boxcom howto

April 27, 2013

Contents

1 USB board

1.1 The front panel switch

Figure 1 shows how the front panel power switch should be wired.

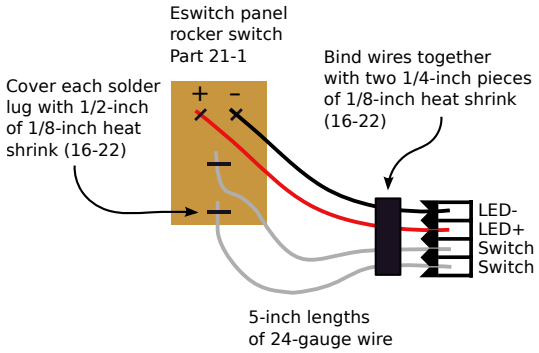

Figure 1: Panel switch wiring

1.2 Board checkout

1.2.1 Voltage rails

Use table 1 to keep track of voltage rails.

Net name	Test points	Acceptable	Actual
V_{bus}	TP100 vs. TP101	4.5V → 5.5V	
$+3.3V_{aux}$	TP400 vs. TP401	3.14V → 3.45V	
$+3.3V_{mon}$	TP500 vs. TP401	3.14V → 3.45V	

Table 1: Voltage rail checkout table for the USB board.

1.2.2 Current monitor

The current monitor output at J500 will have a fixed DC output, since the voltage regulator following it always draws at least 1mA. As illustrated in figure 2, the slope set in hardware should give $\Delta V_{out} = 1V$ for each additional 10mA of current draw from J501. Since the voltage output from J501 is controlled at 3.3V, a test load of 3.3kΩ should increase the voltage at J500 by 100mV.

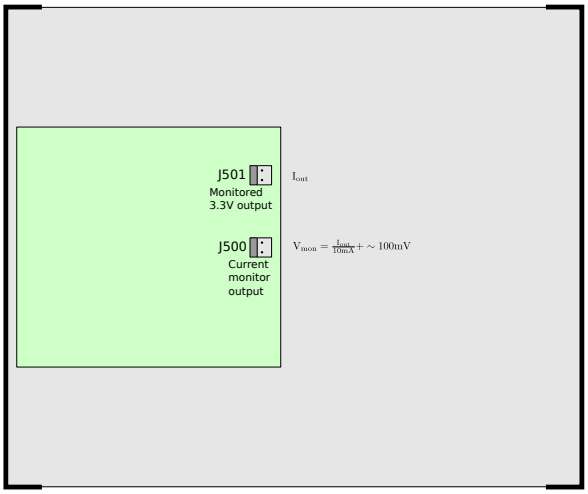

Figure 2: The output connectors used during the current monitor test.

Load applied to J501	Acceptable V_{out} at J500	Measured V_{out} at J500
Open	90mV → 110mV	$V_{out,o}$ =
3.3kΩ	$V_{out,o}$ + 100mV	

Table 2: Passing voltage measurements for the current monitor test.

1.2.3 Serial loopback

The serial loopback test is a basic test of the USB/serial interface and the RS-232 transceiver. Make the breakout cable shown in figure 3, then make connections to the board as shown in figure 4.

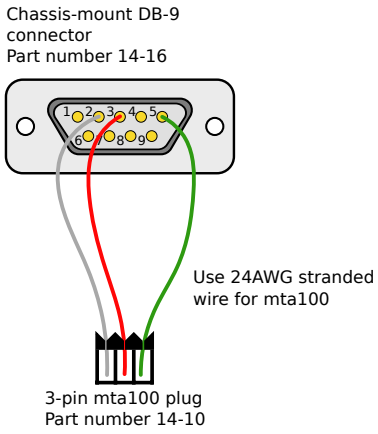

Figure 3: Wiring the DB9 breakout cable for the serial loopback test.

The serial loopback test script is:

`boxcom/implement/data/scripts/tty_loopback.py`

...and the test should pass at the speed listed in table 3.

Minimum passing baud	Measured passing baud
115200	

Table 3: Passing baud measurement for the serial loopback test. The usb board should be able to reliably pass the loopback test for data flowing in both directions at the minimum baud.

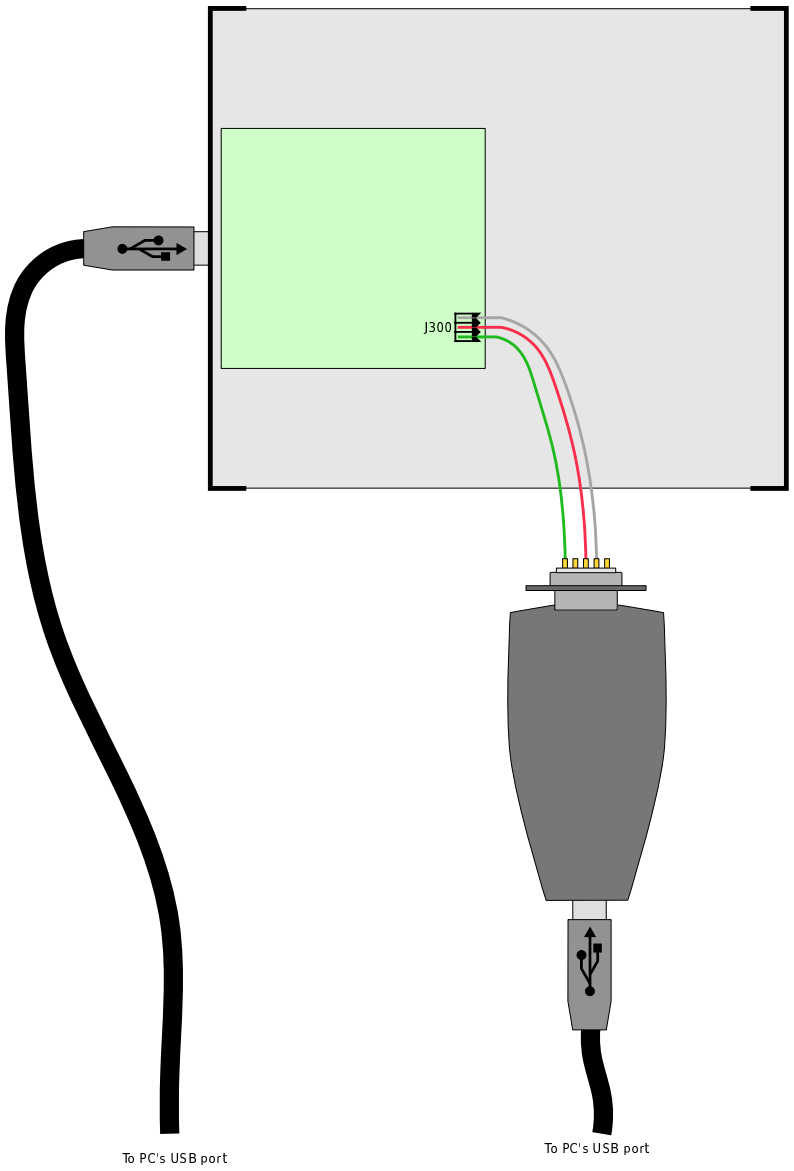

Figure 4: The setup for the serial loopback test.

2 Butterfly board

2.1 Making connections

Figure 5 shows the connections that should be made to the Butterfly board.

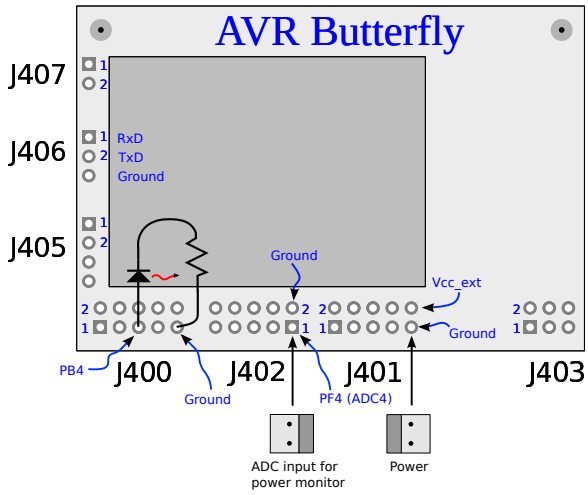

Figure 5: Connections to the AVR Butterfly

Figure 6 shows how the UART cable should be made.

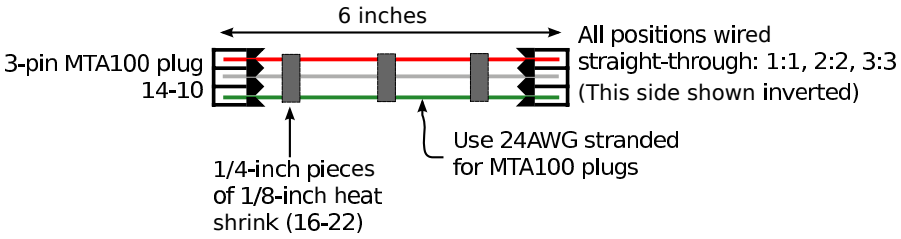

Figure 6: The UART cable connecting the Butterfly and USB boards.

3 Firmware

3.1 Adding a new remote command

1 – Choose a name for the command The command characters, the argument characters, one space, and one string terminator must all fit in the received command buffer. The definition of this size is shown below.

File: bx_command.h

```
...
#define RECEIVE_BUFFER_SIZE 20
...
```

2 – Think about the command's arguments The code can only handle unsigned hexadecimal number arguments formatted as strings. If you need something else, you'll have to write more code.

3 – Add a function for your command to call I like to put the new function in the module where it belongs, and just add a "cmd_" prefix to it. If the command is a query, I add a "_q" suffix. The command corresponding to the `vcounts?` query is shown below.

File: bx_adc.c

```
...
void cmd_vcounts_q(uint16_t nonval) {
  uint16_t adc_temp = 0;
  adc_temp = adc_read();
  usart_printf_p(PSTR("0x%x\r\n"),adc_temp);
}
...
```

4 – Give the new command an entry in the command array A sample entry in the command array is shown below. New entries must be added before the "end of table indicator." Remember that hexadecimal arguments larger than 4 characters don't make sense for 16-bit integers (leading `0x` characters are not allowed).

File: bx_command.c

```
...
command_t command_array[] ={
  // hello -- Print a greeting.
  {"hello",           // Name of the command
    "none",           // Argument type (can be "none" or "hex" right now)
    0,                // Maximum number of characters in argument
    &cmd_hello,       // Address of function to execute
    helpstr_hello},   // The help text (defined above)
  // End of table indicator.  Must be last.
  {"","",0,0,nullstr}
};
...
```

3.2 Logger functions

3.2.1 logger_msg_p

```
void logger_msg_p( char *logsys, logger_level_t loglevel, const char *logmsg, ... );
```

Send a message to the logger module from permanent memory

Parameters

- `logsys`

  Pointer to a string matching one of the logger system strings.

- `loglevel`

  One of the logger level identifiers:

  `log_level_ISR` (lowest level)

  `log_level_INFO`

  `log_level_WARNING`

  `log_level_ERROR` (highest level)

- `logmsg`

  Pointer to a string stored in permanent (flash) memory. This might be a C format string.

- `...` *(additional arguments)*

  Depending on the format string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a format specifier in the format string. There should be at least as many of these arguments as the number of values specified in the format specifiers. Additional arguments are ignored by the function.

Examples

```
logger_msg_p("command",log_level_INFO,
             PSTR("Command '%s' recognized.\r\n"),command_array -> name);
```

Output (After receiving the `hello` command):

```
[I](command) Command 'hello' recognized.
```

Alphabetical command index

Internal command index