SPACESUIT user manual

Edward Keyes <edward.keyes@astra.com>
Last modified 2023-01-18

Introduction

SPACESUIT is the desktop tool that we use to speak to the new family of Apollo thrusters over their binary remote command interface. SPACE is the “Serial Protocol for ACE”, and SUIT stands for “Scriptable User Interface Tool”.

It is a Python command-line application, which can be run in several ways:

- It can send a single command to the thruster from the command line, possibly as part of a larger shell script.
- It can execute a series of commands from a script file. Any commands executed in a session are automatically logged, which can be edited and replayed as a script file.

- It can run interactively, accepting typed commands one by one and printing the responses from the thruster.
- It can be imported as a library to use in your own Python scripts to send remote commands and retrieve telemetry according to more complex control logic.

This document explains how to use it. Please feel free to contact Ed if you run into any trouble or have additional questions or suggestions for improvement.

For more information, also see the Communication protocol design doc for the SPACE protocol itself, and the Remote command set design doc for the definition of all of the specific ACE remote commands and defined telemetry fields.

Installation

These instructions assume you are running Linux, and probably a recent Ubuntu version such as 20.04. Other environments are possible, since Python is pretty cross-platform, but it'll be up to you to adapt the specifics for your setup.

The tool is part of the flight repo on Apollo's GitHub, the same repo used for PPU firmware builds. Follow the instructions there to clone it into a directory on your PC. Specifically, the ---spacesuit tool is in the `utils/space` subdirectory of the repo.

The tool is written for **Python v3.8** or higher. This should be the default system Python version on Ubuntu 20.04, but you may need to manually install a newer Python version on other systems, either as the OS Python version or through a local virtual environment.

It also requires the PySerial library to access the USB serial ports, which is not part of the default system install. To get it, you'll also want to have the `pip3` package manager:

```
$ sudo apt update
$ sudo apt install python3-pip
$ pip3 install -r flight/utils/requirements.txt
```

(Replace the file path to `requirements.txt` appropriately for your local repo.)

To test whether everything is working, try running `spacesuit.py` from the `utils/space` directory. Without any parameters, it will just print some brief command-line help:

```
flight/utils/space$ ./spacesuit.py
usage: spacesuit.py [-h] [--port PORT] [--address ADDRESS]
    [--thruster {ace_radhard,ace_max}] [--logname LOGNAME]
    [--macro_file MACRO_FILE] [--debug]
    (--command COMMAND | --script SCRIPT | --interactive)
```

```
spacesuit.py: error: one of the arguments --command/--cmd/-c
--script/--scr/-s --interactive/--int/-i is required
```

If you have several Python versions installed, you might need to specify the exact Python interpreter to use:

```
flight/utils/space$ python3.8 spacesuit.py
```

Serial ports

The ACE PPU has three different serial ports, with specific designated uses:

- **UART 0**: This is the **debug port**, used to flash new applications via the bootloader and provide access to a low-level console menu. It is TTL-level, 115200 baud at 8-N-1, and is typically accessed via a FTDI serial-to-USB adapter cable from a 6-pin header on the MCU coupon board, a 3-pin header on the Vorago eval board, or a couple of designated pins on the integrated PPU's feed-system connector.
- **UART 1**: This is the first RS-422 / RS-485 port, which is used for the **SPACE remote protocol**. It also runs at 115200, 8-N-1, and is typically accessed using a RS-422 USB adapter module from the first RS-422 port on the MCU coupon, or the designated pins of the PPU main external connector. On a Vorago eval board or with an MCU coupon interposer board, it can also be accessed via pins PF12 (MCU TX / PC RX) and PF13 (MCU RX / PC TX) with a TTL-to-USB FTDI adapter cable.
- **UART 2**: This is the second RS-422 / RS-485 port, which by default also runs the SPACE remote protocol. Sometimes it is switched to a **high-speed logging protocol**, which currently runs at 1.0 Mbaud, 8-N-1, though this speed might be changed in the future. It is typically accessed using a RS-422 USB adapter module from the second RS-422 port on the MCU coupon, or the designated pins of the PPU main external connector. On a Vorago eval board or with an MCU coupon interposer board, it can also be accessed via pins PF8 (MCU TX / PC RX) and PF9 (MCU RX / PC TX), although note that it is output-only, so the RX pin isn't really used for anything.

On the PC side, the mapping of these serial ports to a given Linux device is somewhat arbitrary and depends on the order in which the cables are plugged into the PC or otherwise powered up. Typically the ports will appear as `/dev/ttyUSB[0..N]`, but you may need to experiment with plugging and unplugging things to figure out which cable is which. If you want to keep things more constant from test to test, you can create symlinks to particular hardware adapters through the `/dev/serial/by-id` paths, which include a persistent unique ID.

For the purposes of this document, I will assume:

`/dev/ttyUSB0` = UART 0, the bootloader and debug console port

`/dev/ttyUSB1` = UART 1, the SPACE remote-command port
`/dev/ttyUSB2` = UART 2, the high-speed logging port

So if they are different on your system, just substitute the appropriate device name in the example commands. For the most part we will only be using `/dev/ttyUSB1`.

Command-line parameters

Providing the `--help` or `-h` flag prints out some more detailed command-line help:

```
space$ ./spacesuit.py --help
usage: spacesuit.py [-h] [--port PORT] [--address ADDRESS]
                    [--thruster {ace_radhard,ace_max}] [--logname
LOGNAME]
                    [--macro_file MACRO_FILE] [--debug]
                    (--command COMMAND | --script SCRIPT |
--interactive)

SPACESUIT: Serial Protocol for ACE Scriptable User Interface Tool

optional arguments:
  -h, --help          show this help message and exit
  --port PORT, -p PORT  serial port device to use
  --address ADDRESS, --addr ADDRESS, -a ADDRESS
                      SPACE device address, range 0-15
  --thruster {ace_radhard,ace_max}, -t {ace_radhard,ace_max}
                      thruster product name
  --logname LOGNAME, --log LOGNAME, -l LOGNAME
                      prefix for log filename
  --macro_file MACRO_FILE, --macro MACRO_FILE, -m MACRO_FILE
                      file containing script macro definitions
  --debug, -d         output additional packet debugging information
  --command COMMAND, --cmd COMMAND, -c COMMAND
                      SPACE command string to execute
  --script SCRIPT, --scr SCRIPT, -s SCRIPT
                      log file containing SPACE commands to execute
  --interactive, --int, -i
                      start interactive command prompt
```

Generally the only parameters which are required are `--port` and one of the final three options to tell the tool how to operate: run a command, run a script, or start an interactive session.

--port, -p

This should generally be provided and will typically be `/dev/ttyUSB[n]` to specify the USB adapter being used to connect to the SPACE port of the PPU, either via RS-422 or a TTL-level serial port. It defaults to `/dev/ttyUSB0`, so if you get some weird errors about invalid packets, make sure you are connecting to the remote command serial port instead of to the debug console serial port.

--address, -a

Devices on a SPACE port have a 4-bit address (range 0-15) to allow multiple devices to coexist on a shared RS-485 bus. So far all of the firmware images just use address `5`, and this is the default, so you shouldn't have to specify it.

--thruster, -t

Some features differ between ACE Rad-Hard and ACE Max, such as the mag-coil commands. By default the tool assumes `ace_radhard`, so you shouldn't usually need to specify this until we start working with ACE Max more regularly.

--logname, -l

Every protocol session will create a timestamped `.log` file in the current directory to record the commands sent and the replies received. The files are just plain text, and can be directly replayed as script files, or edited and then replayed. By default they are saved with the prefix "`suit_`", but this parameter allows you to name them something else if you want to keep track of what test scenario you are performing, etc. An example log file looks like this:

```
$ cat suit_20210206T052205.311Z
# 20210206T052210.915Z
> ping
# Status: 1
< ping

# 20210206T052212.800Z
> thrust on
# Status: 16
< thrust_ack on
```

Comments are prefixed with # at the start of the line (and blank lines are ignored). Outgoing commands are prefixed with > (and these are the only lines that will be replayed if the log file is used as a script), and incoming replies are prefixed with <. Commands will also be executed if they do not have any prefix character. Timestamps are given in UTC to millisecond precision.

--macro_file, -m

Commands and sequences of commands can get a little verbose, so for convenience the tool allows you to define macros to execute one or more commands via an abbreviation. The format of the macro definition file is pretty simple, just looking like:

```
$ cat macros.txt
macro1
  command a
  command b
# Comment
macro2
  command c
```

A macro abbreviation is introduced at the start of a line, and any indented lines underneath it are the commands which will be replayed when that macro is executed. Note that macros are allowed to reference other macros (in any order) as commands, too.

TODO: It will probably be useful to define some more complex logic in macros to support common operational and test scenarios, for instance "pause for N seconds", "repeat until telemetry value X is greater than Y", etc. Suggestions are welcome for what you want to see supported in the tool.

--debug, -d

This enables more verbose output for each command (though only when printed: the log files are the same). In particular it will detail the binary packets actually being sent and received, and will also provide the timing of the responses (though Python's time resolution is a bit inaccurate, so don't trust those numbers too much). Here's an example of the debug output for a command:

```
> ping
Sending: ping
 Packet: 1a ce a5 00 65 53
Elapsed time for reply: 4.883 msec
Received: ping
  Packet: 1a ce b5 10 00 01 76
# Status: 1
< ping
```

--command, -c

This executes a single command (or a macro) given directly on the command line, and then exits the session. For example:

```
space$ ./spacesuit.py -p /dev/ttyUSB1 -c "thrust on"
> thrust on
```

```
# Status: 16
< thrust_ack on
```

You might use this as part of a larger shell script, for instance. Note the use of quotes around the command string, which is required if it contains a space in order for it to be treated as a single command-line parameter value.

--script, -s

This executes multiple commands from a text file, which could be a spacesuit log file or a manually-created script. For example, the log file from the previous example can be replayed:

```
$ cat suit_20210206T052904.141Z.log
# 20210206T052904.141Z
> thrust on
# Status: 16
< thrust_ack on

$ ./spacesuit.py -p /dev/ttyUSB1 -s suit_20210206T052904.141Z.log
> thrust on
# Status: 16
< thrust_ack on
```

Note that a replayed script file will ignore the original timestamps of the recorded commands, instead just sending them back-to-back as fast as each reply is received.

--interactive, -i

This is likely to be the most common execution context to get started with, as it just starts an interactive session where the tool provides a command prompt and sends any commands you type, printing out the responses in real time. Bold text below shows user input:

```
space$ ./spacesuit.py -p /dev/ttyUSB1 -i
Enter 'quit' to exit. Type 'help' for more info.

> ping
# Status: 1
< ping

> thrust on
# Status: 16
< thrust_ack on

> tele discharge_setpoint time
# Status: 16
```

```
< tele_ack discharge_setpoint=300 time=2523.32
```

The interactive mode can be exited by entering the `quit` command. The tool also provides context-sensitive help for the available commands via a `help` command (note that the particular commands supported may be more extensive than when this manual was written):

```
space$ ./spacesuit.py -p /dev/ttyUSB1 -i
Enter 'quit' to exit. Type 'help' for more info.

> help
Known commands:
  delay: Get a reply after a delay
  discharge: Discharge control
  echo: Repeat the same packet data
  igniter: Igniter control
  ping: Send an empty SPACE packet
  tele: Retrieve telemetry fields
  telechan: Send extra channels in high-speed logging
  telefreq: Send telemetry field in high-speed logging
  telexl: Retrieve many telemetry fields
  thrust: Thruster high-level control
Type 'help [command]' for more info on each one.

> help discharge
Discharge control
Syntax: discharge [on/off]
Starts or stops the discharge controller with the currently-
configured (default) parameters. If a health condition prevents
attempting startup, the reply will be 'off' instead of
mirroring
the request.
```

The known telemetry fields (and their units) are listed under `help tele`.

Commands

Please see the Remote command set design doc for the canonical listing of the various remote commands, their exact syntax, and all of the defined telemetry fields. Not all commands and fields are currently implemented in the firmware, but the online `help` text should generally be up to date with what the desktop tool actually supports.

You may get a couple of different error messages depending on the state of implementation for a command:

```
space$ ./spacesuit.py -p /dev/ttyUSB1 -i
Enter 'quit' to exit. Type 'help' for more info.

> flow 0.101
Deadline of 2.000 msec expired.
# No reply received

> foobar 0.101
Unrecognized identifier label: foobar
```

In the first example above, the tool knows about the `flow` command, but the firmware has not yet implemented it, so the command actually gets sent on the RS-422 port but is ignored by the PPU, producing no reply within the timeout deadline.

In the second example, there is no such thing as a `foobar` command, so the tool cannot even parse the input line, and just prints an error message instead of being able to send anything on the serial port. An error of this type will abort a macro or a script to avoid producing unexpected behavior in the rest of the commands.

NOTE: When running these tools, please try to keep your PPU firmware version and the desktop tool version in sync (that is, from the same git commit) to avoid any weirdness in mismatched command behavior between sender and receiver.

There are also some SPACESUIT-specific commands in addition to the real ACE remote commands, to assist with scripted behavior. For example:

```
> pause [duration in seconds]
```

will cause the script to delay for a given interval before executing the next statement.

It is also possible to reflash the PPU over the SPACESUIT interface, either the application or the bootloader:

```
> flash [application_crc.bin]

> flashboot [bootloader_crc.bin]
```

These will both send a `sysreset` command afterwards to run the new firmware.

Configuration management

In addition to the `cvalue`, `cstring`, and `cerase` remote commands for updating individual configuration parameters, SPACESUIT also has a couple of higher-level scripted commands for dumping or restoring a complete set of all known parameters:

```
> csave [filename.cfg]

> cload [filename.cfg]
```

The `csave` command will read all of the configuration parameters from the board and save them to a file. You may edit it in your favorite editor, as the file format is plain text:

```
# Comment lines are ignored. The format is either:
#   name: live (default)
#   name: live (default -> local)
serial_num: 'ACE-0001' ('n/a' -> 'ACE-0001')
plenum_0psi: 0.0 (0.0)
plenum_100psi: 0.12 (0.1 -> 0.15)
```

The `cload` command will update the currently connected board with all of the parameter settings in the file, both the live and local values. Note that if no local value is provided for a parameter, it will be erased from the board's NVM to match the contents of the file. However, any parameters that are not listed at all in the file are left untouched, so you can overlay the effects of multiple configuration files for different thruster subsystems.

Note that in provisioning a new PPU, a `cerase all` command will be needed to format nonvolatile memory for the first time. Otherwise it will experience a `config_error` system fault on boot.

Python scripting

The `spacesuit.py` file, in addition to being a runnable tool, is also importable as a library in your own Python code, in order to send commands and retrieve telemetry with more custom intelligence.

Here's an example script showing some very basic usage:

```
$ cat spacesuit_example.py
#!/usr/bin/env python3

import spacesuit
```

```
parser = spacesuit.get_arg_parser()
arguments = parser.parse_args()
protocol = spacesuit.get_protocol(arguments)
reply = protocol.run_line("tele time", echo=True)
timestamp = protocol.telemetry['time']
print(reply)
print(timestamp)

$ ./spacesuit_example.py -p /dev/ttyUSB1
> tele time
# Status: 1
< tele_ack time=18.885

tele_ack time=18.885
18.885000228881836
```

This makes use of the normal `spacesuit.py` command-line options to specify things like the serial port. You first obtain a parser object (from the standard `arg_parse` library) with `get_arg_parser()` and may add your own parameters before using it to parse the command line. The resulting arguments are passed to the `get_protocol()` function to obtain a `LoggingSpaceProtocol` object that establishes a connection to the PPU.

The two most useful methods of that object are `run_line()` and `run_script()`, which do what you would expect. The first returns the reply as a string if applicable, though scripts and macros just return `None` instead. There is also a `send_text()` method which bypasses things like the macro engine to only send a single remote command.

The protocol object also contains a `telemetry` dictionary which captures the last value seen for each telemetry field (keyed by the field's canonical text name), which makes it easier for your code to examine those values as numbers instead of needing to re-parse the reply from a `tele` command. Note that the dictionary only captures values which have been previously requested, so it won't retrieve a new value by itself if you are monitoring things in a loop.

I'm not quite sure how people will want to make use of these features, so let me know what additional capabilities would make sense to put into the tool, since there are some tradeoffs between what you might want to do in a macro versus what you might want to do in a full custom Python script.

High-speed logging

NOTE: This feature is a little on the experimental side, but it's supported in the firmware and may be useful for thruster testing, so feel free to give it a try.

In addition to the on-demand telemetry remote commands, the firmware can be configured to continuously stream data for selected fields on one of the RS-422 ports. This path is optimized for relatively high throughput, so you can capture the PPU's behavior at kilohertz rates to see what the high-speed control loops (such as the discharge controller) are doing.

To make use of this feature, you will want to open up a second terminal window and run the `flight/utils/logstream_test.py` utility in parallel to the regular remote-command tool, providing it with the device name of the second RS-422 port.

The `telestart` remote command begins the logging data stream, and `telefreq` begins streaming a specific telemetry variable with a particular maximum sampling frequency. (A frequency of 0 stops that field's output.) Here's some example usage, showing the two terminal windows side by side:

```
space$ ./spacesuit.py -p /dev/ttyUSB1 -i
Enter 'quit' to exit.
Type 'help' for more info.

> telestart uart2 1000000
# Status: 1
< telestart_ack uart2 1000000

> telefreq igniter_on=1000
# Status: 1
< telefreq_ack igniter_on=1000

> telefreq discharge_voltage=25
# Status: 1
< telefreq_ack discharge_voltage=25

> igniter 0.5
# Status: 1
< igniter_ack 0.5

> igniter 3
# Status: 1
< igniter_ack 3
```

```
utils$ ./logstream_test.py /dev/ttyUSB2



Registered: 1001 as console_input
Registered: 1002 as console_output


Registered: 1 as igniter_on



Registered: 13 as discharge_voltage
```

The logstream tool will print out a message whenever a new telemetry field starts up, including a few default ones established at the start of a stream. When you're done with the tool, just type CTRL-C to terminate it. As it's running, it will create `.tsv` tab-separated-value files in the current directory, each one named for a telemetry field. The format should be suitable for graphing the data in tools like `gnuplot`, or importing into your favorite spreadsheet. For instance, the above example creates these two files:

```
$ cat igniter_on.tsv
621.350592    1
621.849897    0
628.212448    1
631.211893    0
```

```
$ cat discharge_voltage.tsv
595.361945    0.0
595.401965    0.0
595.441985    0.0
595.482005    0.0
595.522025    0.0
```
… etc.

The first column is the timestamp in seconds since the board was booted, to microsecond precision. The debug telemetry variable `time` gives this value as well if you need to sync up events between a command script log and a high-speed log file. The second column gives the telemetry field value at that instant.

The igniter file shows that the igniter was turned on at *t*=621.35, then off at *t*=621.85, producing the 0.5-second ignition pulse requested in the remote command. And the discharge file is continuously logging the voltage value (here just 0.0 V, since the controller is inactive) at an interval of 0.04 seconds, matching the 25 Hz frequency requested by the `telefreq` command. The discharge control loop is actually measuring this value at 25000 Hz, so it is being subsampled at a 1:1000 reduction to produce this output.

The exact performance limits of the logging stream have not been fully explored, but it definitely should be happy at kilohertz rates, and it should be able to handle a few variables at even the full discharge control-loop frequency. In scripts, it may be useful to change the output frequency dynamically so that you selectively record data only during an ignition event, for instance.

NOTE: Let me know how useful you find this capability and if you have any suggestions for improvements. For instance, one penciled-in plan is to integrate the tool with a data-graphing library so it'll just pop up a window with a plot of the telemetry variables.