
RoverCore-S Documentation

Release 1.0.0


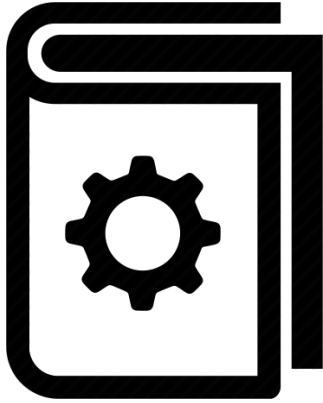

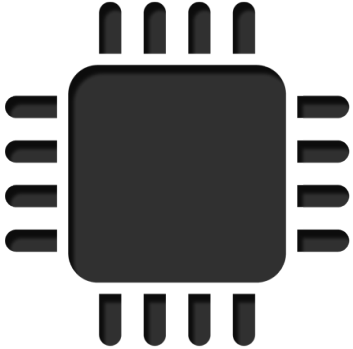
Khalil A. Estell

Aug 31, 2017

1	Quick Links	1
1.1	Getting Started	1
1.1.1	Getting Started	1
1.1.1.1	Prerequisites	1
1.1.1.2	Installation	2
1.1.1.3	Building and Loading Hello World Application	2
1.1.1.4	Building and Loading FreeRTOS Project	3
1.1.2	Understanding The Framework Layout	3
1.1.2.1	File Hierarchy	3
1.1.2.2	Folder: firmware	3
1.1.2.3	Folder: firmware/default	3
1.1.2.4	Folder: firmware/default/bin	4
1.1.2.5	Folder: firmware/default/application	4
1.1.2.6	Folder: firmware/default/application/<application>/_can_dbc	4
1.1.2.7	Folder: firmware/default/application/<application>/L5_Application	4
1.1.2.8	Folder: firmware/default/lib	4
1.1.2.9	Folder: firmware/default/L%d_%s	4
1.1.2.10	Folder: firmware/default/obj	4
1.2	Guides	5
1.2.1	Debugging with OpenOCD and GDB	5
1.2.1.1	Step 0: Installing OpenOCD	5
1.2.1.2	Step 1: Rebuild Application with Debug flag	5
1.2.1.3	Step 2: Solder JTAG Headers to SJOne	5
1.2.1.4	Step 3: Connecting BusBlasterv3 to SJOne	5
1.2.1.5	Step 4: Run OpenOCD	5
1.2.1.6	Step 5: Run GDB	6
1.2.2	Unit Testing with CGreen	7
1.2.2.1	Unit Testing Tools	7
1.2.3	Telemetry: Embedded Runtime Monitoring and Tuning	7
1.2.3.1	Setting up Telemetry	7
1.2.3.2	Using Telemetry	7
1.2.4	Press Next To Get Started	8
1.3	About	8
1.4	Copyrights	9

CHAPTER 1

Quick Links

			
Get Started	Guides	API Reference	Hardware Reference

Getting Started

Getting Started

Prerequisites

Need a running version of Ubuntu 14.04 LTS or above. Ubuntu in a virtual machine such as VirtualBox or VMPlayer will work as well.

Note: The it is possible to get SJSU-Dev-Linux to work completely on Windows and Mac OSX if you have all of the necessary PATH dependencies installed, but that is not covered here. You will need to manually install all of the necessary components in the installer script.

Installation

Step 1 Clone the repository

```
git clone --recursive https://github.com/kammce/SJSU-DEV-Linux.git
```

Step 2 Change directory into SJSU-Dev-Linux

```
cd SJSU-DEV-Linux
```

Step 3 Run setup script.

```
./setup
```

Warning: Do not run this script using **SUDO**. The script will ask you for **sudo** permissions once it runs.

Note: This will install gtkterm, mono-complete, and gcc-arm-embedded packages

Building and Loading Hello World Application

Step 1 From the root of the repository

```
cd firmware/default
```

Step 2 Run build script. A HEX file bin/HelloWorld/HelloWorld.hex and subsequent folders should have been created after this script finishes.

```
./build HelloWorld
```

Note: use the `--help` argument to get additional information on how to use the build script.

Step 3 To load the hex file into your SJOne file you will use the `hyperload.py` file. Run the following:

```
./hyperload.py /dev/ttyUSB0 bin/HelloWorld/HelloWorld.hex
```

The first argument is the path to the serial device. The second argument is the hexfile to load into the SJOne board.

Step 4 To view serial output, run GTKTerm by using the following command:

```
gtkterm -p /dev/ttyUSB0 -s 38400
```

How to use GTKTerm

1. Set *CR LF Auto* to true by going to the Main Menu > Configuration > CR LF Auto and click on it.
2. Press F8 (Clears RTS signal), then press F7 (Clears DTR signal) to start SJOne.
3. You should see the board counting up on the 7-Segment display and in binary on the LEDs.

Step 5 Done!!

Building and Loading FreeRTOS Project

Instructions are the same as HelloWorld, but you need to change the run the build script's last argument to *FreeRTOS* rather than HelloWorld.

Understanding The Framework Layout

File Hierarchy

```
firmware
- default
  - applications
    |   - FreeRTOS
    |   |   - _can_dbc
    |   |   - L5_Application
    |   |   - examples
    |   |   - periodic_scheduler
    |   |   - source
    |   |   - cmd_handlers
    |   - HelloWorld
    |       - _can_dbc
    |       - L5_Application
    |       - examples
    |       - periodic_scheduler
    |       - source
    |       - cmd_handlers
  - bin
    |   - FreeRTOS
    |   - HelloWorld
  - lib
    |   - _can_dbc
    |   - L0_LowLevel
    |   - L1_FreeRTOS
    |   - L2_Drivers
    |   - L3_Utills
    |   - L4_IO
    |   - newlib
  - obj
    |   - lib
    |   - FreeRTOS
    |   - HelloWorld
```

Folder: `firmware`

This folder is meant to hold **projects**. **default** is, understandable, the default project setup.

Folder: `firmware/default`

Important: This is how you start a new project.

If you want to change, modify, or update files in the `:code:lib` folder, then it is **RECOMMENDED** for you to make a new project by copying and renaming the default folder to something else. Example: renaming the new folder to `cmpe146` to hold all of your course work that could result in changing the lib files.

Making new projects is helpful, because, the default folder is the one that is modified when there is a new feature added to the repository. To keep your changes, make a new folder.

Note: If you would like to contribute to this project, then editing the lib files in the default folder is permitted.

Folder: `firmware/default/bin`

This folder holds the executables that can be loaded into the SJOne board `.hex` as well as a disassembly file `.lst`, linker file `.map` and the Executable and Linkable Format `.elf` file.

Folder: `firmware/default/application`

This folder holds all of the applications for a given project. Applications use the same base libraries but have different files for using them. Majority of code should be written here.

Important: This is how you start a new application.

To **start** a new application, copy the **FreeRTOS** or **HelloWorld** (depending on what you want to do) folder and rename it to the name of your application.

Folder: `firmware/default/application/<application>/_can_dbc`

The `_can_dbc` folder holds the CAN message description files and header generator.

Folder: `firmware/default/application/<application>/L5_Application`

The `L5_Application` folder holds the `main.cpp` file and other application layer files.

Folder: `firmware/default/lib`

This folder holds the core firmware files for the project, such as abstractions for using GPIO, I2C, UART, Interrupts, etc.

Folder: `firmware/default/L%d_%s`

The folders that start with **L<some number>_<some folder name>** are kind of self explanatory as to what they hold. For example, `L1_FreeRTOS` holds files pertaining to FreeRTOS and the FreeRTOS port files. `L2_Drivers` are device drivers and so on and so forth.

Folder: `firmware/default/obj`

This folder holds object files created during the compilation stage of building. They are then all linked together to create an `.elf` file afterwards.

Guides

Debugging with OpenOCD and GDB

This tutorial will use **HelloWorld** as an example. But this will work for any application you build.

Step 0: Installing OpenOCD

OpenOCD was installed when you ran the initial `./setup` script.

Step 1: Rebuild Application with Debug flag

Run:

```
./build spotless
./build -d HelloWorld
```

Note: `./build spotless` will delete all of the files in the `obj` and `bin` folder. This is necessary because some files in the `lib` folder need to be updated with the new `-d` (debug) flag.

Step 2: Solder JTAG Headers to SJOne

Do as the title says if you haven't already.

Step 3: Connecting BusBlasterv3 to SJOne

Connect jumpers from the GND, TRST, TDI, TMS, TCK, and TDO pins on the **BusBlasterv3** to the SJOne's JTAG headers.

Danger: DOUBLE AND TRIPLE CHECK THAT YOUR CONNECTIONS! The SJOne costs \$80 and the BusBaster costs \$35! That's \$115 down the drain if you burn them out!

Step 4: Run OpenOCD

Run:

```
# If you used make install
openocd -f ./tools/OpenOCD/sjone.cfg
```

Tip: Successful output is the following:

```
Info : clock speed 100 kHz
Info : JTAG tap: lpc17xx.cpu tap/device found: 0x4ba00477 (mfg: 0x23b (ARM Ltd.),
↳part: 0xba00, ver: 0x4)
Info : lpc17xx.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Error: If you see the following message:

```
Error: JTAG-DP STICKY ERROR
Info : DAP transaction stalled (WAIT) - slowing down
Error: Timeout during WAIT recovery
Error: Debug regions are unpowered, an unexpected reset might have happened
```

Then the SJOne board is being held in a RESET state. To fix this, either by power cycling the SJOne board or by deassert the RTS and DTR signals through GTKTerm.

Error: If you see your terminal get spammed with this:

```
Error: JTAG-DP STICKY ERROR
Error: Invalid ACK (7) in DAP response
Error: JTAG-DP STICKY ERROR
Error: Could not initialize the debug port
```

Then its a good chance that one of your pins is not connected.

Step 5: Run GDB

Open another terminal and run the following command in the `firmware/default/` folder.

```
arm-none-eabi-gdb -ex "target remote :3333" bin/HelloWorld/HelloWorld.elf
```

Tip: You can run `arm-none-eabi-gdb` without arguments and use the following `gdb` commands file `bin/HelloWorld/HelloWorld.elf` then `target remote :3333` in the `gdb` command line interface to get the same effect as the above command.

At this point the SJOne board has been halted. You should be able to add breakpoints to the program at this point and step through the code.

At this point you will not see any source code. Do the following in the `gdb` command line interface:

```
>>> break main
>>> continue
```

Tip: Don't use the typical `run` command to "start" the code. It is already... kinda started. Also, `run` does not exist when using `target remote :3333` to OpenOCD. It exists with `target extended-remote :3333`, but causes issues... just don't use it OK.

At this point you should see the source code of your `main.cpp` show up. Now you can step through your code and set breakpoints using `step`, `next`, `finish` and `continue`, `break`, etc.

For a `gdb` cheat sheet, see this PDF:

<http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>

Error: If your board keeps restarting, this is due to the Watchdog not getting fed. Although, this shouldn't happen if you ran step 0 correctly. If you do a build spotless and build your project again with the -d flag, and this still does not work, then as a last resort, go into the `lpc_sys.c` file and comment out the `enable_watch_dog()` function call.

Unit Testing with CGreen

Unit Testing Tools

To unit test we use CGreen.

Warning: This section is not complete

Telemetry: Embedded Runtime Monitoring and Tuning

Wikipedia:

Telemetry is an automated communications process by which measurements and other data are collected at remote or inaccessible points and transmitted to receiving equipment for monitoring.

Telemetry is another means of testing your firmware. Unit test are useful for testing your code's logic and making sure the behavior of your code operates as intended. A debugger allows you step through your code one line at a time, inspecting variables to see when adverse behavior arises in your firmware. Telemetry, more or less, is a means of feeding back information to the user about the current state of the firmware during runtime.

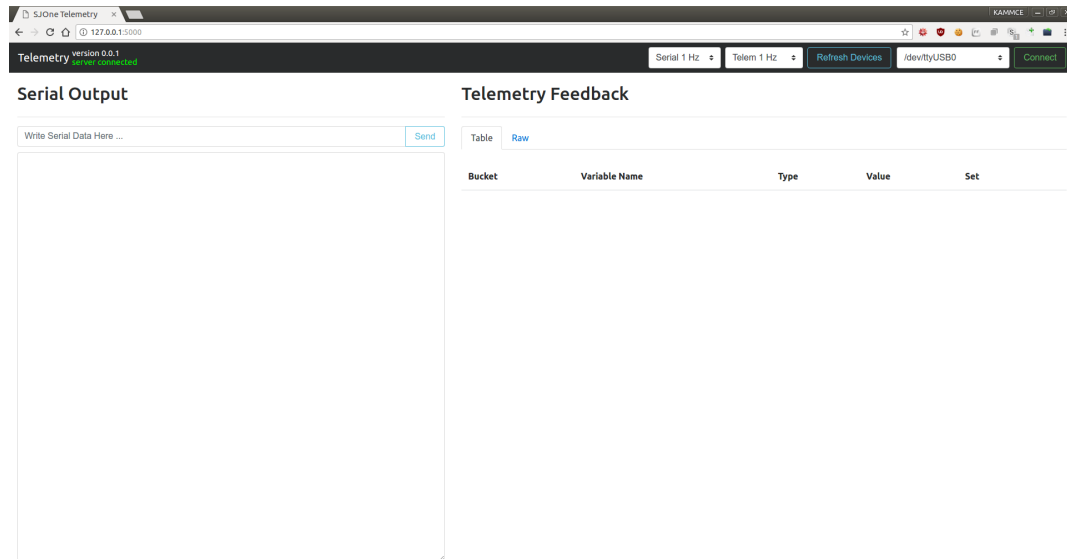
Setting up Telemetry

Telemetry was setup when you ran the initial `./setup` script.

Using Telemetry

Step 1 Run `./start` script. It should open up a webpage in your browser.

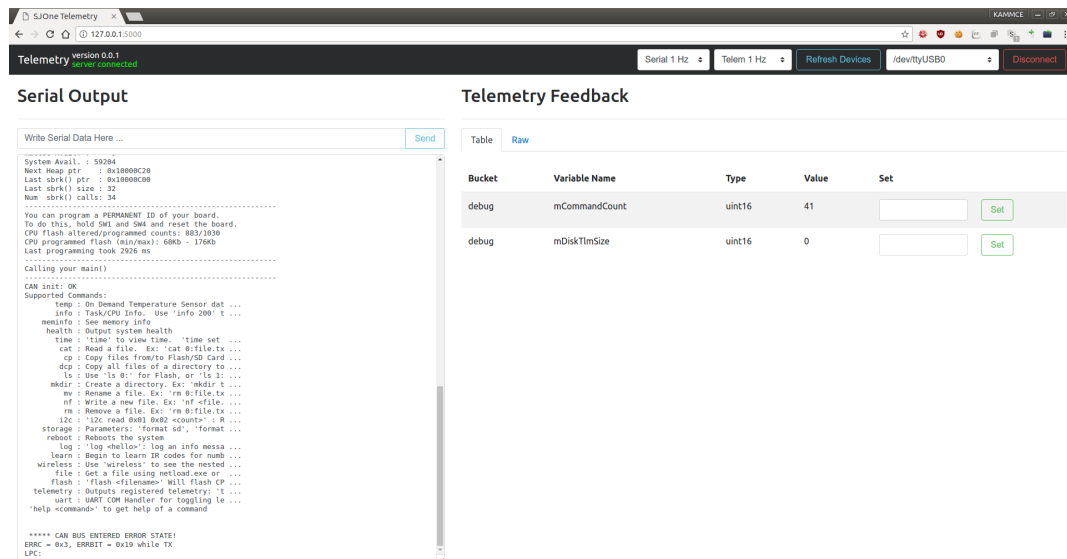
You should see the following:



Step 2 Connect your SJOne Board to your computer.

Step 3 Press the **Refresh Devices** button to check your system for serial devices.

Step 4 Press the **Connect** button to connect to the serial device. At this point, you should see the serial output of the SJOne board being written to the Serial Output textarea. If Telemetry is running on the SJOne, then a table will be generated in the Telemetry Feedback area.



Press Next To Get Started

About

Warning: This section is not complete

Copyrights

Warning: This section is not complete

CHAPTER 2

Press Next To Get Started
