ARA Smart Power System-Power Box (ASPS-Power) Protocol and Firmware Information

This document describes the ASPS-Power microcontroller communication protocol and firmware information.

Physical Layer

ASPS-Power communicates with the ASPS-DAQ system, which has an Ethernet communications link for communication with South Pole. Details of the ASPS-DAQ/Ethernet communication are not located here.

There are 4 control lines between ASPS-DAQ and ASPS-Power.

Name (at ASPS-Power)	Function
ASPS_TX	Data from ASPS-Power to ASPS-DAQ
ASPS_RX	Data from ASPS-DAQ to ASPS-Power
RST_ASPSPWR	Reset from ASPS-DAQ to ASPS-Power
RST_ASPSDAQ	Reset to ASPS-DAQ from ASPS-Power

The data communication occurs over the ASPS_TX/ASPS_RX pair, in a standard UART communications link, running at 9600 bps, 8 bits, no parity, 1 stop bit.

Note: *neither* RST_ASPSPWR or RST_ASPSDAQ are the same thing as the *direct* reset line to the MSP430F2274 microcontroller itself. That signal is RST/SBWTDIO, and is available on the programming header on the ASPS-Power board.

Overall Protocol Description

Communication to/from the ASPS-Power is via JavaScript Object Notation (JSON) strings, terminated by a newline ('\n').

Data from ASPS-Power

There are 4 JSON message types sent from ASPS-Power. All of them are followed by an array of data.

- "ON" messages: indicates which outputs are on.
- "V" messages: voltage measurements.
- "I" messages: current measurements.
- "T" messages: temperature measurements.

ON message example

{"on":[1,2,3]}

This indicates that outputs 1, 2, and 3 are on, and therefore output 0 is off.

V message example

{"v":[525,680,16987]}

3 output voltages are measured by ASPS-Power: in order, they are the +15V Vicor output (divided by 10), 3.3V MSP430 voltage, and the input 300V voltage. In this example the 300V measurement is *not present*. It would appear as a 3rd value after the second.

Nominal (uncalibrated) conversions are:

- +15V Voltage: V = (ADC*2.5/1023)*10 = ADC*0.02444
- MSP430 Voltage: V = (ADC*2.5/1023)*2 = ADC*0.004888
- VIN Voltage: V = (ADC*2.5/32768)*249 = ADC*0.0189972

The above would correspond to +15V = 12.831 V, MSP430 = 3.323 V, VIN = 322.7 V.

I message example

{"i":[-618,-525,-452,-341]}

This message returns the 4 currents measured from the downhole system. These currents are measured as the voltage after an 0.02 shunt resistor, divided by 10: as in,

Vshunt = (15V – I*(0.02 ohms))/10.

After this, the result is sent through a difference amplifier with a gain of ~30 against the +15V measurement (divided by 10), giving:

Vmeasured = (+15V/10) – (+15V/10 – Vshunt)*30

In theory this should give

Vmeasured = +1.5V – I*(0.06 ohms)

The final measurements are oversampled and averaged to improve resolution and also have the 1.5V subtracted from them. They have a range from -32768 to 32767. Because the opamps are not precision, the offsets and gain to get back to current in milliamps will have to be calibrated if any precision is required.

In the above example, *no* current was being drawn, so this is a measurement of the offset.

T message example

{“t”:[432,24,20,-64]}

This message returns the 4 temperatures measured. They are (in order) the MSP430 internal temperature, the TMP422 internal temperature, external sensor 0, and external sensor 1.

The 3 TMP422 measurements are calibrated (by the TMP422) and reported in degrees C. Note that the sensor actually reports value in fractions of degrees C, but the software reads out only to degree C precision since the precision is not extremely important.

If a sensor reports ‘-64’ that indicates that the sensor is not connected.

A *typical* conversion for the MSP430 internal sensor (non-calibrated) is

- T = ((ADC*2.5/1023)-0.986)/0.00355 = 0.6884*ADC – 277.7

The above example corresponds to 20 C.

Commands to ASPS-Power

ASPS-Power receives 5 messages currently: “set”, “calib”, “disable”, “sn”, “fw”.

“set” message example

{“set”:[14,8]}

The set message controls the power to the 4 outputs. Both values are bitmasks, and should be interpreted as binary values. The *first* value (here 14: 1110 in binary) indicates *which outputs are being addressed*. The *second* value (here 8: 1000 in binary) gives *what to do with those outputs.*

So in this example, outputs 1, 2, and 3 are being controlled (since bits 1, 2, and 3 are set). Output 3 is being turned on (since bit 3 is set) and therefore outputs 1 and 2 are being turned off.

Note: the “set” message must be sent with a ‘\n’ (newline) at the end. Some serial terminals (e.g. Tera Term Pro) by default send a ‘\r’ when Enter is pressed. Tera Term Pro can send a ‘\n’ via Ctrl-Enter on Windows.

“calib” message example

{“calib”:[0,656]}

The calib message sets the offset for the *I message* outputs. The first value is the index (0-3), and the second is the offset value.

This example would change channel 0’s offset to be 656 (so a raw value of 656 would be output as 0).

“disable” message example

{“disable”:[0,1,2]}

The disable message sets the turn-on behavior for the outputs. If a channel is disabled, it does *not* power on immediately. Note that this does not prevent the channel from being turned on by a *“set”* command. It also does *not* turn off the channel. It simply changes initial power-on behavior.

In the above example, output channels 0, 1, and 2 would not turn on at power on.

One can disable any combination of channels. Eg, {“disable”:[0]} and then {“disable”:[1]} (in sequence, one right after another) would disable channels 0 and channels 1. You could also have entered {“disable”:[0,1]} for the same effect.

“enable” message example

{“enable”:[0,1,2]}

The enable message sets the turn-on behavior for the outputs. If a channel is enabled, it *does* power on immediately. Note that this does not prevent the channel from being turned off by a *“set”* command. It also does *not* turn on the channel. It simply changes initial power-on behavior.

In the above example, output channels 0, 1, and 2 would turn on at power on.

One can enable any combination of channels. Eg, {“enable”:[0]} and then {“enable”:[1]} (in sequence, one right after another) would enable channels 0 and channels 1. You could also have entered {“enable”:[0,1]} for the same effect.

"sn" message example

{"sn":0}

This message requests the serial number of the ASPS-Power board. The serial number is actually calib[30], which means it can be written with {"calib":[30,###]} where ### is the serial number.

It responds with

{"sn":###}

"fw" message example

{"fw":0}

This message requests the version number of the firmware on the ASPS-Power board.

It responds with

{"fw":"1.0.0"}

Firmware

The ASPS-Power firmware is located at

http://github.com/barawn/energia-aspspower

based on the Energia (http://energia.nu) framework. Instructions for building the firmware are located in the README.md file.

To program the microcontroller, any MSP430 programmer (including a simple MSP-EXP430G2 LaunchPad – see http://43oh.com/2011/11/tutorial-to-use-your-launchpad-as-a-programmer/) can be used. Connect to RST/SBWTDIO, TEST/SBWTCK, and GND pins on J11 and "Verify/Upload" on Energia.

More info on using an MSP-EXP430G2 and Energia to *directly* program

The MSP-EXP430G2 has a 20-pin socket to house an MSP430 microcontroller, and all of those 20 pins are broken out on the J1/J2 headers.

So to use the LaunchPad to interface with *any* MSP430, you just leave the microcontroller *off the board*, and connect to the pins you need. For programming, this is TEST/SBWTCK, RST/SBWTDIO, and GND. See http://processors.wiki.ti.com/index.php/JTAG_(MSP430) for more information on MSP430 programming. Note that the extra components (pullup resistor, capacitors, etc.) aren't needed in this case because they're already on the MSP-EXP430G2. See the schematic (http://www.ti.com/lit/ug/slau318g/slau318g.pdf , pages 18-20).

Note: the signals you connect to on the ASPS-Power board are on the programming header (J11), labelled

- TEST/SBWTCK
- RST/SBWTDIO
- GND

You *do not need to connect 3.3VSB* because the ASPS-Power board is already powered. Note that TEST/SBWTCK is *not* directly connected to the corresponding pin on the microcontroller because the TEST pin is multiplexed with ASPS_RX , depending on RST_ASPSPWR. This detail is covered in the next section.

For communication, you can hook up the ASPS_RX/ASPS_TX pins on the J2 connector on ASPS-Power to the corresponding RX/TX pins on the LaunchPad. See the schematic to determine which pins to connect to.

Programming the microcontroller *in-situ* through the Bootstrap Loader (BSL)

The MSP430 series contains a built in "Bootstrap Loader" which allows you to program the microcontroller through the serial connection directly. For more information on bootloaders, see http://www.engineersgarage.com/tutorials/bootloader-how-to-program-use-bootloader . For more information on the MSP430 built-in (ROM) bootloader, see http://www.ti.com/tool/mspbsl .

The ROM bootloader is entered by sending the correct sequence on the RST/SBWTDIO and TEST/SBWTCK **pins on the microcontroller**. (Note that the TEST/SBWTCK net on the ASPS-Power board is ***not*** the pin on the microcontroller, but it is multiplexed onto it).

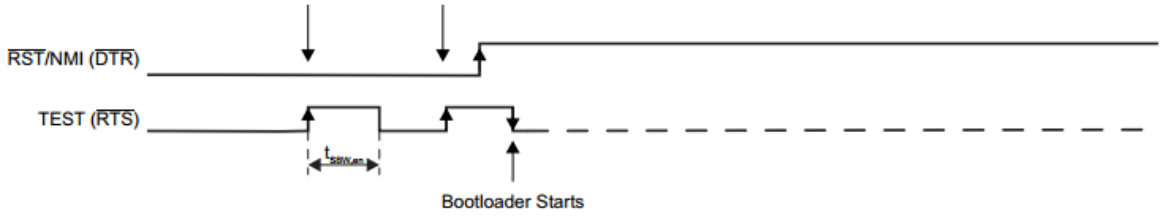

Figure 2. BSL Entry Sequence at Shared JTAG Pins

TEST and RST are not directly connected via the ASPS-DAQ connection, but this sequence can still be generated. To send this sequence via the control lines from the ASPS-DAQ board, do

1. Make sure ASPS_RX is high (idle).
2. Assert (lower) RST_ASPSPWR.
3. Lower ASPS_RX.
4. Raise ASPS_RX.
5. Lower ASPS_RX.
6. Raise ASPS_RX.
7. Deassert (raise) RST_ASPSPWR.
8. Wait some time (1 ms). The microcontroller should be in the BSL, and ASPS_RX can again be used as the serial port transmit (receive at microcontroller).

The multiplexing is done via U14, an NL7SZ57. When RST_ASPSPWR is asserted (low), the output of this logic chip (which is MSP430_TEST, the actual microcontroller TEST/SBWTCK pin) is the ***inverted*** ASPS_RX signal. This is why the sequence looks inverted (lower/raise instead of raise/lower).

In addition, RST_ASPSPWR is ***not*** the microcontroller reset line: when asserted (lowered), it generates a *reset pulse* on RST/SBWTCK. This prevents ASPS-DAQ from accidentally forcing the microcontroller permanently into reset. This is why you must wait 1 ms (for the reset pulse to expire) at the end of the sequence.

For details on the MSP430 BSL protocol, see http://www.ti.com/lit/ug/slau319l/slau319l.pdf .

Note that you *cannot* program the microcontroller via the BSL in Energia, which requires a direct connection to a programming tool (e.g. the LaunchPad). Instead, the binary program itself must be exported from Energia and programmed via a separate tool.