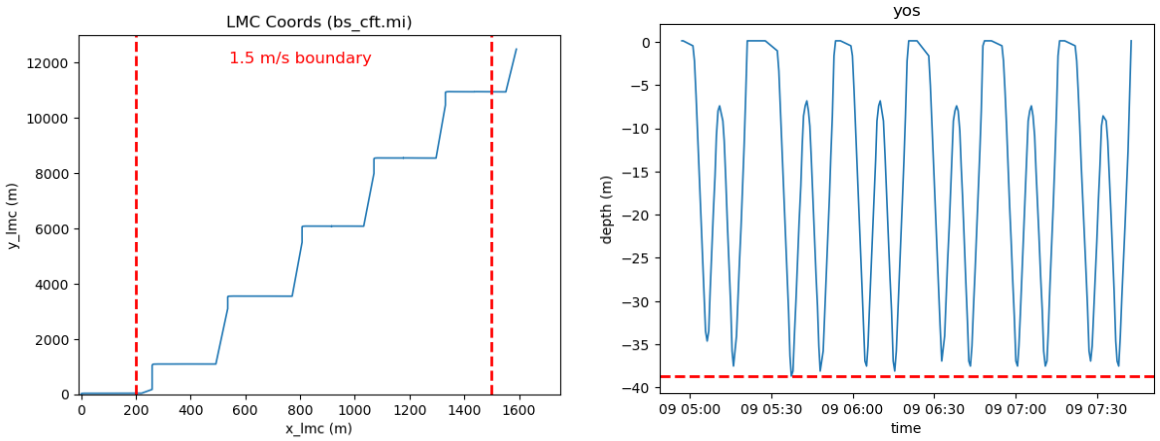


**Figure 3.6: Current Field Test:** Output plots, viewing by LMC coordinates

# 4

# Simulating a Large-Scale Ocean Current using an SFMC Script

**Chapter Concepts:**

1. An explanation of SFMC scripts and their typical use-cases.
2. Changing the sensors in the surface dialog.
3. Using the Glider Guidance System to find depth-averaged currents, then turning that information into an SFMC script using Python to build the .xml file.

## 4.1. Overview of SFMC Scripts

SFMC scripts allow for automated control of gliders while on the surface by looking for exact phrases from the gliders surface output and sending a varying list of commands based on user configuration. This is typically done to automate the retrieval of `.sbd` and `.tbd` files and to send `.ma` files then re-read them before resuming the mission. In our case, however, we will be using this ability to send commands to the surfaced glider mid-mission. There are five types of commands we utilize in our .xml.

```
−<gliderScript>
   <!-- The start state of the script. -->
  −<initialState name="checkNoComms">
    −<transitions>
      −<!--
          Check what glider surfaced for. If no_comms, send "ctrl-R"
       -->
       <transition matchExpression="Because:no comms for a while" toState="sendWaterSpeedDir"> </transition>
       <!-- Happens during testing so put here-->
       <transition matchExpression="Because:nothing commanded" toState="sendWaterSpeedDir"> </transition>
     </transitions>
   </initialState>
  −<!--
      From checkNoComms. This sends updated current direction before sendResume. This state "simulates" a current field
         by matching the longitude from surface dialogs, then !put s_water_direction and s_water_speed to what it should be for
         the location of the glider before diving. It only reads the logitude part of the surface dialog, from following part
         in quotes: sensor:m_lon(lon)='-8630'

   -->
  −<state name="sendWaterSpeedDir">
    −<transitions>
      −<transition matchExpression="=-8630" toState="sendResume">
         <action type="glider" command="!put s_water_direction 0.7853981633974483"> </action>
         <action type="glider" command="!put s_water_speed 0.6"> </action>
       </transition>
```

**Figure 4.1: SFMC Script Example:** initialState, matchExpression, toState, state, action

### 4.1.1. Breakdown of How a Script Works: sendWaterSpeedDir.xml

Using Figure 4.1 (above) as the example:

1. **initialState name =”checkNoComms”:** When selected in SFMC the script starts in this state, then waits for a glider surfacing to be detected. When a glider is detected it matches words in the surface dialog, then;
2. **matchExpression=”Because:no comms for a while”:** From the initialState we match an expression (words and/or numbers from the surface dialog), then;
3. **toState=”sendWaterSpeedDir”:** Send the script to another state named ”sendWaterSpeedDir”.
4. **state name=”sendWaterSpeedDir”:** Once called, this state containing commands to send to the glider by first matching an expression (in this case -8630, longitude without a decimal), then;
5. **command=”!put s_water_direction 0.78”:** The command send to the glider. For example, the command ”Ctrl-R” would send the resume mission command.

In this way, any value of longitude can have any value for `s_water_speed` and `s_water_direction` within the sendWaterSpeedDir state.

### 4.1.2. Changing the Surface Dialog: Adding Sensors to matchExpression With

In these script examples, the script reads the phrase `calculations COMPLETE`, but if you desire the script to read a sensor name and/or value (like `m_lon`) , it’s possible to put a sensor in the surface dialog. View possible commands in the gliderDOS terminal by typing `help`. From there, we can find `sfr_display`, which is how we control what sensors are displayed in the surface dialog. If we send `help srf_display` we get the following:

```
SRF_DISPLAY + <sensor_name ..> | - <sensor_name ..> | clearall | list ; config (or list) which
sensors to display in surface dialog
```

For example, `srf_display + c_heading` will show that sensor in the surface dialog. We can verify that it is in the surface dialog by sending `srf_display list`.

```
GliderDos I -3 >help

?                   ballast             ballvalve           boot
callback            capture             CAT                 CD
CHDIR               CLRDEVERRS          compass_cal         consci
COPY                core                CP                  date
DEL                 DELLOG              DEVICES?            DF
digifin             DIR                 drift_table         echo
EXECUTE             exit                freport             GET
glider_test         HARDWARE?           HEAP                HELP
hs                  LAB_MODE            LIST                loadmission
logging             LONGTERM            LONGTERM_PUT        LS
MBD                 MKDIR               MV                  PRUNEDISK
PURGELOGS           PUT                 RENAME              REPORT
RM                  RMDIR               run                 SBD
SEND                sequence            SETDEVLIMIT         SETNUMWARN
SIMUL?              SRF_DISPLAY         strobe              sync_time
SZR                 SZS                 talk                tvalve
TYPE                USE                 VER                 WHERE
whoami              whoru               WHY?                wiggle
ZERO_OCEAN_PRESSUREZR                   ZS

     help                list all commands in alphabetical order
     help -full          list all commands with their usage and descriptions
     help -desc          list all commands with their descriptions
     help -usage         list all commands with their usage
     help <cmd> .. <cmd> prints the full help msgs for listed commands.
                         Note that -desc and -usage can also be used
     command names in lower case are NOT executable in mission via !
```

**Figure 4.2: SFMC Script:** help command on glider

## 4.2. Glider Simulator Limitations

1. The simulation only runs is real-time, meaning there is no way to ”speed” up the simulation to make it faster. This means if you're simulating a 200km ocean crossing it will take approximately five days. The simulator has been designed to operate real glider parts on a glider (was not originally made with a the blue shoebox simulators in mind).
2. Ocean and atmosphere parameters are all static (with one exception). They are set once and stay that way all mission. Wind speed and direction, water depth, and current direction are all static. The one exception is water speed (as explained in Chapter 3, there is a built in method to make a static fast zone of current), but that solution does not allow for a variable speed and direction like an ocean current.

The glider simulators simulated fast current zone from Chapter 3 could be thought of as the spiritual predecessor to the following method of simulating an ocean current with a glider simulator. In order to: (1) achieve a (*mostly*) realistic current field similar to the Yucatan Current and (2) ensure the external controller's Python script correctly updated the heading based upon the measured current field we use an SFMC script to change the current direction based upon the longitude of the glider. This is achieved by the SFMC script as previously explained.

Besides the single-time initialization of current magnitude and direction, which we overcome here with the SFMC .xml script, the remaining limitation of the simulated ocean is the uniform current velocity with depth; however, for the purposes of the external controller's Python program that uses current vectors this is acceptable because near real-time depth-averaged current (DAC) data from ocean models is readily available. Thus, by using the DAC for 0-1000m we retain a close enough approximate for our purposes.

## 4.3. Plotting Depth-Averaged Currents Using GGS: Yucatan Channel Example

The Glider Guidance System (GGS) made by Sal Fricano has many uses. One of them is pulling data from the RTOFS (Global Real-Time Ocean Forecast System) ocean model, calculating the DAC, then creating a stream plot based on coordinate inputs. For information on how to use the GGS please see Sal Fricano's GitHub. Using (an early version of) GGS for depth-averaged current data for the Yucatan Current the following table was made by inspecting current speed and directions in a grid pattern:

| **Latitude** | **Direction °T / Magnitude m/s** | | | | | | |
|---|---|---|---|---|---|---|---|
| **21.50°** | 000 | 340 | 350 | 355 | 350 | 000 | 000 |
| | 0.0 | 0.1 | 0.2 | 0.3 | 0.1 | 0.1 | 0.1 |
| **21.25°** | 000 | 020 | 020 | 020 | 020 | 000 | 310 |
| | 0.0 | 0.0 | 0.2 | 0.3 | 0.1 | 0.1 | 0.1 |
| **21.00°** | 000 | 045 | 045 | 045 | 045 | 0 | 270 |
| | 0.0 | 0.3 | 0.3 | 0.3 | 0.1 | 0.0 | 0.1 |
| **20.75°** | 000 | 045 | 045 | 060 | 090 | 130 | 250 |
| | 0.0 | 0.5 | 0.4 | 0.3 | 0.1 | 0.0 | 0.1 |
| **20.50°** | 045 | 045 | 050 | 085 | 100 | 145 | 170 |
| | 0.6 | 0.6 | 0.4 | 0.3 | 0.3 | 0.3 | 0.3 |
| **Longitude** | **86.5°** | **86.25°** | **86.00°** | **85.75°** | **85.50°** | **85.25°** | **85.00°** |

**Table 4.1: Yucatan Depth-Averaged Currents:** Current magnitude and direction in table form

Then, as the SFMC .xml script **only incorporates longitude**, I approximated a cross-section of longitude (a change in latitude wasn't required to verify proper behavior). Using this table as reference, the SFMC .xml script was made.

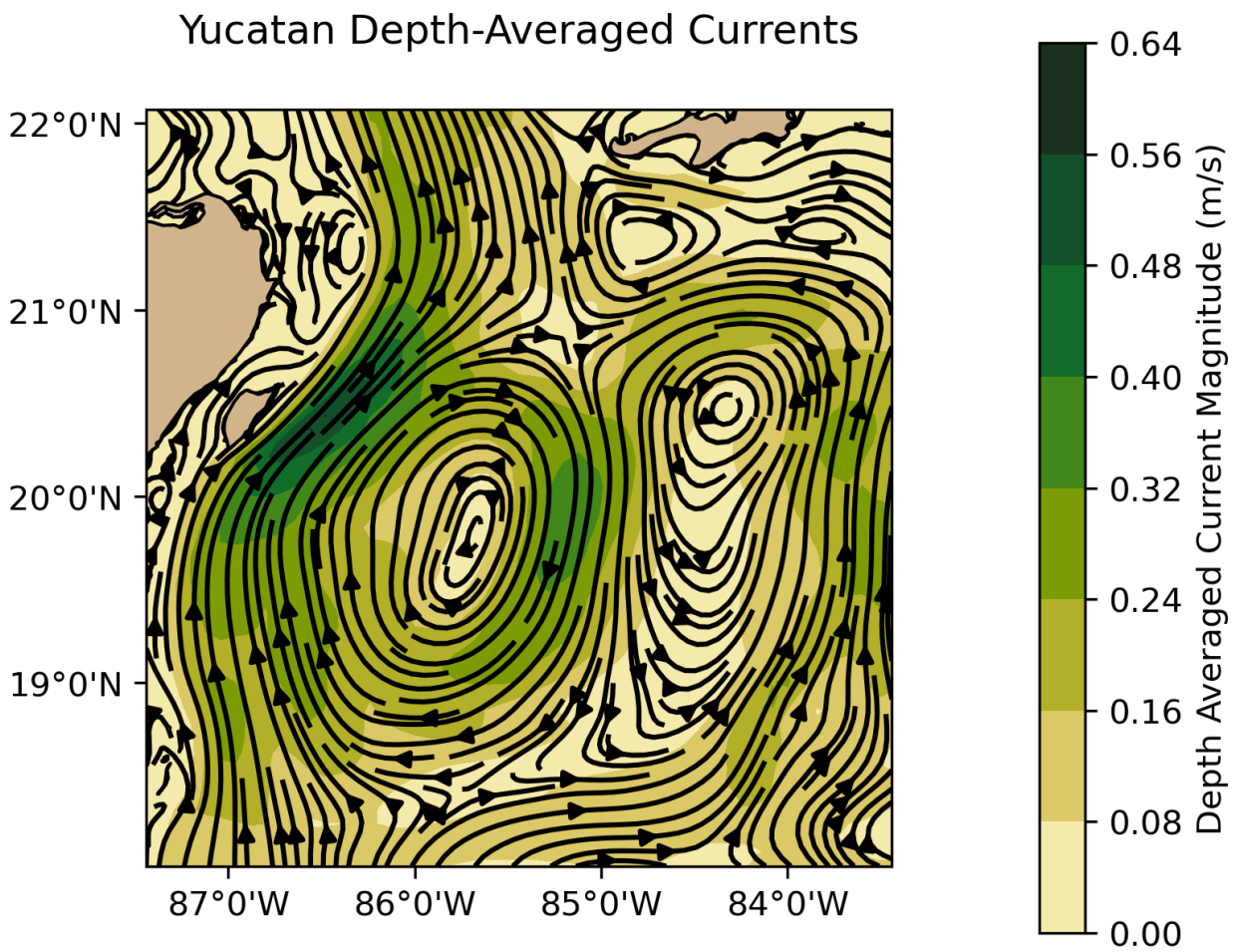


**Figure 4.3: Yucatan Depth-Averaged Currents:** An early output from the Glider Guidance System

## 4.4. Creating the SFMC Script to Control Simulated Variables (sendWaterSpeedDir.xml)

As discussed, the glider sets the simulated ocean variables (`s_water_speed`, `s_water_direction`) once at mission start. To work around this we the SFMC script to read the glider's longitude from the surface dialog, then have the script change the ocean variables during the surfacing before commanding the glider to resume the mission.

The .xml script reads the expression "`Calculations COMPLETE`" instead of the typical `Because: no comms for a while` or `Because: timeout expired` because the simulator is always connected to SFMC. We also want to update and resume only *after* the glider has finished it's water velocity calculations. The by-product of this method is a more accurate representation of time at the surface. If, for example, we used the timeout expired expression the glider would resume the mission and dive immediately upon surfacing, which would not: (1) give time to calculate the water velocity and not; (2) more accurately representing the sending of files when surfacing during a mission.

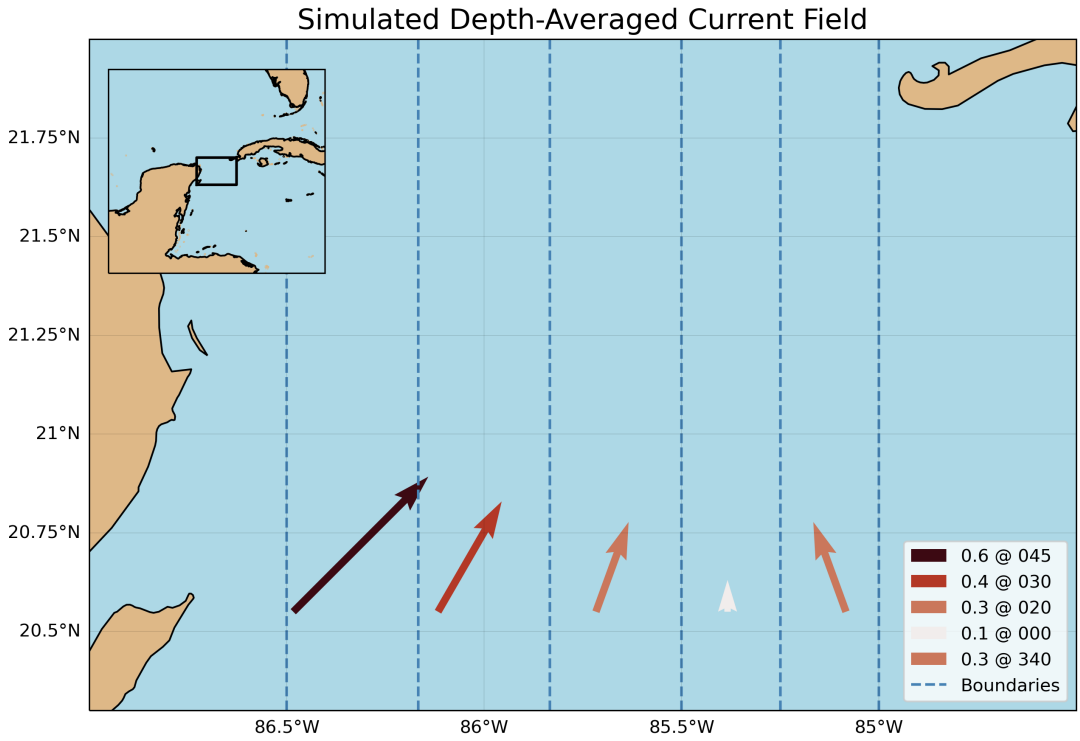


**Figure 4.4: SFMC Script:** Blue dashed lines delineate the boundaries between changing currents, with the arrow in each vertical slice of longitude indicating the magnitude and direction of currents within.

The current field only changes in the longitude direction. If one desired a more accurate current simulation in a glider simulator the code could be adapted to incorporate latitude as well.

[this is empty space]

### 4.4.1. SFMC Script Part One: Building the simulated ocean current field

```
# IMPORTS

start_lon = -86.5
end_lon = -85
filename = 'znew6things_sendWaterSpeedDir.xml'
wheresave = '../f_SFMC_script/'
complete_filename = os.path.join(wheresave,filename)

# how wide to make the sections across the ocean area
# sections, 20, 20, 20, 15, 16
sec0 = 20
sec1 = 20
sec2 = 20
sec3 = 15
sec4 = 16

# location of sections
pltsec0 = 0
pltsec1 = sec0+sec1-1
pltsec2 = sec0+sec1+sec2-1
pltsec3 = sec0+sec1+sec2+sec3-1
pltsec4 = sec0+sec1+sec2+sec3+sec4-1

# for current direction, make length of each section
# if not all good lengths get lots of errors
west0 = [45]*sec0
west1 = [30]*sec1
west2 = [20]*sec2
west3 = [0]*sec3
west4 = [340]*sec4

# put all the things above into one
water_dir_deg = west0+west1+west2+west3+west4

# convert each degree into radians, needed for plots
rads = []
for i in water_dir_deg:
    rads_loop = math.radians(i)
    rads.append(rads_loop)

# for current speed, make length of each section
speed0 = [0.6]*sec0
speed1 = [0.4]*sec1
speed2 = [0.3]*sec2
speed3 = [0.1]*sec3
speed4 = [0.3]*sec4

# put all above into one
water_speed = speed0 + speed1 + speed2 + speed3 + speed4

# convert the start and end lon to proper Python mappable
lon_range_str,lon_map = decimal_degrees_to_degrees_minutes(start_lon,end_lon)
loopnum = len(lon_range_str) # for generating the .xml script, how many values there are (91 in
      this case)
lat_map = [20.55]*len(lon_map) # for plotting, but must be the same length as lon_map

# if these things aren't equal it's bad
print(f'Length of things should be equal: {len(lon_map)} = {len(lon_range_str)} = {len(rads)} =
      {len(water_speed)} = {len(lat_map)}')
```

**File 4.1: SFMC Script:** Building the simulated ocean current field

This part programmatically builds each section or vertical slice of longitude as seen in Figure 4.4 to have an associated magnitude and directions. We will use this same chunk later for build the SFMC .xml script and the map plot of glider locations.

### 4.4.2. SFMC Script Part Two: Defining .xml script states

```
header = (
    '<gliderScript>\n\n'
    )

checkNoComms = (
    '    <!-- The start state of the script. -->\n'
    '    <initialState name="waterCalcComplete">\n'
    '        <transitions>\n'
    # '            <!-- start_when(enum) 9 says this at the top of surface. Send to to sendWaterSpeedDir-->\n'
    # '            <transition matchExpression="Because:timeout expired" toState="sendWaterSpeedDir">\n'
    # '            </transition>\n'
    # '            <!-- Check what glider surfaced for. If no_comms, send to to sendWaterSpeedDir"-->\n'
    # '            <transition matchExpression="Because:no comms for a while" toState="sendWaterSpeedDir">\n'
    # '            </transition>\n'
    # '            <!-- Happens during testing so put here. Send to to sendWaterSpeedDir -->\n'
    # '            <transition matchExpression="Because:nothing commanded" toState="sendWaterSpeedDir">\n'
    # '            </transition>\n'
    '            <!-- Must wait for glider to calculate water velocity! Send to to sendWaterSpeedDir -->\n'
    '            <transition matchExpression="COMPLETE" toState="sendWaterSpeedDir">\n'
    '            </transition>\n'
    '        </transitions>\n'
    '    </initialState>\n\n')

sendWaterSpeedDirHeader = (
    '<!-- From checkNoComms. This sends updated current direction before sendResume. This state "simulates" a current field\n'
    '    by matching the longitude from surface dialogs, then !put s_water_direction and s_water_speed to what it should be for\n'
    '    the location of the glider before diving. It only reads the logitude part of the surface dialog, from following part\n'
    '    in quotes: sensor:m_lon(lon)=\'-8630\'\n'
    '    -->\n'
    '    <state name="sendWaterSpeedDir">\n'
    '        <transitions>\n')

footer = (
    '<!-- SEND RESUME/DIVE ======================================== -->\n'
    '    <state name="sendResume">\n'
    '        <transitions>\n'
    '            <transition matchExpression="xxx command verify fail xxx" toState="sendResume">\n'
    '            </transition>\n'
    '            <!-- Ready to dive, but send report first! MUST SEND TWO CTLR-R\'S OR ELSE IT DOESN\'T WORK\n'
    '                 Script needs more time after the !put so sends two -->\n'
    '            <transition matchExpression="Hit Control-R to RESUME" toState="waterCalcComplete">\n'
    '                <action type="glider" command="Ctrl-R">\n'
    '                </action>\n'
    '                <action type="glider" command="Ctrl-R">\n'
    '                </action>\n'
    '            </transition>\n'
    '            <transition timeout="10" toState="waterCalcComplete">\n'
    '            </transition>\n'
    '        </transitions>\n'
    '    </state>\n'
    '\n'
    '    <finalState name="final">\n'
    '    </finalState>\n'
    '</gliderScript>\n')
```

**File 4.2: SFMC Script:** Defining .xml script states

### 4.4.3. SFMC Script Part Three: Building the .xml script

```
file = open(complete_filename, 'w')
file.write(header)
file.write(checkNoComms)
file.write(sendWaterSpeedDirHeader)
for zz in range(0, loopnum):
    # file.write(f'           <transition matchExpression="   sensor:m_lon(lon)={lon_range_str[zz]}" toState="sendResume">\n')
    file.write(f'           <transition matchExpression="={lon_range_str[zz]}" toState="sendResume">\n')
    file.write(f'              <action type="glider" command="!put s_water_direction {rads[zz]}">\n')
    file.write('              </action>\n')
    file.write(f'              <action type="glider" command="!put s_water_speed {water_speed[zz]}">\n')
    file.write('              </action>\n')
    file.write(f'              <action type="glider" command="!put X_PRIOR_SEG_WATER_VX 0">\n')
    file.write('              </action>\n')
    file.write(f'              <action type="glider" command="!put X_PRIOR_SEG_WATER_VY 0">\n')
    file.write('              </action>\n')
    file.write(f'              <action type="glider" command="!put M_WATER_VY 0">\n')
    file.write('              </action>\n')
    file.write(f'              <action type="glider" command="!put M_WATER_VX 0">\n')
    file.write('              </action>\n')
    file.write('           </transition>\n')
file.write('       </transitions>\n')
file.write('   </state>\n')
file.write('\n')
file.write(footer)
```

**File 4.3: SFMC Script:** Building the .xml script

Within the python notebook to make and export the .xml script used, there are diagnostic plots to visualize the simulated ocean and location of gliders.

### 4.4.4. Alternate Current Field

To test in principle if another current field also works, I made this and tested this alternate. Further details on GitHub.

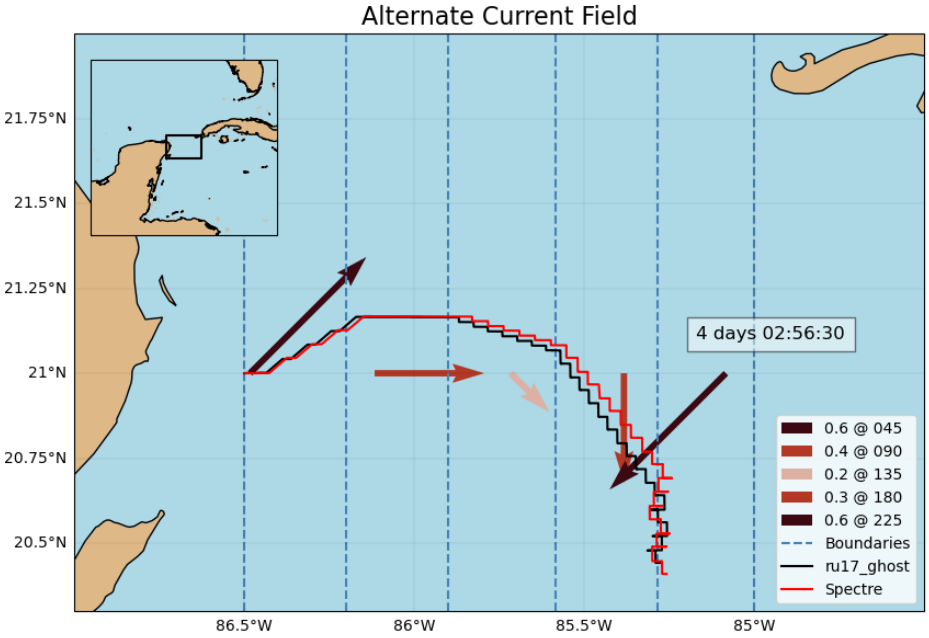


**Figure 4.5: SFMC Script Alternate Current Field:** Blue dashed lines delineate the boundaries between changing currents, with the arrow in each vertical slice of longitude indicating the magnitude and direction of currents within.

# 5

# Verify .xml Script Behavior

**Chapter Overview**
In this mission we will use the goto_list behavior of the glider and:

1. Not use the external controller (RasPi)
2. Fly the simulation glider through our SFMC current-created field.

Before using the script to test proper BSD, we must verify proper .xml script behavior. This also allows us to see what the glider will do when given waypoints instead of controlling the heading from the external controller.

## 5.1. Files Used

The rest of the glider files remain extremely similar to what was gone over in the current field test, so please see the GitHub for mission files.

### 5.1.1. goto_l98/97.ma

```
# goto_l98.ma   # filename
<start:waypoints>
#     LON     LAT            name
#   -8630.0    2100.0    |  location of simulation start
    -2522.0    2100.0    |  far far away (westernmost point of Africa)

# goto_l97.ma   # filename
<start:waypoints>
#     LON     LAT            name
#   -8630.0    2100.0    |  location of simulation start
    -2522.0   -3612.0    |  to give the initial heading hopefully 45deg more
```

**File 5.1: Verify .xml Script Behavior:** goto_l98/97.ma - waypoints

These waypoints were selected following the standard practice for glider piloting when crossing strong currents (Figure 5.2). By putting a waypoint far away in the desired direction of travel a nearly-consistent heading is maintained (Figure 5.3). Thus, I picked WPT Due East by continuing on the parallel of latitude, picking the westernmost point of Africa for longitude at 21N. For WPT Southeast I used navigational triangles and Google Maps to approximate a 45 degree angle and thus an initial heading of approximately 135T.

## 5.2. Uploading and Selecting the SFMC .xml Script

Now that we have our .xml script, we must put it in SFMC and activate it in the glider terminal.

### 5.2.1. Upload .xml script to SFMC

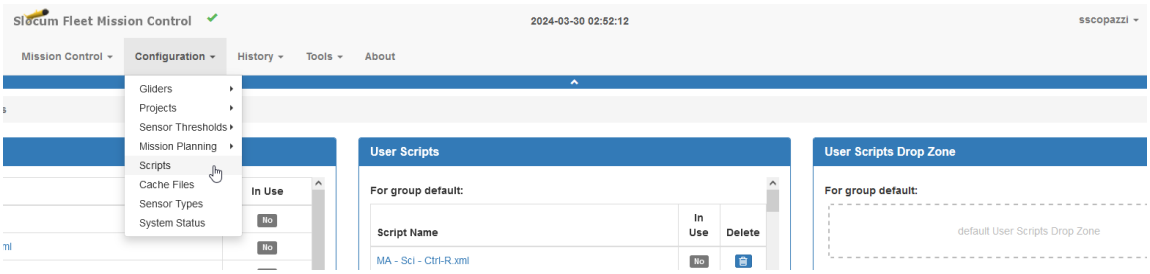


**Figure 5.1: SFMC Script:** Where to upload scripts in SFMC

Under ”Configuration” select ”Scripts”. From there, drop the file in the ”default User Scripts Drop Zone” on the far right. If you have many scripts (like we do at Rutgers) used for day-to-day piloting, I recommend using a prefix of ”z_sim_” so the simulation scripts are only at the bottom and they can’t be mistaken for anything else.

### 5.2.2. Select .xml script (sendWaterSpeedDir.xml)

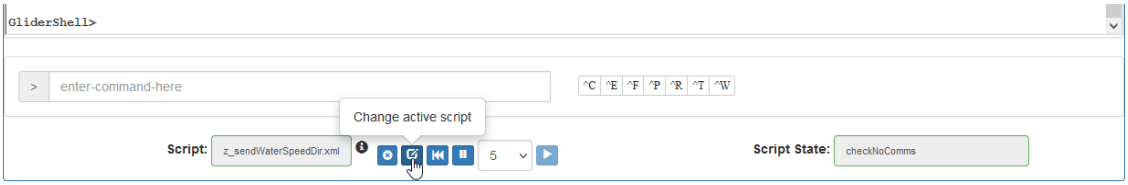


**Figure 5.2: SFMC Script:** Where to select scripts in SFMC glider terminal

In the glider’s terminal, immediately below the enter command box, select ”Change active script” and select the .xml script. If the script was made correctly, the name of the script will show up on the left side box, and the ”Script State” will populate with checkNoComms. If after selecting the script nothing shows up in these boxes it means something is wrong with the formatting of the .xml file. There is a small bug in SFMC that causes an error message to not appear (through communication with Teledyne they now now know this is a bug), but it’s safe to assume if the box doesn’t turn green with the name of the script, it has a formatting error.

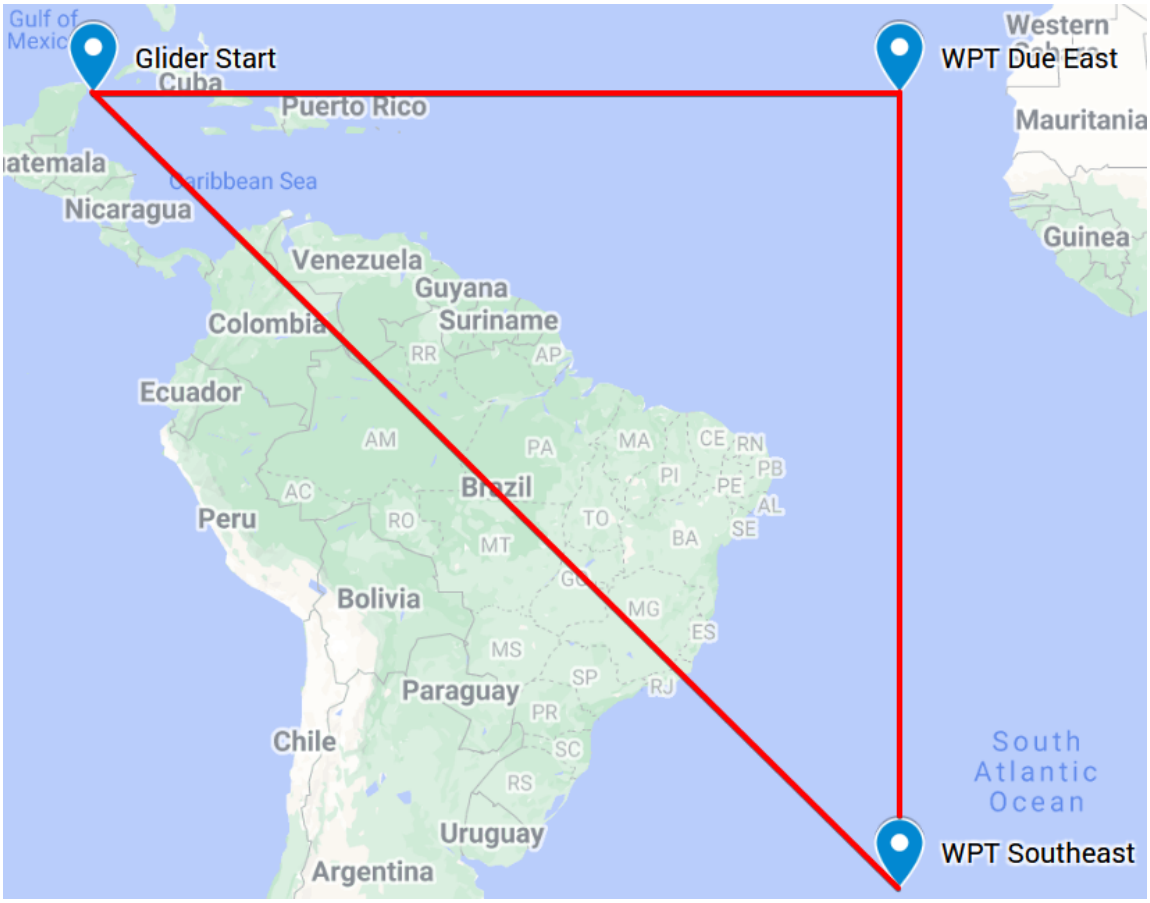


**Figure 5.3: SFMC Script:** Location of waypoints used for both missions.

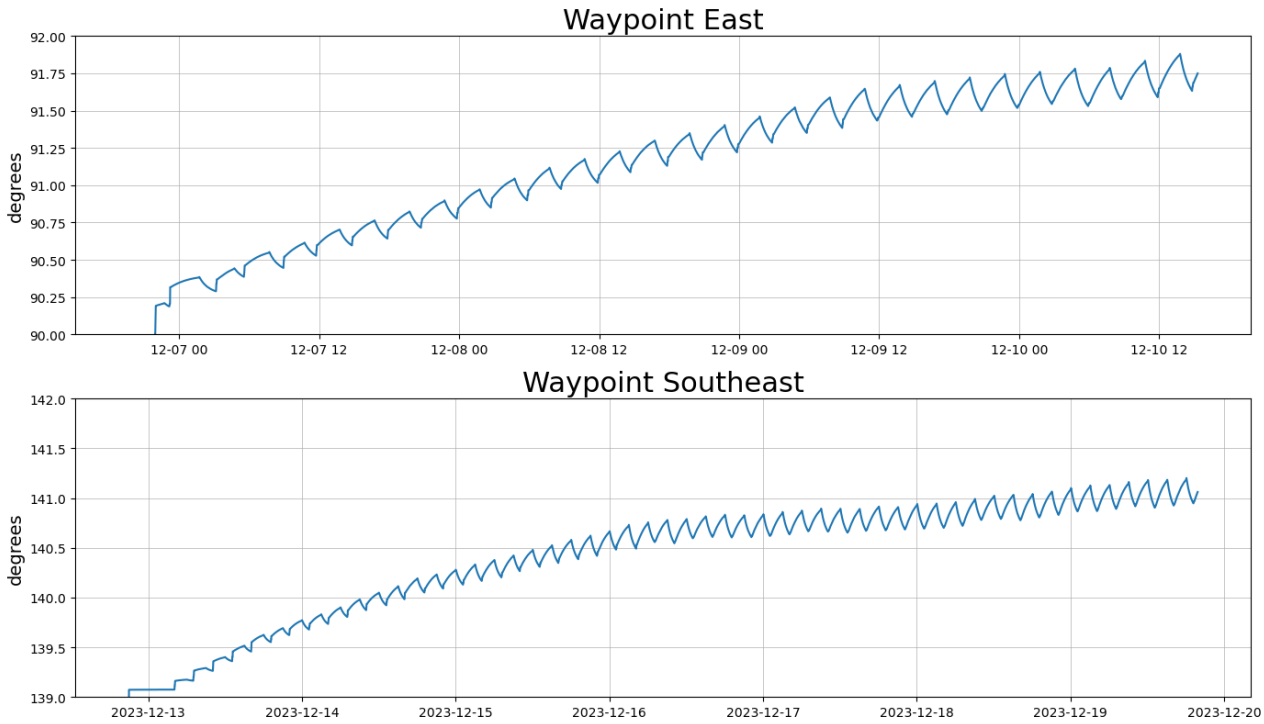


**Figure 5.4: SFMC Script:** `c_heading` comparison between both missions

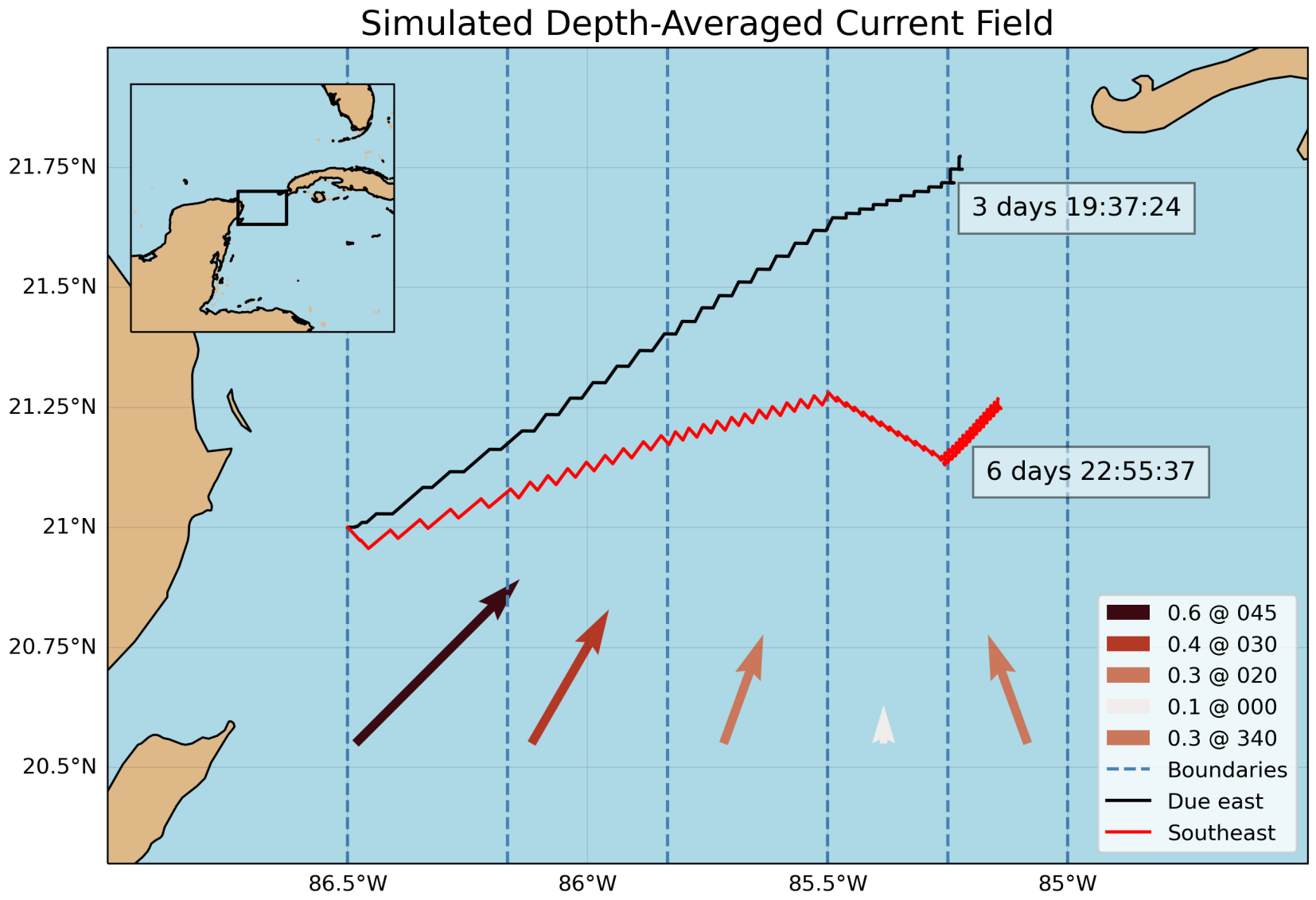


**Figure 5.5: SFMC Script:** Blue dashed vertical lines are longitude current boundaries. Arrows are current direction and magnitude within each boundary. See figure 5.2 for waypoint locations.

# 6

# Simple Backseat Driver Behavior Application: Commanding `c_heading`

**Chapter Overview**
In this mission we will use the set_heading behavior of the glider and:

1. Have the RasPi remain powered on for the duration of the mission.
2. Have the RasPi receive `m_water_vel_dir` from the glider.
3. Calculate the new heading 90° to the right of `m_water_vel_dir`.
4. Send the glider a new heading while also printing the new heading to the glider terminal.

## 6.1. Flying 90° From Depth-Averaged Currents

### 6.1.1. extctl.ini

Strictly speaking we do not need to define a `baud_rate` or `packet_size` as it defaults to 9600 and 600 respectively, but I left them in for visual clarity.

```
# c_heading in set_he95.ma, used with b_arg: heading_value(X) 1000000
                            # INDEX NUMBER FOR RASPI PYTHON
u_mission_param_a           # 0

os # OUTPUT SENSORS SECTION
sci_generic_a nodim         # 1

is # INPUT SENSORS SECTION
# Glider coords
m_present_time timestamp    # 2
m_water_vel_dir rad         # 3

# Specify the baudrate of the uart, don't technically have to specify but to not forget
baud
9600 # default value

packet_size # don't technically have to specify but to not forget
600 # default value
```

**File 6.1: Simple Backseat Driver:** extctl.ini - used to control c_heading

### 6.1.2. set_he95.ma

The required .ma file for the `set_heading` behavior.

```
behavior_name=set_heading
<start:b_arg>
        b_arg: use_heading(bool)        1
    b_arg: heading_value(X)     1000000 # u_mission_param_a
<end:b_arg>
```

**File 6.2: Simple Backseat Driver:** set_he95.ma file for set_heading behavior

### 6.1.3. right.mi

Instead of goto behavior, we use the `set_heading behavior` so the RasPi can update the glider's heading.

```
sensor: u_allowable_cycle_overrun(msec) 3000        # how large x_cycle_overrun_in_ms can be
sensor: u_mission_param_a(nodim)        1.570796      # 90deg = radians. Must init something
sensor: u_extctl_debug(nodim)           1             # Received messages from RasPi displayed
     only
                            # helps w/ debug and knowing if it's working

# HEADING: Command heading
behavior: set_heading
    b_arg: args_from_file(enum) 95 # >= 0 enables reading from mafiles/set_he<N>.ma
    b_arg: start_when(enum) 0      # Options: 0,1,2,6,7,8,15
    b_arg: end_action(enum) 2      # 0 = quit, 2 = resume
```

**File 6.3: Simple Backseat Driver:** right.mi - mission to fly 90° from ocean current

## 6.2. How the Raspberry Pi reads the Glider's Message

The RasPi progressively splits the incoming `cur_line` from the glider using delimiters. Following each `*.split` command the output is shown.

```
cur_line = '$SW,0:0.12,1:508.0*3B$SW,0:0.24,1:520.0*3B'
   # incoming line from the science computer to the RasPi
msg = cur_line.split('*')
   # output: msg = ['$SW,0:0.12,1:508.0', '3B$SW,0:0.24,1:520.0', '3B']
   # has been broken into two separate messages, or however many there are
   # pulls the first, index 0 (I assume because this is the most recent?)

fields = msg[0].split(',')
   # output: fields = ['$SW', '0:0.12', '1:508.0']
   # further splits the message by commas

for subscription in fields:
   # Loops through for each item in the fields variable
   # subscription because this is what glider value's we asked for in
   # the is section of the extctl.ini file
    if '0:' in subscription:
       # If a field starts with ‘’:0 it does the following
        subs = subscription.split(':')
           # output: subs = ['0', '0.12']
           # Splits the field's value that starts with '0:' into two parts

        first_value = float(subs[1])
           # put the first value into a variable, type float
           # this is where I do the heading math, then write new heading to glider
        print(f'First_Value: {first_value}')
           # print value to RasPi terminal

    if '1:' in subscription:
        # does the same, but for '1:'
        subs = subscription.split(':')
        second_value = float(subs[1])
        print(f'Second_Value: {second_value}')
```

If no value is found the script ends. To see how to wait until all values are received see the advanced version in Chapter 10.

### 6.2.1. Simple Application: Raspberry Pi Script

Class extctl is from Teledyne's manual, although here we define the baud, port_address and port at the top of the script initializing the values once.

```
import serial
import numpy as np

# Desired baudrate for uart comms
baud = 9600
# amount of bytes per chunk when writing file to proglet
send_chunk_size = 600

# Define comms port address
port_address = '/dev/ttyUSB0'
port = serial.Serial(port_address, baudrate=baud)

class extctl(object):
    def __init__(self, tty):
        self.port = serial.Serial(tty, baudrate=baud, xonxoff=False, rtscts=False,
        dsrdtr=False, stopbits=serial.STOPBITS_ONE, parity=serial.PARITY_NONE,
        bytesize=serial.EIGHTBITS, timeout=None)

    def send(self, s):
        csum = 0
        # bs = s.encode('ascii')
        bs = s
        for c in bs:
            csum ^= c
        nmea = b'$' + bs + (b'*%02X\r\n' % csum)
        print(nmea)
        self.port.write(nmea)

    def write(self, index, value):
        self.send(b'SW,%d:%f' % (index, value))

    @staticmethod
    def read(self):
        #ser = serial.Serial(port_address, baudrate=uart_baud)
        #line = ser.readline()
        line = self.port.readline()
        return line

x = extctl(port_address)

cur_line = str(port.readline())     # full line coming from glider
#print(f'cur_line: {cur_line}')     # view full line if viewed from RasPi
msg = cur_line.split('*')           # Split into separate fields
fields = msg[0].split(',')          # Split into separate fields
print(f'Fields: {fields}')

# the ifs MUST match the num order in extctl.ini
for subscription in fields:
    if '3:' in subscription:              # index number of m_avg_speed
        print('found 3: -> processing!')
        subs = subscription.split(':')    # from TWR
        m_water_vel_dir = float(subs[1])

        new_heading = m_water_vel_dir + (np.pi / 2)    # Adding 90 degrees still in radians
        new_heading = np.round(new_heading,5)
        print(f'From Glider (m_water_vel_dir): {m_water_vel_dir} | From RasPi (perp_angle): {
            new_heading}') # see them for debug
        x.write(0,new_heading)              # change u_mission_param_a
        glider_terminal_msg = new_heading
        x.write_TXT(glider_terminal_msg)    # send the new_heading to the glider terminal to
            see when it is updated in SFMC
        # this way you don't have to SSH into the RasPi and use tail -f on the cronlog
```

**File 6.4: Simple Backseat Driver:** sendHeadDir.py - Python script ran on the Raspberry Pi BSD

The rest of the files used in the mission remain unchanged from previous.

## 6.3. 90deg_bsd Test Results

For the first test with BSD controlling the heading we have the RasPi powered on the entire time. I would recommend this for a first test because it eliminates one place errors may occur and the ease of SSHing into the RasPi to see what fields are sent on each cycle for debugging purposes.

**It is important to note** that when the desired value is not found the RasPi prints the fields to end the script. If the script is not ended, because the CronJob runs the script every five seconds, the script ”hangs” on the if statement. This leads to a ”que” of scripts on the RasPi that ”pile up”, potentially leading to unintended consequences with RasPi behavior (memory usage crashes).

The following figure shows the `c_heading` for the backseat driver mission. Note the steps of `c_heading`, as it was updated every three hours upon new calculation of `m_water_vel_dir` as the heading driven glider, controlled by backseat driver, proceeded through the current field.
Not every message contains the desired value corresponding to `m_water_vel_dir` (Figure 6.1's mission it was position 5 and not 3 as shown previously).

```
Fields: ["b'$SD", '4:0.231009111066112']
Fields: ["b'$SD", '2:1703229641.789', '3:1.65108147238664']
Fields: ["b'$SD", '5:0']
found 5: -> processing!
From Glider (m_water_vel_dir): 0.0 | From RasPi (perp_angle): 1.5708
b'$SW,0:1.570800*37\r\n'
Fields: ["b'$SD", '4:0.231192270006687']
Fields: ["b'$SD", '4:0.23123812680972']
Fields: ["b'$SD", '2:1703229661.811', '3:1.59872159482681']
Fields: ["b'$SD", '4:0.231097803694824']
Fields: ["b'$SD", '4:0.231143754964168']
Fields: ["b'$SD", '5:0']
found 5: -> processing!
From Glider (m_water_vel_dir): 0.0 | From RasPi (perp_angle): 1.5708
b'$SW,0:1.570800*37\r\n'
Fields: ["b'$SD", '2:1703229681.831', '3:1.54461638801498']
Fields: ["b'$SD", '4:0.231326730991636']
```

**Figure 6.1: Simple Backseat Driver:** Cronlog from Simple Backseat Driver Application: The fields in variable cur_line on each glider cycle. Note intermittent ”5:”, processing, and sending of new heading from RasPi to glider.

Figure 6.2 illustrates how the RasPi properly updated the heading of the glider mid-mission every three hours as desired. However, the heading has a lot more steps than five (one for each current boundary it passed through). This is due to how the glider simulator itself calculates `m_water_vel_dir`.

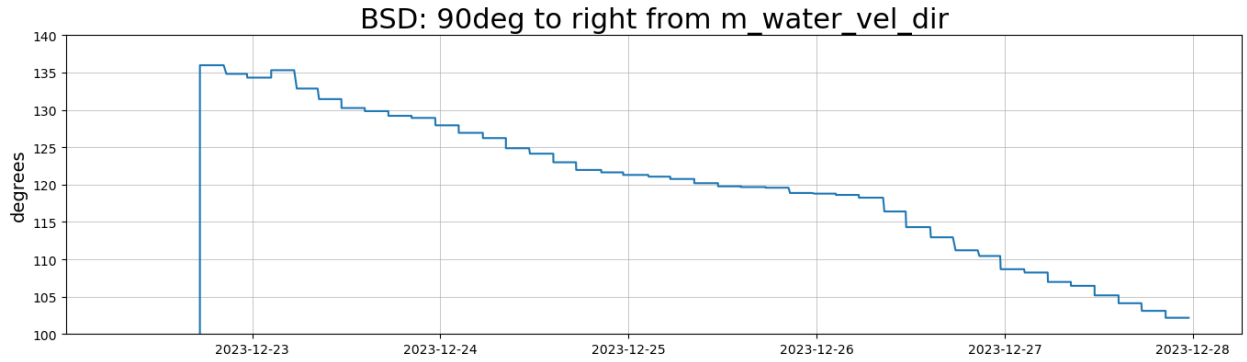


**Figure 6.2: Simple Backseat Driver:** `c_heading` 90° from `m_water_vel_dir` during simple backseat driver application

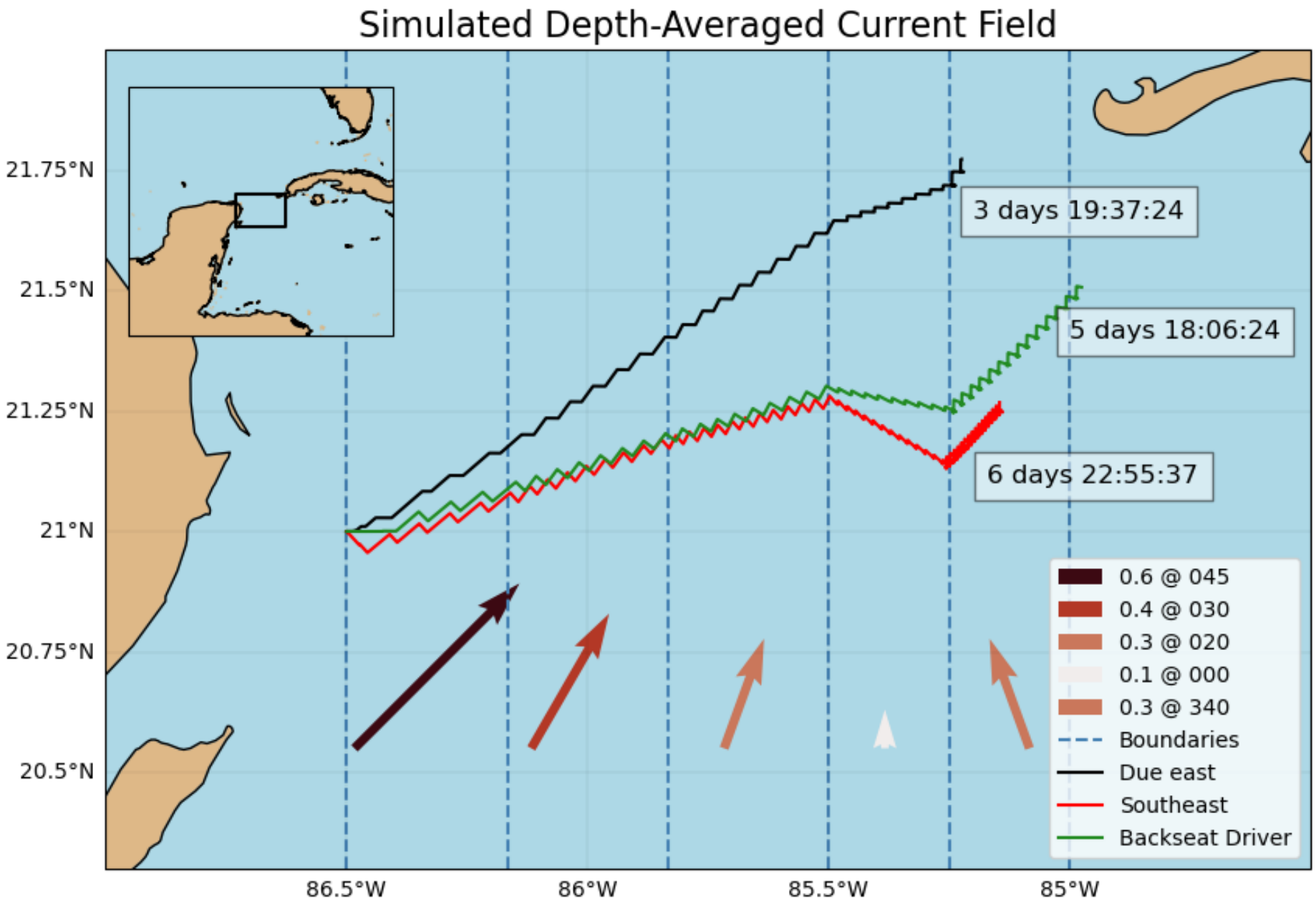


**Figure 6.3: Simple Backseat Driver Location Comparison:** Green line is location of `spectre` ran with the BSD control architecture (rest of map the same as Figure 5.4). Note green's arrival to eastern boundary faster than red's with its southeast waypoint, but with greater deflection distance northward.

# 7

# Advanced Backseat Driver Behavior Application: Commanding `c_heading`, more sensors, make a plot

**Chapter overview:**
We will ingests more sensors from the glider and do more calculations than the simple case.

1. The RasPi will be powered on for surfacing and diving only
2. The RasPi script contains a while loop to wait for multiple sensor values before continuing (in contrast to running the script despite the presence of desires values).
3. The RasPi upon receiving all sensor values required for the calculation, will calculate the most optimal heading for the prevailing current conditions, glider speed, and heading.
4. The RasPi will generate a script run-time file, used for debugging and troubleshooting before deployment.

For this application, ”most optimal” means the heading that will result with the least northward deflection distance compared to flying a heading 90° from the current.

### 7.0.1. extctl.ini

```
mp # MISSION PARAMS SECTION - u_mission_param[a-l]
# c_heading in set-he94.ma, used with b_arg: heading_value(X) 1000000
                                    # INDEX NUMBER FOR RASPI PYTHON
u_mission_param_a                   # c_heading

os # OUTPUT SENSORS SECTION
sci_generic_a nodim                 # 1

is # INPUT SENSORS SECTION
# Glider coords
#flight
m_present_time timestamp            # 2
m_heading rad                       # 3
m_avg_speed m/s                     # 4
m_water_vel_dir rad                 # 5
m_water_vel_mag m/s                 # 6
m_lat lat                           # 7
m_lon lon                           # 8

# Specify the baudrate of the uart, don't technically have to specify but to not forget
baud
9600 # default value

packet_size # don't technically have to specify but to not forget
600 # default value
```

**File 7.1: Advanced Backseat Driver:** extctl.ini - control c_heading, send more sensors (lat/lon, etc)

### 7.0.2. sample90.ma

```
behavior_name=sample
# SAMPLE90.MA (EXTCTL)

<start:b_arg>
    b_arg: sensor_type(enum)                    90  # c_extctl_on
    b_arg: sample_time_after_state_change(s)    0   # start sampling right away

    # Sampling Arguments
        b_arg: intersample_time(sec)    0
        b_arg: state_to_sample(enum)    9   # diving | on_surface
        b_arg: nth_yo_to_sample(nodim)  1   # +n: first and nth: 1,3,5
<end:b_arg>
```

**File 7.2: Advanced Backseat Driver:** sample90.ma - power the RasPi on only on surface and diving

The rest of the glider's files remain the same as in the previous chapter's basic case.

## 7.1. Advanced Application: Raspberry Pi Script

This script is long, so these are the most important parts to explain. This is best looked at as an example for you to expand upon based upon your specific use-case requirements.

### 7.1.1. extctl Python class

```
# SKIPPING IMPORTS, SEE GITHUB
timer_start = datetime.datetime.now()   # TIMER START
values_present = 3      # PROGRESS MESSAGES SENT VIA GLIDER TERMINAL
all_done = 42

class extctl(object):
    def __init__(self, tty):
        self.port = serial.Serial(tty, baudrate=baud, xonxoff=False, rtscts=False,
        dsrdtr=False, stopbits=serial.STOPBITS_ONE, parity=serial.PARITY_NONE,
        bytesize=serial.EIGHTBITS, timeout=None)

    def send(self, s):
        csum = 0
        # bs = s.encode('ascii')
        bs = s
        for c in bs:
            csum ^= c
        nmea = b'$' + bs + (b'*%02X\r\n' % csum)
        print(nmea)
        self.port.write(nmea)

    def write_SW(self, index, value):
        self.send(b'SW,%d:%f' % (index, value))

    def write_TXT(self,value):
        self.send(b'TXT,%f' % (value))

    def mode(self, mask, value):
        self.send(b'MD,%d,%d' % (mask, value))

    @staticmethod
    def read(self):
        #ser = serial.Serial(port_address, baudrate=uart_baud)
        #line = ser.readline()
        line = self.port.readline()
        return line
```

**File 7.3: Advanced Backseat Driver:** sendHeadDir_WAIT.py - extctl Python class

### 7.1.2. while loop - parse strings from glider

```
m_present_time_norm = None # 2
m_heading_rad       = None # 3
m_avg_speed         = None # 4
m_water_vel_dir_rad = None # 5
m_water_vel_mag     = None # 6
m_dr_time           = None # 7
m_lat               = None # 8
m_lon               = None # 9

port = serial.Serial(port_address, baudrate=baud)
x = extctl(port_address)

while done_with_for_loop == 0:
    print('start while')
    cur_line = str(port.readline())     # full line coming from glider
    print(f'cur_line: {cur_line}')     # view full line if viewed from RasPi
    msg = cur_line.split('*')           # Split into separate fields
    fields = msg[0].split(',')          # Split into separate fields
    # print(f'{datetime.datetime.now()}: Fields: {fields}')   # see fields not cur_line

    # number must match index number in extctl.ini
    for subscription in fields:
        if '2:' in subscription:              # index number of m_present_time_epoc
            subs = subscription.split(':')    # from TWR
            m_present_time_epoc = float(subs[1])
            m_present_time_norm = datetime.datetime.fromtimestamp(m_present_time_epoc) #
                convert to human timestamp
            print(f'found 2 = {m_present_time_norm}')

        if '3:' in subscription:              # index number of m_heading
            subs = subscription.split(':')    # from TWR
            m_heading_rad = float(subs[1])
            print(f'found 3 = {m_heading_rad}')
            m_heading_deg = math.degrees(m_heading_rad)

        if '4:' in subscription:              # index number of m_water_vel_mag
            subs = subscription.split(':')    # from TWR
            m_avg_speed = float(subs[1])
            print(f'found 4 = {m_avg_speed}')

        if '5:' in subscription:              # index number of m_avg_speed
            subs = subscription.split(':')    # from TWR
            m_water_vel_dir_rad = float(subs[1])  #
            print(f'found 5 = {m_water_vel_dir_rad}')
            m_water_vel_dir_deg = math.degrees(m_water_vel_dir_rad)

        if '6:' in subscription:              # index number of m_water_vel_mag
            subs = subscription.split(':')    # from TWR
            m_water_vel_mag = float(subs[1])  #
            print(f'found 6 = {m_water_vel_mag}')

        if '7:' in subscription:              # index number of m_water_vel_mag
            subs = subscription.split(':')    # from TWR
            m_dr_time = float(subs[1])
            print(f'found 7 = {m_dr_time}')

        if '8:' in subscription:              # index number of m_water_vel_mag
            subs = subscription.split(':')    # from TWR
            m_lat = float(subs[1])
            print(f'found 8 = {m_lat}')

        if '9:' in subscription:              # index number of m_water_vel_mag
            subs = subscription.split(':')    # from TWR
            m_lon = float(subs[1])
            print(f'found 9 = {m_lon}')
```

**File 7.4: Advanced Backseat Driver:** sendHeadDir_WAIT.py - parse strings from glider

### 7.1.3. while loop - continue when all values are present

```
    if (m_present_time_norm             != None) and \
        (m_heading_rad                  != None) and \
        (m_avg_speed                    != None) and \
        (m_water_vel_dir_rad    != None) and \
        (m_water_vel_mag                != None) and \
        (m_dr_time                              != None) and \
        (m_lat                                  != None) and \
        (m_lon                                  != None):
        #print(m_present_time_norm,m_heading_rad,m_avg_speed,m_water_vel_dir_rad,
            m_water_vel_mag,m_dr_time,m_lat,m_lon)

        print(f'{datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")}_values present, lets go
            !')
        x.write_TXT(values_present) # first progress message

        if 270 <= m_water_vel_dir_deg < 360 or 0 <= m_water_vel_dir_deg < 45:
            # use optimal heading
            new_heading_send = np.round(heading_with_min_deflection_rad,3)       # round to
                three digits #.## because we don't need all
            x.write_SW(0,new_heading_send)                                       # change
                heading on glider
            x.write_TXT(new_heading_send)                                        # print
                message in glider terminal
            print(f'{current_time}: new_heading = {new_heading_send}') # print new heading with
                 timestamp in RasPi terminal

        elif 45 <= m_water_vel_dir_deg < 90:
            # use 90 degrees from current (relative to current)
            new_heading_send = np.round(math.radians(m_water_vel_dir_deg + 90),3)
            x.write_SW(0,new_heading_send)                                       # change
                heading on glider
            x.write_TXT(new_heading_send)                                        # print
                message in glider terminal
            print(f'{current_time}: new_heading = {new_heading_send}') # print new heading with
                 timestamp in RasPi terminal

        elif 90 <= m_water_vel_dir_deg < 270:
            # use 90 degrees (due east)
            new_heading_send = math.radians(90)
            x.write_SW(0,new_heading_send)                                       # change
                heading on glider
            x.write_TXT(new_heading_send)                                        # print
                message in glider terminal
            print(f'{current_time}: new_heading = {new_heading_send}') # print new heading with
                 timestamp in RasPi terminal

        else:
            x.write_TXT(er_m_water_vel_dir)
            pass

        done_with_for_loop = 1 # true, finish while loop and done with script

        # MORE CODE USING THE VALUES IF DESIRED

# after the while loop
x.write_TXT(all_done)

# TIMER END AND CALCULATE SCRIPT RUN TIME
timer_end = datetime.datetime.now()
script_time = timer_end - timer_start
elapsed_seconds = str(round(script_time.total_seconds(),2))

# PUT SCRIPT RUN TIME IN A FILE
with open('script_times.txt','a') as file:
    file.write(f'{elapsed_seconds}\n')

print(f'script is done! ({elapsed_seconds}s)')
```

**File 7.5: Advanced Backseat Driver:** sendHeadDir_WAIT.py - continue when all values are present

### 7.1.4. crontab

```
# to open crontab thru the RasPi's terminal (only open via terminal, not the file on the RasPi)
sudo crontab -e

# to run once every minute, as the script takes between 18-48 seconds
* * * * * sh /home/rutgers/backseat_driver/launcher.sh >> /home/rutgers/logs/cronlog 2>&1

# to display updates written to cronlog while crontab is running
# tail -f cronlog
```

**File 7.6: Advanced Backseat Driver:** Setup the Crontab

As we'll see later, this script takes 45 seconds max to run; thus, the crontab runs every minute. Manually run your script a few times on the RasPi (sudo python3 [script_name.py]) before using the crontab so you can see how long it takes.

## 7.2. Advanced Backseat Driver Output Plot

The Python file for this outputs two files: 1) a plot (figure 7.2) and 2) a script_times.txt that contains the run-time of the script. In a real deployment this type of plot would not be generated onboard the glider using the EC - it would be generated shore-side. This was done onboard the EC to 1) show the flexibility and computing improvement of adding an EC to the glider system, and 2) allow for ease of troubleshooting mid-mission.

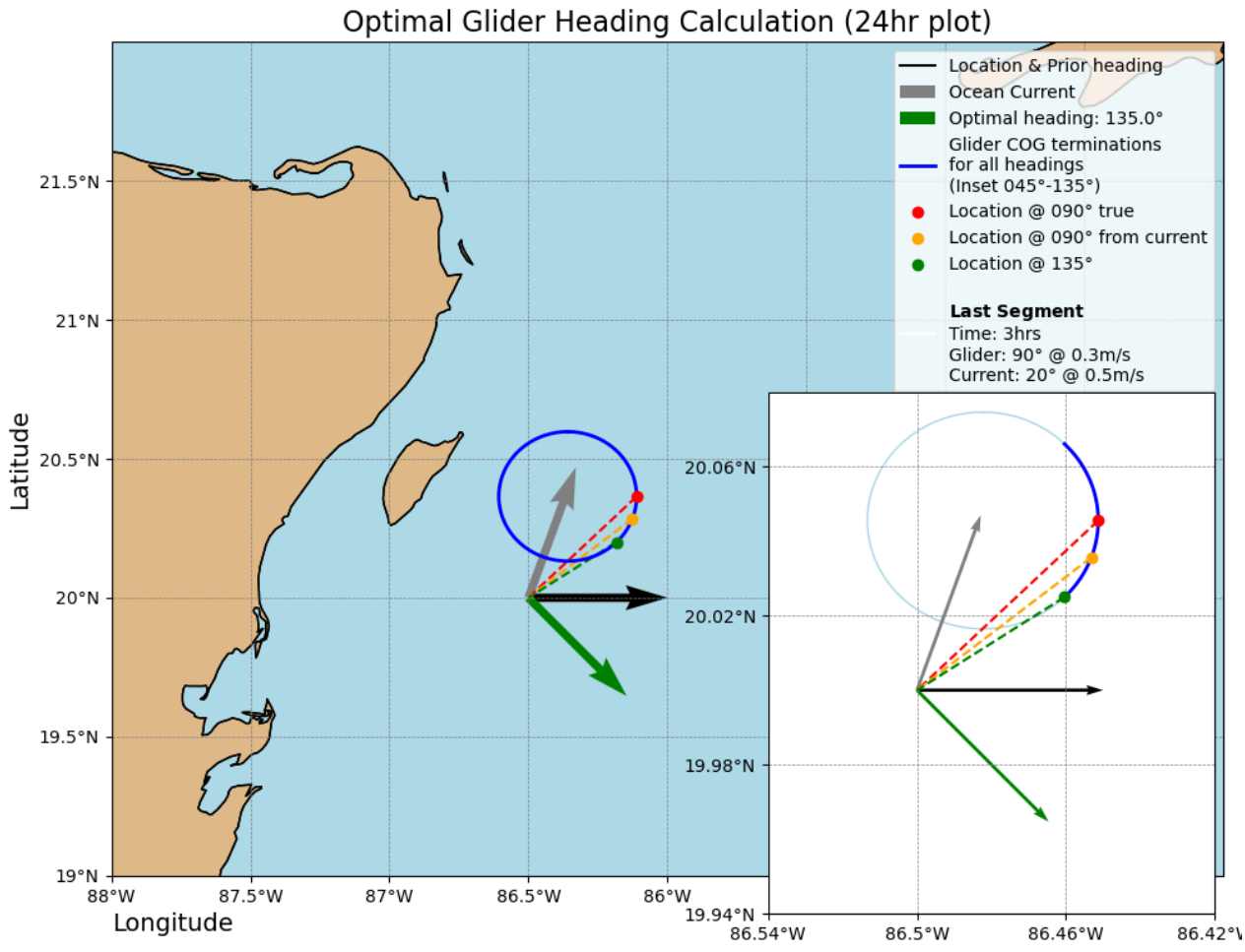


**Figure 7.1: Advanced Backseat Driver:** External controller output plot

If you are planning on making a diagnostic plot from values sent ashore from the glider, especially if using the EC to read a sensor's output (like an AD2CP) and then generate some intermediary file for mid-mission data, I recommend doing it all onboard the EC while simulating. You can immediately know if any part breaks (EC reading the sensor data or your diagnostic plot's code) without having to wait for the mission to end, going through SFMC, or troubleshooting during a real deployment in the ocean.

## 7.3. External Controller Python Script Run-Time Considerations

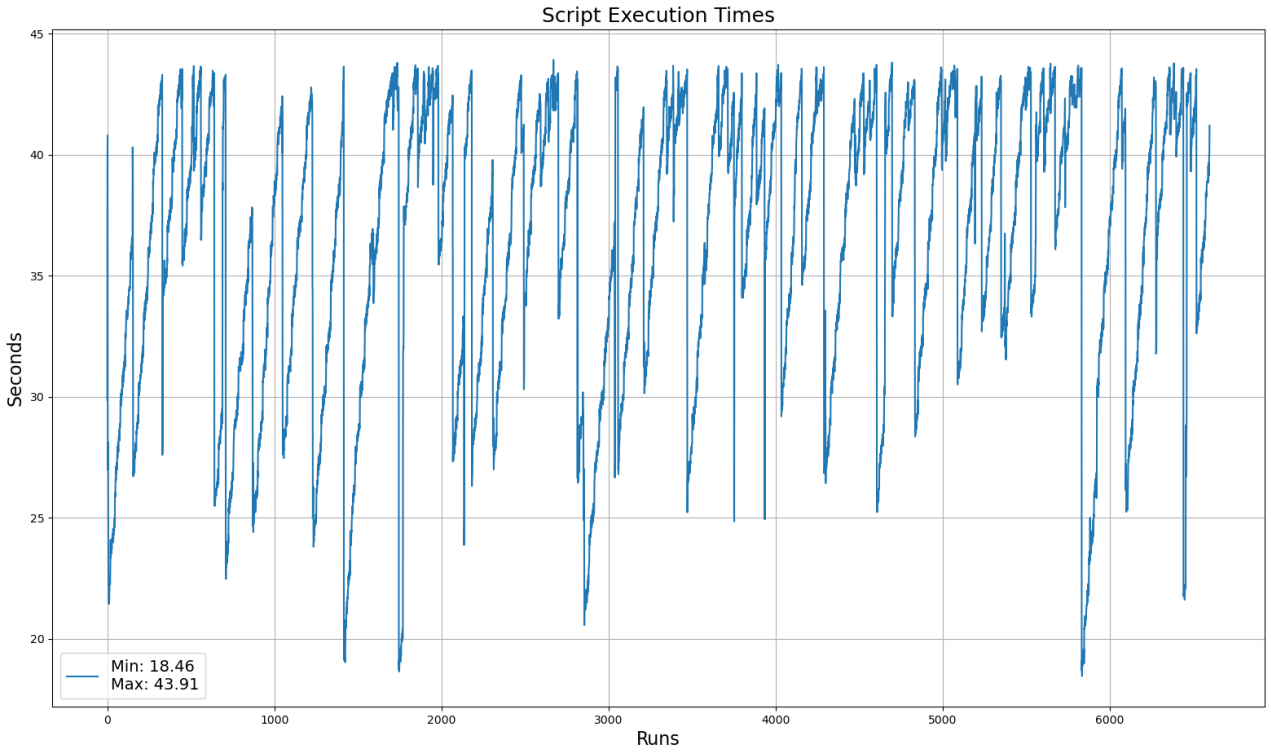


**Figure 7.2: Advanced Backseat Driver:** Script Run Times

The EC script has a highly variable run-time, ranging from 18.5 seconds to 44 seconds. As the Python script itself can run in under 18 seconds, the majority of this time is waiting for all sensor values to be sent before continuing.

If one starts requesting for many sensor values, this time will only increase and should be tested before any deployment. Additionally, through pure speculation from looking at the output from the glider for many hours, certain sensors are updated more frequently while others come through often in some states and not others.

For example,`m_avg_speed` shows up basically every cycle, while values like `m_water_vel_dir` show up more often while surfacing and the first five or so minutes of diving behavior before it seems to taper off. Thus, if you are interested in one specific sensor value (like I was with `m_water_vel_dir`) test things out and get a "feel" for how the sensor value will appear. Basically, sit and watch for it to see how it comes in, or quantify

It bears repeating that "there is not currently a way to have the BSD send every sensor requested each cycle. From Teledyne, "this was done to reduce the number of calls over the clothesline. When the SC attempts to send a $SD message the proglet will check which sensors were updated and only request those that have been updated. Only sensors that were updated will be sent in the $SD. A way of keeping every thing in sync on the EC side of the system would be to have a accessible struct that is continuously updated based on inputs over the UART. This will help keep a EC side memory of what all the values are."

## 7.4. More Glider Simulator Limitations

During testing for this chapter I ran into an issue with how the glider simulator itself calculates water velocities. After communicating with Teledyne (much of the following text in this section comes from communication with Teledyne, see email in GitHub files) and troubleshooting myself. An overview for how the glider calculates water velocities is as follows:

**Non-simulation glider (real glider and in the ocean):** Water velocities get recalculated whenever m_gps_x_lmc is updated. In a normal mission this will occur when the glider surfaces and gets a GPS fix.

**Simulation glider (shoebox, in a lab):** The `s_water_speed` and `s_water_direction` are used in the dead reckoning while in simulation mode and act as the true water velocity for how the simulated glider positions will change while underway. `m_water_vel_dir` and `m_water_vel_mag` however are used for heading correction and every time a surfacing occurs will be calculated, see the method below. Glider tries goes underway running calculations with `m_water_vel_dir` and `m_water_vel_mag`. When it surfaces it compares dead reckoned positions to actual position to calculate water velocities.

- `m_dr_x_ini_err = m_gps_fix_x_lmc - m_dr_surf_x_lmc`
- `m_dr_y_ini_err = m_gps_fix_y_lmc - m_dr_surf_y_lmc`
- `delta_vx = m_dr_x_ini_err / m_dr_time`
- `delta_vy = m_dr_y_ini_err / m_dr_time`
- `m_water_vx = delta_vx + x_prior_seg_water_vx`
- `m_water_vy = delta_vx + x_prior_seg_water_vy`

I've bolded the end of the last two lines where `m_water_vel_dir` and `m_water_vel_mag` have memory of the prior segments. Because of this it was recommended to reset the following values (instead of only `m_water_vel_dir` and `m_water_vel_mag`):

- `x_prior_seg_water_vx`
- `x_prior_seg_water_vy`
- `m_water_vx`
- `m_water_vy`
- `m_water_vel_mag`
- `m_water_vel_dir`

I ran a few missions attempting to use an SFMC .xml script to `!put` these sensor values to zero before resuming the mission (immediately before diving), to no success. Putting them to zero seemed to have no effect on the calculations onboard the glider. As this was not "mission critical" for my thesis work to continue, no further development into solving this issue was pursued. A potential way forward, that also avoids at surface calculations overwriting any of the puts would be for the EC to update these sensor values. But, as that would basically defeat the purpose of the EC calculating from these values, it has little application in this specific example, with potential uses in other EC applications.

Furthermore, `m_dr_time` does not seem to work in the simulator. Again, I accepted this and moved on to further work.

# 8

# Sending Files To/From External Controller

**Chapter overview:**

1. The extctl Python class used
2. How to use the extctl functions
3. The Python code required

The required functions for this operation are much more involved. As this is a Python heavy chapter, please be reminded that all files used throughout this manual are on GitHub. Click **here** to open.

For this chapter, the RasPi will be powered on the entire time. We can use the external controller to send and/or receive a file anytime in the mission, so we begin the mission, let the glider dive, then work through examples in this chapter

## 8.1. Baud Rate and Chunk Size

The `extctl.ini` file specifies the baud rate and chunk size the science computer and external controller use to communicate. Changing these will change how long it takes to transfer files back and forth.

**Baud Rate:** The default value is 9600, and the max recommended in Teledyne's Backseat Driver Software Guide is 115200 for the most stable communications. This can be pushed, see following table.

**Chunk size:** This sets the max size of data sent back and forth (in bytes per packet). The default value is 500, and the max value in Teledyne's Backseat Driver Software Guide is 730. However, if sending a comparatively big file it may be desired to push this some. Do so requires a faster baud rate, see following table.

**Table 8.1:** Recommended Baud Rates and Chunk Sizes

| Baud Rate | Chunk Size | Details |
|---|---|---|
| 9600 | 500 | Default communications that will be slower than other settings |
| 115200 | 730 | Fastest Teledyne recommended settings |
| 230400 | 1200 | Upward capability possible from the system from our testing |

While it's certainly possible to use values beyond Teledyne's recommended fastest communication settings, extensive lab-based testing should be conducted to ensure stability of the system. The manufacturer recommends the settings for a reason - they don't want a glider to have an issue in the ocean - so properly test all systems in the lab before a deployment. This should be done anyways, but *especially* when attempting to push settings towards there limits.

### 8.1.1. extctl Python class: sending/receiving files

```
# define locations and generics
uart_baud = 9600
send_chunk_size = 600
port_address = '/dev/ttyUSB0' # CHANGE x TO PORT USED

# Set path to save processed data files to give back to glider
to_dir='/home/rutgers/backseat_driver/to_glider/'

# Set path for .tbd files from glider
from_dir = '/home/rutgers/backseat_driver/from_glider/'

# Set path for completed files
pros_dir = '/home/rutgers/backseat_driver/files_processed/'

class extctl(object):
    def __init__(self, tty):
        self.port = serial.Serial(tty, baudrate=uart_baud, stopbits=serial.STOPBITS_ONE, parity
            =serial.PARITY_NONE,
        bytesize=serial.EIGHTBITS)

    def send(self, s):
        csum = 0
        # bs = s.encode('ascii')
        bs = s
        for c in bs:
            csum ^= c
        nmea = b'$' + bs + (b'*%02X\r\n' % csum)
        print("Sending: ", nmea)
        self.port.write(nmea)

    def write(self, index, value):
        self.send(b'SW,%d:%f' % (index, value))

    def mode(self, mask, value):
        self.send(b'MD,%d,%d' % (mask, value))

    def read(self):
        return self.port.readline()

    @staticmethod
    def send_file_data(data):
        # Encode the data in b64
        output_string = b'FO,' + base64.b64encode(data)
        x.send(output_string)

    def file_read(self, filename, sci_dir, data_type):
        # Send open read message to glider and open destination file
        command = b'FR,' + bytes(filename, 'utf-8') + b',' + bytes(sci_dir, 'utf-8')
        self.send(command)
        print(f'opening file {filename}')
        if data_type:
            file = open(f'/home/rutgers/backseat_driver/from_glider/{filename}', 'wb')
        else:
            file = open(f'/home/rutgers/backseat_driver/from_glider/{filename}', 'w')
        self.recieve_file_data(file, data_type)

    def GO(self):
        self.send(b'GO')

    # This function sends the command for the extctl proglet to list files in 'directory'
        matching 'wild_card'
    def list_directory(self, directory, wild_card):
        self.send(b'LS,' + bytes(directory, 'utf-8') + b',' + bytes(wild_card, 'utf-8'))

    # File write but with a specifier for
    def file_write_new(self, source_file, directory):
        # Send command to open a file in directory
        self.send(b'FW,' + bytes(source_file, 'utf-8') + b',' + bytes(directory, 'utf-8'))
```

**File 8.1: Advanced Backseat Driver:** sendHeadDir_WAIT.py - continue when all values are present

### 8.1.2. extctl Python class: sending/receiving files

```
    # Runs full loop to send a file
    @staticmethod
    def send_file(uart_port, file_path, dest_name, dest_dir, byte_bool):
        # Bool for if need confirmation message
        x.file_write_new(dest_name, dest_dir)
        # open the specified file.
        with open(file_path, 'rb') as file_object:
            while True:
                 # This for binary based files
                if byte_bool:
                    data = file_object.read(send_chunk_size)
                # This for text data
                else:
                    data = bytes(file_object.read(send_chunk_size), 'utf-8')
                # End of file, break loop
                if not data:
                    print("file send break")
                    break
                # Send data and set waiting bool
                x.send_file_data(data)
                waiting = 1
                # Read the uart line until you read get the confirmation message from the
                    proglet
                while waiting:
                    check_line = uart_port.readline()
                    check_line = check_line.decode('utf-8')
                    check_line = check_line.split('*')[0]
                    # Recieved ok to send next chunk, otherwise keep checking for it
                    if check_line == '$NEXT':
                        print('Got the next')
                        waiting = 0

    def recieve_file_data(self, file, data_type):
        line = self.read()
        print(line)
        # print('chunk sent')
        if line.startswith(b'$ER'):
            return
        if line.startswith(b'$FI'):
            if line.startswith(b'$FI,'):
                line = line.split(b',')
                line = line[1]
                # print(f'line: {line}')
                content = line.split(b'*')[0]
                # print(f'content: {content}')
                if data_type == 'bin':
                    content = base64.decodebytes(content) # binary data
                else:
                    # original
                    # content = base64.b64decode(content).decode('utf-8') # other data

                    # newtry
                    content = base64.decodebytes(content)
                    print(f'content = {content}')
                    # content = base64.b64decode(content) # other data

                file.write(content)
                self.GO()
                self.recieve_file_data(file,data_type)
            else:
                print('Reached EOF, closing file and exiting')
                file.close()
                return
        else:
            self.recieve_file_data(file,data_type)
```

**File 8.2: Advanced Backseat Driver:** sendHeadDir_WAIT.py - continue when all values are present

### 8.1.3. Python implementation for sending/receiving files

```
serial.Serial(port_address, baudrate=uart_baud).close()
x = extctl(port_address)
port = serial.Serial(port_address, baudrate=uart_baud)

test=0
count=0

# ask SC to list files in a directory, sends names to EC)
# init file_list, place to put the file names we get back from the list_directory command
file_list=[]
x.list_directory("logs",".tbd")

print('before␣while')
while test == 0:
    cur_line = str(port.readline())
    # print(cur_line) results in
        # ["b'$DIR","c:/logs/00100000.tbd', 'c:/logs/00110000.tbd', 'c:/logs/00120000.tbd', '']
    # Split into separate fields
    msg = cur_line.split('*')
    fields = msg[0].split(',')
    # print(fields) results in
        # ["b'$DIR", 'c:/logs/00100000.tbd', 'c:/logs/00110000.tbd', 'c:/logs/00120000.tbd',
            '']

    for subscription in fields:
        if 'c:' in subscription: # get files from the list_dir function
            subs=subscription.split(':')
            pfile=subs[1]
            # print(subs)
            file_list.append(subscription)
            print(f'file_list:␣{file_list}')

    if (file_list != None):
        # print(f'file_list -1: {file_list[-1]}')           # for viewing
        # tbd_file_name = os.path.basename(file_list[-1])   # use -1 to get the most recent
            file
        tbd_file_name = 'ru34-2024-089-0-22.tbd' # or ^, but easy to specify for testing

        # Send file from SC to EC
        x.file_read(tbd_file_name,"logs",1)
        # add path to most recent filename
        print('file␣transferred,␣trying␣to␣process␣now...')
        file = '/home/rutgers/backseat_driver/from_glider/'+tbd_file_name

        # the .tbd file is now on the glider and can be used

        # send file from EC to SC
        # to get a tbd (or other file) on the glider for testing mid-mission, use these lines
        # saves having to end the mission, dockzr etc
        # x.send_file(port, "/home/rutgers/backseat_driver/"+str(name), str(name), "logs", 1)
        # send_file(uart_port, file_path, dest_name, dest_dir, byte_bool):
    test = 1
```

**File 8.3: Advanced Backseat Driver:** sendHeadDir_WAIT.py - continue when all values are present

# 9

# Science Sensor Integration, Testing, and Data File Management

**Chapter Overview:**
In this chapter we will have the RasPi read values from a CTD41CP sensor. There are three ways we will accomplish this, each with their own example:

**Example A:** Plugging a real CTD41CP sensor into the science port's of the simulator. Because this is rather simple to execute, we also append these values to a file on the RasPi so it can access and/or make decisions using the prior x amount of information.

**Example B:** Simulating a CTD41CP sensor using a .py script. Ran on a laptop that sends serial data through the same type of cable as the real sensor, we decide what ocean the glider is "seeing" mid-mission.

**Example C:** Sending the latest `.tbd` file from the science computer and processing the values therein, ready to make a decision using them.

## 9.1. Example A: Using an Actual CTD41CP Sensor

This is predicated upon having an actual sensor and it's connecting wire in-hand. If you don't, use all the glider files for Example A and go to Example B, simulating the data.

### 9.1.1. extctl.ini

```
mp # MISSION PARAMS SECTION - u_mission_param[a-l]
# no mission params

os # OUTPUT SENSORS SECTION
sci_generic_a nodim             # 0

is # INPUT SENSORS SECTION
# Glider coords
#flight
m_present_time timestamp        # 1

#science values
sci_water_cond S/m              # 2
sci_water_temp degC             # 3
sci_water_pressure bar          # 4

baud
9600 # default value
packet_size
600 # default value
```

**File 9.1: Example A:** extctl.ini - send CTD data

### 9.1.2. proglets.dat

```
proglet = extctl
    uart       = j5         # place it's plugged in
    bit        = 0          # power, battery
    start_snsr = c_extctl_on(sec)

proglet = ctd41cp
    uart       = j4         # connector
    bit        = 0          # power, battery
    start_snsr = c_profile_on(sec)
    simulator  = ctd41cp_sim
```

**File 9.2: Example A:** proglets.dat - include CTD41CP proglet

The only addition from our usual is the CTD41CP proglet found in `motherproglets.dat`

### 9.1.3. tbdlist.dat

```
# Basic Data
  sci_m_present_time

# CTD41CP Sea-bird  CTD(SBE-41)
# proglet = ctd41cp SIMULATING THE DATA THIS SENSOR SENDS
# ----------------------------
  sci_water_cond        30   15   -1  3
  sci_water_pressure    30   15   -1  3
  sci_water_temp        30   15   -1  3
```

**File 9.3: Example A:** extctl.ini - send CTD data

Now that there is science data, we can include it in the `tbdlist.dat`. In Example C we will send the most recent `.tbb` (or other) file from the science computer to the external controller.

### 9.1.4. sample01.ma

```
behavior_name=sample

# sample01.ma (CTD)

<start:b_arg>
    b_arg: sensor_type(enum)                    1   # c_profile_on, is CTD
    b_arg: sample_time_after_state_change(s)    0   # start sampling right away

    # Sampling Arguments
        b_arg: intersample_time(sec)    1   # if = 0 then fast as posible, > 0 secs
        b_arg: state_to_sample(enum)    15  # diving|hovering|climbing,  all
        b_arg: nth_yo_to_sample(nodim)  1   # after the first yo, sample only every nth to
<end:b_arg>
```

**File 9.4: Example A:** tbdlist.dat - include CTD science data

The generic sample file for the CTD41CP that has it always on (during all behaviors). It's important to note that if ran in air there will be no salinity value, which, while it looks weird, is fine.

### 9.1.5. science.mi

```
# FULL FILE ON GITHUB
# SAMPLE : CTD41CP, THE ADDED SENSOR
behavior: sample
    b_arg: args_from_file(enum)     01 # 01 CTD
```

**File 9.5: Example A:** science.mi - calls sample01.ma as well

The mission file for this example is almost the same for Example B and Example C, as it utilizes both the flight computer's and science computer's CTD41CP proglet and `sample01.ma` files.

## 9.2. Example B: Simulating a CTD41CP Sensor

In this example, we fully simulate the sensor and send data to the science computers port via an RS232 USB adapter and a Python .py on a laptop. To test this, and verify proper operation, we use all the same files as Example A, plus the additional Python .py script to simulate ocean variables. We don't run the Python .py script that simulates the ocean on the external controller so we can observe and verify proper BSDA behavior when reading specific science sensor values. We will simulate our ocean two ways, by sending data using:

**simCTD_synthetic.py:** Creating fake science data that roughly approximates an ocean, stored in a Pandas DataFrame, then sending values therefrom.

**simCTD_file.py:** Reading `.ebd` files containing real ocean variable data from an oceangoing glider mission.

### 9.2.1. Why Simulate a Sensor?

Depending on the type of BSDA application it may not be possible to properly test the application without reading a specific type of ocean variables and/or ocean conditions from the glider itself. In a lab setting this can often be impossible using an actual sensor because the glider is not actually in the ocean. Thus, to allow certain types of external controller application tests we must simulate the incoming science sensor data.

### 9.2.2. simCTD_synthetic.py

Figure 9.1 is just one example of a type of data possible can send. Using simCTD_synthetic.py, you could send any values desired to the science computer and thus external processor.

1. First we define the ctd41cp Python class with the required commands.
2. Then we approximate changing ocean conditions following a typical depth profile. Figure 9.1 (left) shows the three values that are sent to the glider. The rate of change of temperature and salinity is approximated with cosine functions, while pressure assumes a moment of initialization at the surface before a linear downward trajectory. In this, we assume the change is over ten minutes, resulting in 600 data values, as seen on the left.
3. Lastly, we open the computer's COM port and send the simulated data to the glider's science computer at 1hz through the USB to RS232 adapter (using

Note that all variables must be the same length as one another, and the length changes the overall time the script will send the variables to the science computer. Furthermore, while the script sends data at 1hz (simulating the CTD41CP sensor) the external controller won't read the data that fast. Using a crontab running Example A's script every 5 seconds the external controller only read in 105 values. The values were correct, just in less total number (figure 9.2, next page)

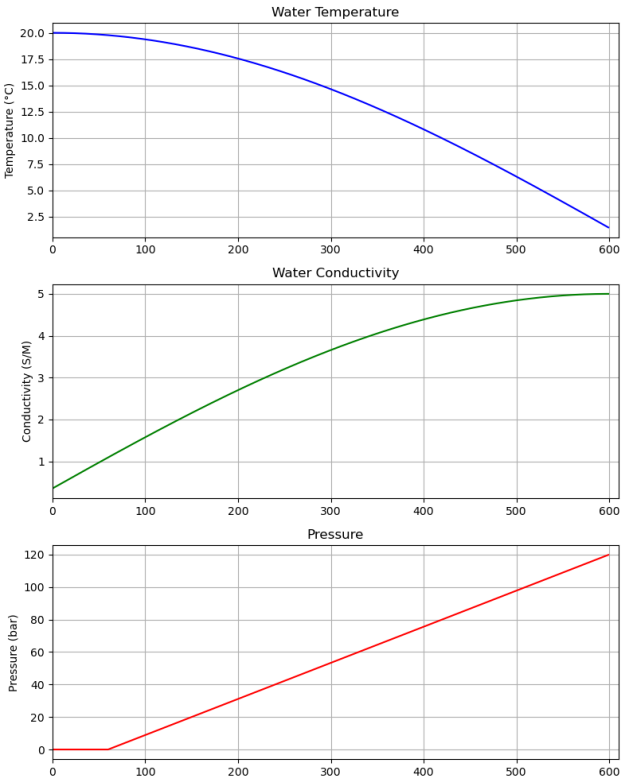


**Figure 9.1: Example B:** simCTD_synthetic.py's simulated CTD data

Once simCTD_synthetic.py is ready to run, you have a connector made to connect your laptop to the science computer (next section), and you have the external controller ready from Example A:

1. Start the glider's `science.mi` mission. Approximately five or so seconds later, approximately once you see the mission stack;
2. Start sending the simulated science data with the Python script, then;
3. Use the crontab, or run one iteration of the file, to run Example A (or another program) on the external controller.

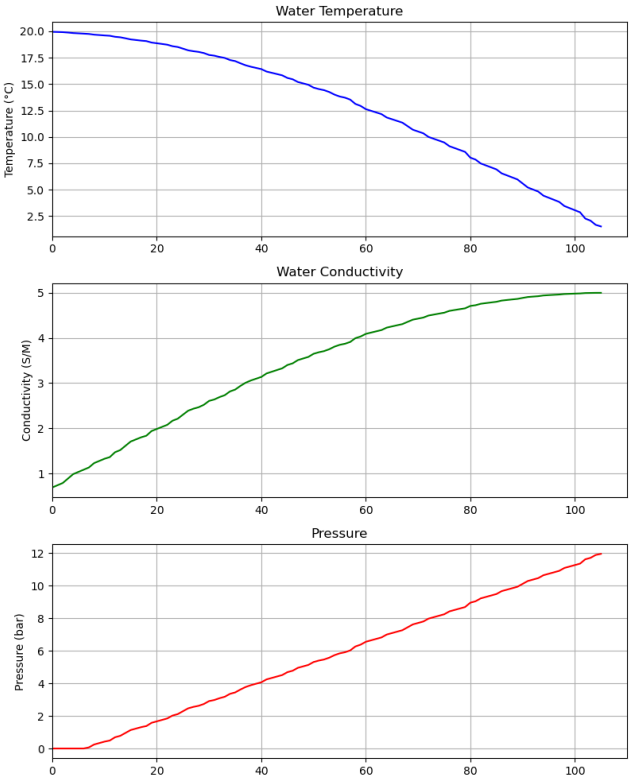


**Figure 9.2: Example B:** simCTD_synthetic.py's simulated CTD data as seen from the external controller

### 9.2.3. simCTD_file.py:

This example is from Ella Wawrzynek who first used this method using `.ebd` files. This Python program:

1. Defines the ctd41cp Python class
2. Decodes `.ebd` files, from a real glider mission
3. Sends the data as before

See file on GitHub or on later page of this manual.

### 9.2.4. Making the Connector to Simulate a Science Sensor

For this connector we need a USB to RS232 adapter and a 10pin male Molex, like before. We will be connecting the two, allowing the connection of a laptop's USB to the science computer.

### 9.2.5. Example B: simCTD_synthetic.py

```
class ctd41cp(object):
    def __init__(self, port):
        self.port = serial.Serial(port, baudrate=baud, stopbits=serial.STOPBITS_ONE, parity=
            serial.PARITY_NONE,
        timeout=timeout, bytesize=serial.EIGHTBITS)

    def write(self, s,count):
        print('Sending: ', s,'| count is',count)
        self.port.write(s)

    def read(self):
        return self.port.readline()

min_to_run  = 10
time_to_run = np.arange(0,min_to_run*60,1)
length      = len(time_to_run)

# warmer surface, decrease with depth (time)
surface_temp = [20]
iterate_temp = np.cos(np.arange(0,1.5,(1.5/length)))
water_temp   = surface_temp * iterate_temp

# fresher surface, increase with depth
depth_salt   = [5]
iterate_salt = np.cos(np.arange(-1.5,0,(1.5/length)))
water_cond   = depth_salt * iterate_salt

pressure_pause = [0]*60 # surface time before diving
pressure_data  = (np.arange(0,120,(120/(length-60))))
pressure       = pressure_pause.__iadd__(pressure_data)

sci_data = pd.DataFrame(index=time_to_run)
sci_data['cond']     = water_cond
sci_data['temp']     = water_temp
sci_data['pressure'] = pressure
indx = len(sci_data)-1

port = 'COM5' # USB port
baud = 9600
timeout = 0 # Non-blocking read

print('Opening port...')
u = ctd41cp(port)

print('Sending CTD data to simulator...')
index_len = len(sci_data)-1 #-1 cuz zero starting list
count = 0
while indx > 0:
    round = 4
    cond = np.round(sci_data.cond[count],round)
    temp = np.round(sci_data.temp[count],round)
    pres = np.round(sci_data.pressure[count],round)
    write = (f'{pres},{temp},{cond}\r\n')
    encoded_bytes = write.encode('utf-8')
    u.write(encoded_bytes,count)

    time.sleep(1) # approx 1 Hz data rate
    indx  = index_len - 1
    count = count + 1

u.port.close() # have to close or can't reopen and run again
print('all data sent')
```

**File 9.6: Example B:** simCTD_synthetic.py

Used for simulated CTD values.

### 9.2.6. Example B: simCTD_file.py

```
port = 'COM5' # USB port
baud = 9600
timeout = 0 # Non-blocking read

class ctd41cp(object):
    def __init__(self, port):
        self.port = serial.Serial(port, baudrate=baud, stopbits=serial.STOPBITS_ONE, parity=
            serial.PARITY_NONE,
        timeout=timeout, bytesize=serial.EIGHTBITS)

    def write(self, s):
        print("Sending: ", s)
        self.port.write(s)

    def read(self):
        return self.port.readline()

def get_tCTDxy_data_array(files,comp=False):
    xbd=dbdreader.MultiDBD(files,complemented_files_only=comp,complement_files=comp)
    t_ctd,C,T,P=xbd.get_CTD_sync()
    tCTDxy_data = np.array([t_ctd,C,T,P])
    return tCTDxy_data

def get_data_array(files,comp=False):
    xbd=dbdreader.MultiDBD(files,complemented_files_only=comp,complement_files=comp)
    raw_array = xbd.get("sci_water_cond","sci_water_temp", "sci_water_pressure",return_nans=
        True)
    return raw_array

# %% Get CTD data from previous ebd files

datapath = r''
datafiles = [datapath+"00810000.ebd",datapath+"00820000.ebd",datapath+"00910002.ebd"]

print('Decoding CTD data...')
ctd_data = get_tCTDxy_data_array(datafiles)
#ctd_data_raw = get_data_array(datafiles)

# %% Send CTD data over uart line

print('Opening port...')
u = ctd41cp(port)
# u.write(b'<!--start logging-->\r\n') # This line appears to be unnecessary

print('Sending CTD data to simulator...')
indx = len(ctd_data[0])-1
while indx > 0:
    u.write(b'%f, %f, %f\r\n' %(ctd_data[1][0],ctd_data[2][0],ctd_data[3][0]))
    time.sleep(1) # approx 1 Hz data rate
    indx = indx - 1
```

**File 9.7: Example B:** simCTD_file.py

Used for pulling CTD values from a file.

## 9.3. Example C: Retrieving and Processing the Most Recent .tbd File

This process can be used with any file on the science computer, but here we will use a .tbd file as we generated one in Example A. If you have a specific sensor file in mind, use it instead.

Once the file has been sent to the external controller from the science computer there are a few ways to process the .tbd file. This example shows the same method to send the file as in Chapter 8, with the addition of:

1. Processing the .tbd file into a Pandas DataFrame using numerous functions
2. Saving the Pandas DataFrame containing all the file data as a netCDF.nc
3. Sending the processed .nc file from the external controller to the science computer

### 9.3.1. Processing .tbd files onboard the EC

```
# we save the file as in Chapter 8 (all files for this example are on GitHub)
# this is still within the while loop

file = '/home/rutgers/backseat_driver/from_glider/'+tbd_file_name

if (os.path.isfile(file)>0):
    # must have the cac files for the glider on the RP
    tbd,tbd_meta = dbd_to_dict(file,'rutgers-cac/')

    # imports the data into Pandas DataFrame using proper time for type of file
    tbd_data = pd.DataFrame.from_dict(tbd, orient='columns', dtype=None, columns=None).
        set_index('sci_m_present_time')

    # cleans up the DataFrame time indices.
    tbd_data.index = pd.to_datetime(tbd_data.index,unit='s').round(freq='S')

    # make xarray dataset
    tbd_xr = xr.Dataset.from_dataframe(tbd_data)
    xr_file_name = f'{tbd_file_name}.nc'

    # Save the Dataset to a NetCDF file
    tbd_xr.to_netcdf(to_dir+xr_file_name)

    ncname = os.path.basename(to_dir+xr_file_name)

    #send nc from pi to glider
    x.send_file(port, "/home/rutgers/backseat_driver/to_glider/"+str(ncname), str(ncname), "
        logs", 1)
    #once sent to glider, move processed nc to files_processed folder
    # os.remove(file)

elif (os.stat(tbd_file_name).st_size==0):
    print(tbd_file_name, 'created, but data did NOT write to file.')
    os.remove(tbd_file_name)
elif not os.path.isfile(tbd_file_name):
    print('Failed to create ',tbd_file_name)
    os.remove(tbd_file_name)
    print('Finished')
else:
    print('File does not exist.')
```

**File 9.8: Example C:** Processing .tbd files onboard the EC