boxcom howto

May 11, 2013

Contents

1 USB board

1.1 The front panel switch

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

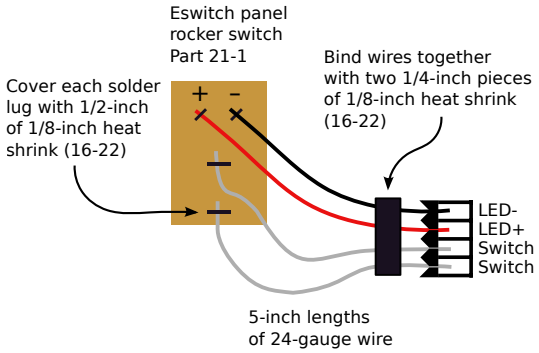


Figure 1: Panel switch wiring

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.

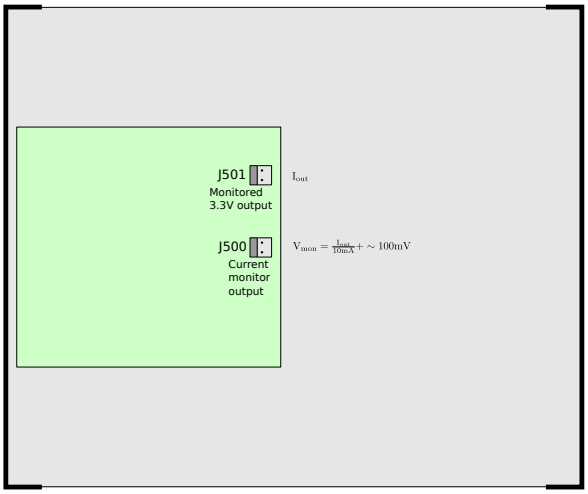


Figure 2: The output connectors used during the current monitor test.

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.

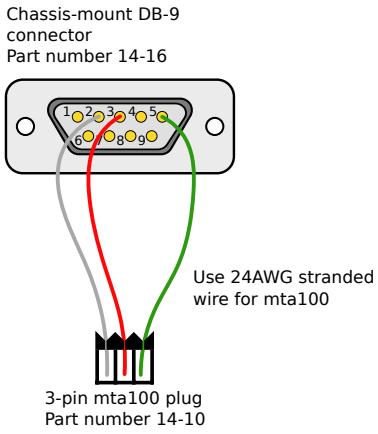


Figure 3: Wiring the DB9 breakout cable for the serial loopback test.

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.

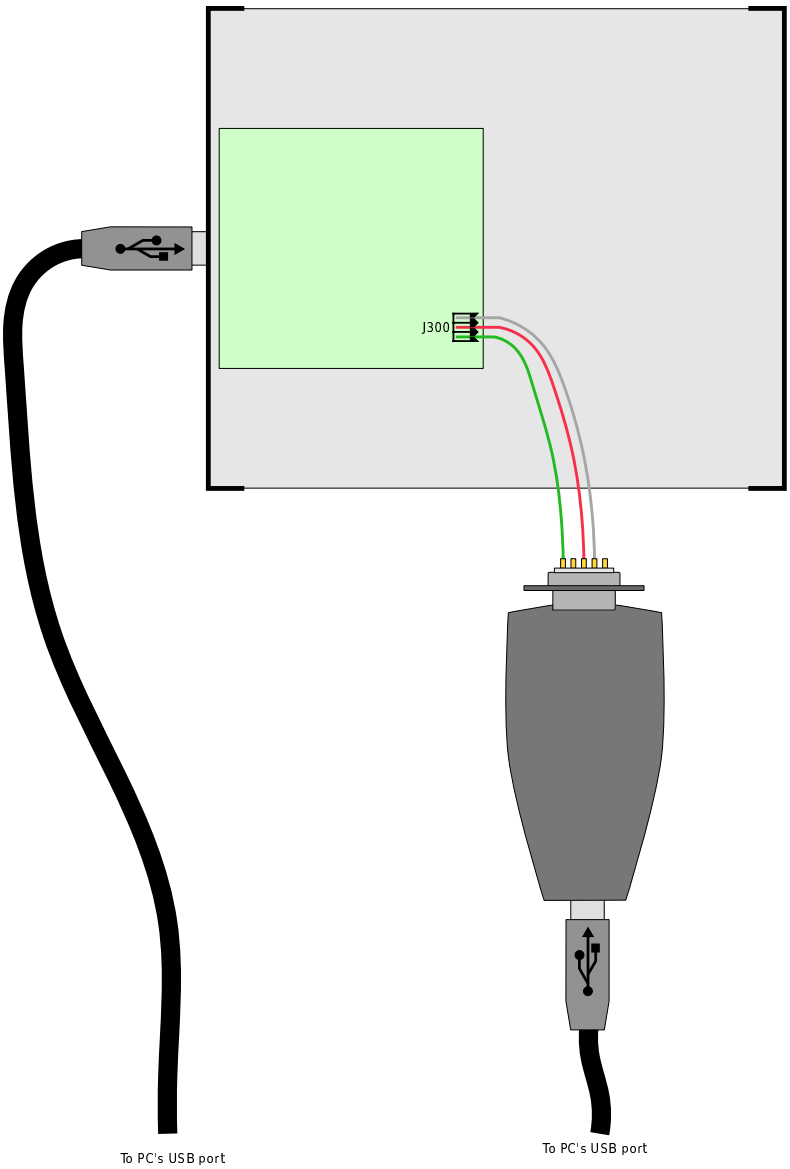


Figure 4: The setup for the serial loopback test.

2 Butterfly board

2.1 Making connections

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

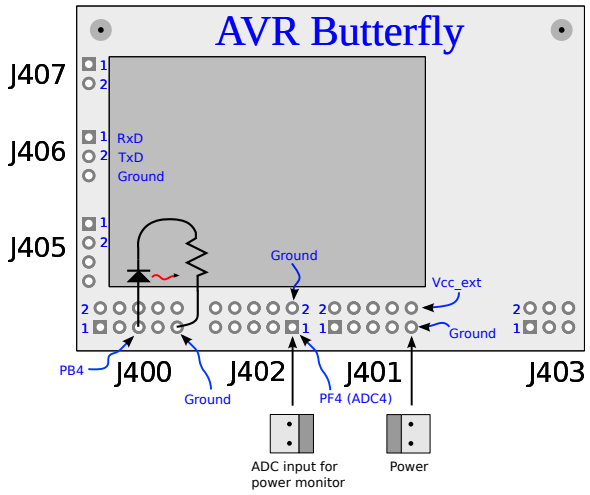


Figure 5: Connections to the AVR Butterfly

Figure 6 shows how the UART cable should be made.

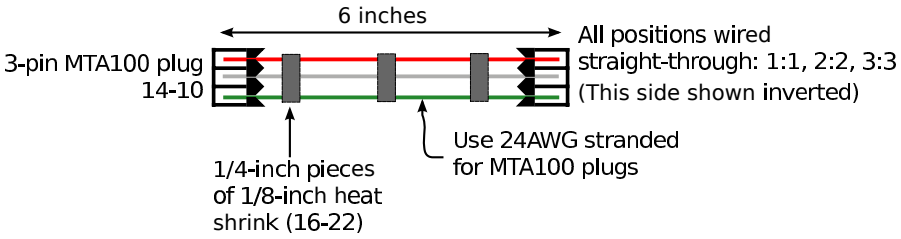


Figure 6: The UART cable connecting the Butterfly and USB boards.

Figure 7 shows how to make the power cable.

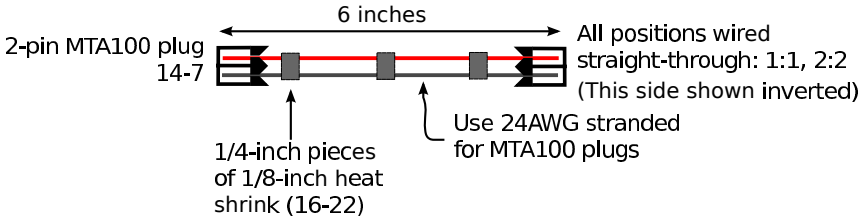
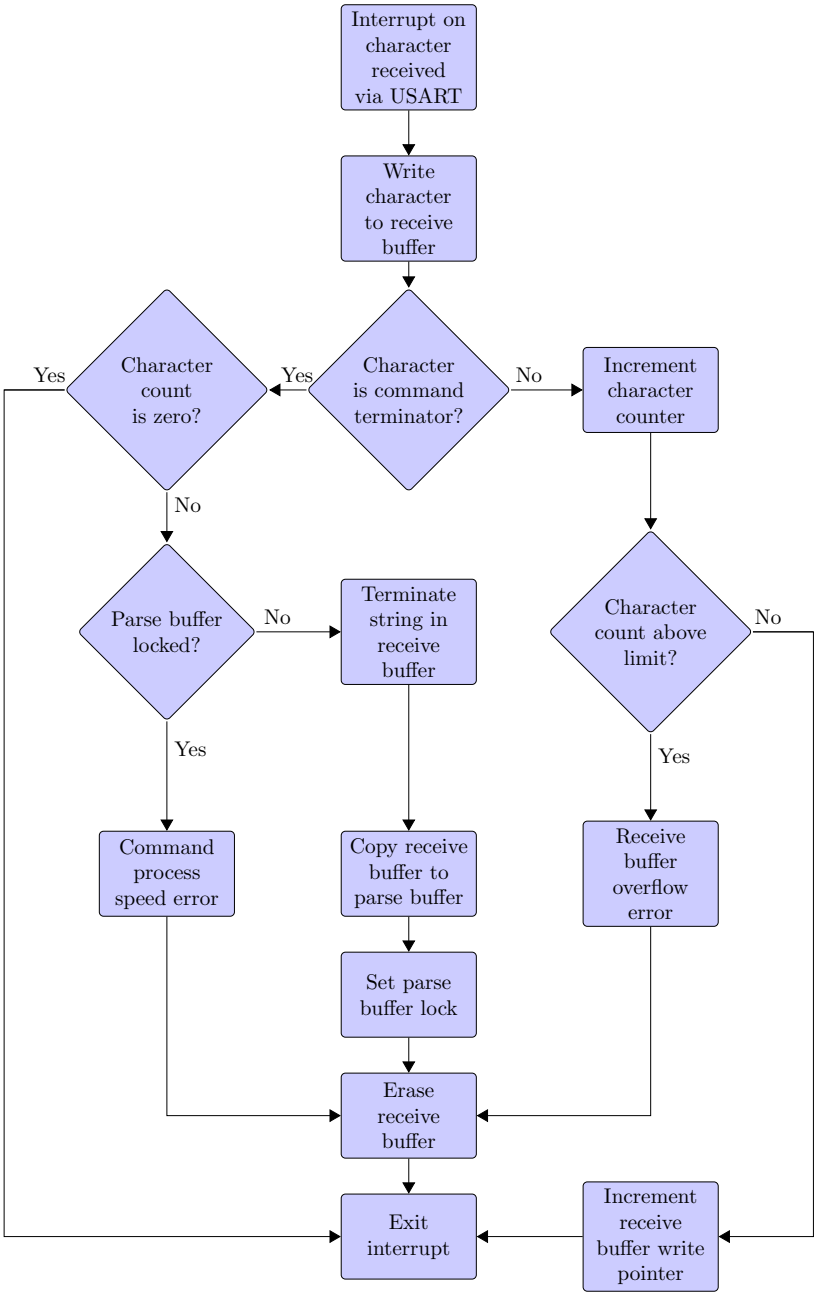


Figure 7: The power cable connecting the Butterfly and the USB boards. The cable should run from J400 on the USB board to the power connector shown in figure 5.

3 Firmware

3.1 Received character flow

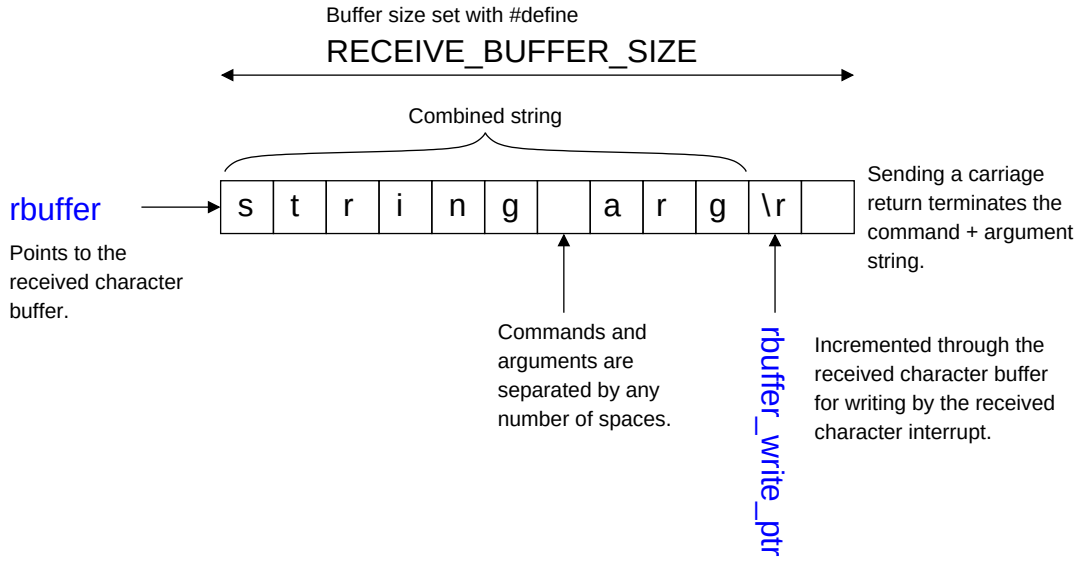
Receiving remote commands begins with receiving characters. Reception of these characters triggers an interrupt service routine (ISR), which handles them according to the flow shown in figure 8. The first step in this flow is loading the characters into the receive buffer.



recv_cmd_flow.tex

Figure 8: Program flow for processing characters received over the Butterfly's USART. Sending a command terminator (carriage return) will always result in an empty receive buffer. This is a good way to make sure there's no garbage in the buffer before writing to it.

Figure 9 illustrates the receive buffer loaded with a combined string. The buffer is accessed with a pointer to its beginning and a second pointing to the next index to be written. These pointers are members of the `recv_cmd_state_t` type variable `recv_cmd_state`. This is just style – I like to try to organize variables used in a flow by making them members of their own structure. Naming conventions aside, it's important to notice that there are no limitations on command or argument size imposed in this first step, provided that the total character count stays under the `RECEIVE_BUFFER_SIZE` limit.



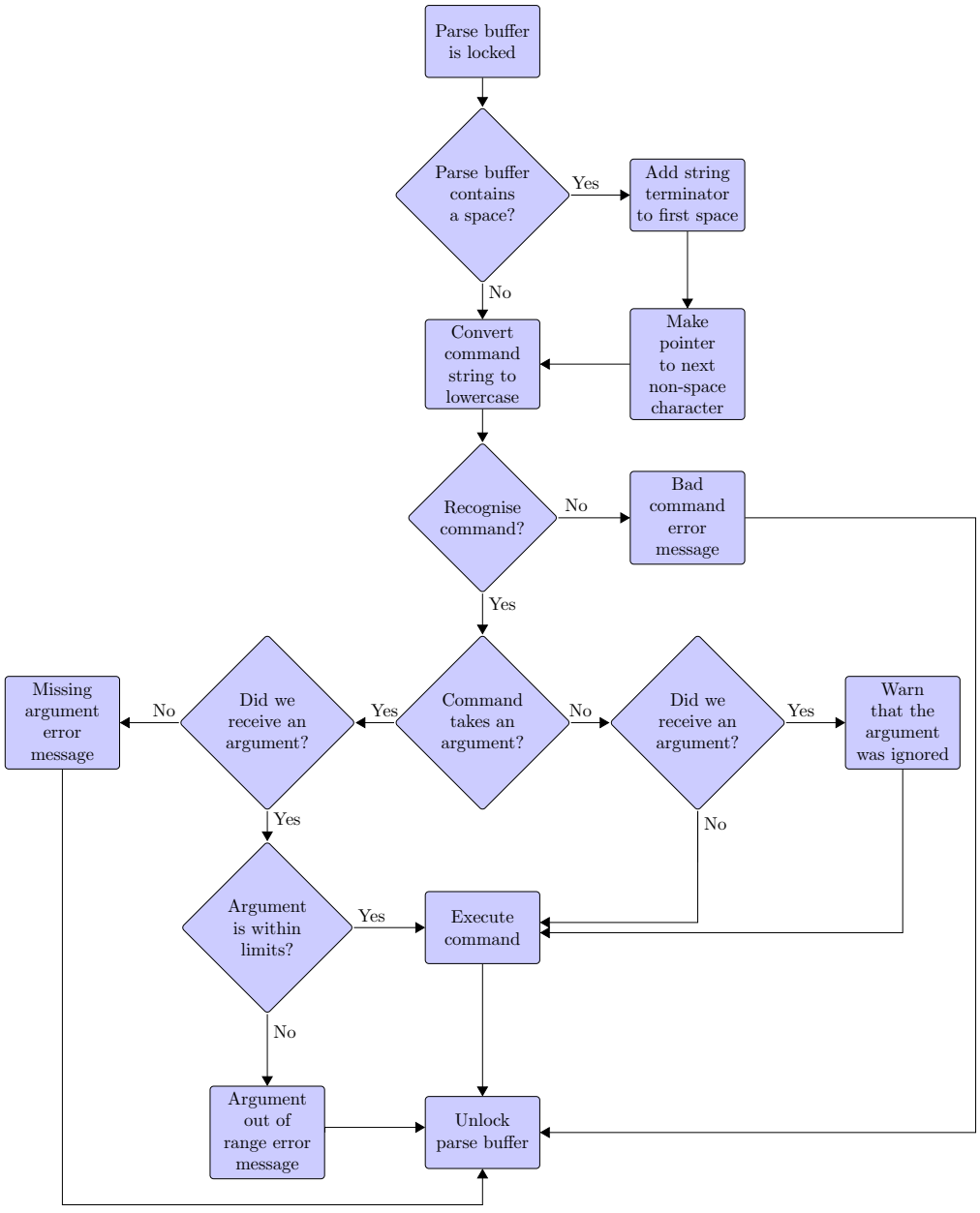
recbuffer.fig

Figure 9: The received character buffer and pointers used to fill it. There is no limit to the size of commands and their arguments, as long as the entire combined string and terminator fit inside `RECEIVE_BUFFER_SIZE`.

When a combined string in the receive buffer is finished with a carriage return, the string is copied over to a second buffer. I call this the parse buffer, since this is where the string will be searched for recognized commands and arguments. This buffer is locked until its contents can be processed to keep it from being clobbered by new combined strings. Sending commands faster than they can be processed will generate an error, and combined strings sent to a locked parse buffer will be dropped. The maximum command processing frequency will depend on the system clock and other system tasks. Not having larger parse or receive buffers is a limitation that puts this project at the hobby level. Extending these buffers to hold more than just one command would make the system more robust.

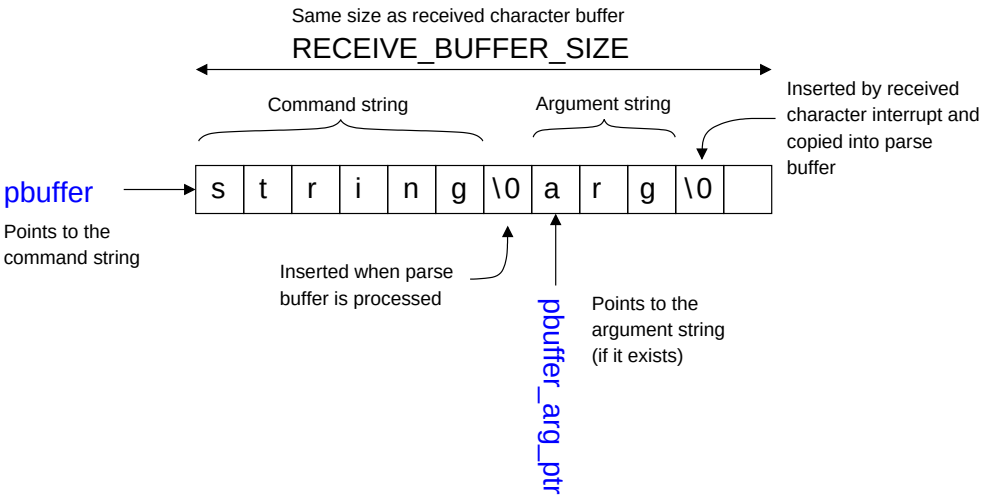
3.2 How remote commands are processed

After combined strings are copied from the receive to the parse buffer, the system separates them into command and argument strings using the flow shown in figure 10. Commands in the parse buffer are then separated from their arguments with a string terminator inserted into the first space between the two. As illustrated in figure 11, pointers to the beginning of the parse buffer and the beginning of the argument will then reference two separate strings. The first of these two, the command string, is converted to lowercase and compared with those in the command definition array to look for a match.



parse_cmd_flow.tex

Figure 10: Program flow for processing fully-formed commands. This processing step happens in the main loop, so it will only run when the system isn't busy doing something else. Notice that all incoming commands are converted to lowercase, so there's no case sensitivity.



prsbuffer.fig

Figure 11: The parse buffer with pointers to the command and argument strings. The single combined string copied from the receive buffer is made into two strings by inserting a string terminator.

3.3 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}
};
...
```

3.4 Logger functions

3.4.1 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.
```

3.5 Calibration factors

Data type	Description
`uint8_t`	Unsigned 8-bit integer
`int8_t`	Signed 8-bit integer
`unit16_t`	16-bit eeprom address

Table 4: The calibration factor structure will consist of two 8-bit integers and a base address for the calibration factor in eeprom memory. The eeprom memory locations will need to be two addresses apart.

Alphabetical command index

Internal command index