KUB manual

Tomas Härdin

November 15, 2018

Contents

List of Figures

List of Tables

1 Assembly

The instrument consists of a stainless steel skeleton into which a number of aluminium plates are screwed. All plates have a printed circuit board (PCB) attached via standoffs or screwed directly to the plate. The combination of board and plate is called a module. In this section we assume all PCBs have already been soldered and cleaned.

The following tools and materials are needed:

- Wire cutters
- Wire strippers suitable for 0.4 mm and 0.9 mm diameter wire (26 and 20 AWG). Strippers with fixed holes are recommended
- A small round or semi-round file
- Torque wrenches capable of 0.3 Nm, 0.8 Nm and 2.0 Nm with the following bits:
  - PH1 Phillips
  - 9IP Torx-Plus or T10 Torx
  - 4.5 mm hex socket
  - 5.5 mm hex socket
  - 7.0 mm hex socket
- Loctite 638
- Scotch-Weld 2216
- Soldering station
- Helping hands
- 60/40 or 63/37 leaded solder
- Solder flux
- White gloves for handling silver
- Two fine paintbrushes
- Citadel Chaos Black primer
- The Army Painter WP1101 Matt Black
- Electrolube SCP03G Silver Conductive Paint
- Toothpicks
- Heat gun
- 0.62 mm^2 / 20 AWG Alpha Wire 5856 WH005 PTFE wire
- 2.4 mm $\rightarrow$ 1.2 mm Kynar shrink tube

- Masking tape
- Maxon EC motor can vice (3D printed)

Each PCB can be screwed to its corresponding aluminium plate in any order, forming modules. The modules must be assembled in a certain order however, due to the way the connectors fit into eachother. The power4 module must be mounted first, then the cpu4 and credits modules can be mounted, and finally the fieldmill10 modules can be mounted. It's useful to mount the credits module last so that the mating of the fieldmill10 and cpu4 modules can be checked, and power-on tests performed.

In order to figure out what ID each DS18B20Z 1-wire temperature sensor has, the instrument should be powered on after each module has been installed and a temperature measurement performed (the 't' command in table 4). Since the cpu4 and power4 modules must be mounted at the same time in order to use the RS-485 interface, the two temperature sensors in that assembly must be distinguished somehow. This can easily done by warming one of them with a finger and watching the rise in temperature in one of the sensors. Take note of which sensor is which. This goes for all modules.

Use white cotton gloves when handling any silver parts, to prevent them from being contaminated. Each aluminium plate requires 16 silver plates M3 screws (?? mm long, Torx T10) to mount it to the skeleton, 96 total.

All M3 screws are torqued to 0.8 Nm and all M2 screws are torqued to 0.3 Nm.

1.1 power4

Parts needed:

- One (1) power4 PCB, soldered and cleaned
- One (1) silver plated bottom aluminium plate (2mm thick)
- Two (2) power4 aluminium standoffs
- Four (4) PEEK washers (top kind)
- Four (4) PEEK washers (bottom kind)
- Twentyfour (24) silver plated M3 screws, ?? length inluding head (Torx T10)
- Four (4) pieces of Alpha Wire 5856 WH005 PTFE wire (TODO: length)
- 2.4 mm $\rightarrow$ 1.2 mm Kynar shrink tube
- One (1) DE-9 male connector
- A set of DE-9 panel mounting screws, washers and nuts

If soldering the power4 module yourself, the pin headers should be inserted from the same side as the DS18B20Z IC and LEDs are on.

Cut four pieces of the PTFE wire to appropriate lengths (TODO: include in table), then solder them between the power4 board the the DE-9 connector according to table 1. It is easiest to solder the +28V wire last, since it crosses the other three wires, see figure 1. The pins on the power4 board

Figure 1: Photograph of power4 wiring. TODO: this should show the weaving of wire through both columns of pins.

are numbered from 1 starting from the side closest to cpu4. There are two columns of identical pins, the purpose of which is so that PTFE wire stripped around 15 mm can be threaded through and soldered to both holes, providing increased rigidity. Use Kynar shrink tubing to protect the solder points on the DE-9 side.

Description	**DE-9 pin**	**power4 pin**
+28V	1	1
GND	5	2
RS-485+ A	3	3
RS-485- B	7	4

Table 1: power4 wiring.

Fasten the DE-9 connector to the aluminium plate (inside or outside??) using the screws, washers and nuts. Torque to 0.8 Nm. Screw the standoffs to the plate, 0.8 Nm.

Place bottom PEEK washers on the standoffs, then the power4 board on the standoffs and finally the top wasters on top. Screw everything in place using M3 screws, 0.8 Nm.

Screw the finished module to the skeleton using M3 screws, 0.8 Nm.

1.2 cpu4

Parts needed:

- One (1) cpu4 PCB, soldered and cleaned
- One (1) silver plated cpu4 aluminium plate
- Two (2) cpu4/credits aluminium standoffs
- Four (4) PEEK washers (top kind)
- Four (4) PEEK washers (bottom kind)
- Twentyfour (24) silver plated M3 screws, ?? length inluding head (Torx T10)
- A Debian or Ubuntu GNU/Linux computer with build-essentials, avr-gcc and avrdude installed
- An STK600
- A USB to RS-485 converter
- A lab power supply
- Instrument converter cable (female DE-9 instrument connector → female RS-485 DE-9 connector(s) + banana plugs)

Be careful to use the aluminium plate with holes countersunk in the correct direction. The credits plates are mirror images of the cpu4 plates.

Screw the standoffs to the plate using M3 screws, 0.8 Nm.

Places bottom PEEK washers, board and top PEEK washers on the standoffs. Screw in place using M3 screws, 0.8 Nm.

Screw the finished module to the skeleton using M3 screws, 0.8 Nm.

1.2.1 Programming the bootloader

The bootloader can be programmed without the power4 board. Connect the cpu4 board to an STK600 using a 6-pin ISP (2x3 ribbon) cable, then connect the STK600 to the computer and turn the STK600 on. Run *make bootloader* in the code directory to build and program the bootloader. This will enable the crystal oscillator, and programming over RS-485.

After programming, make sure the crystal oscillator is working correctly by measuring the waveform on both its pins. The pin closest to the ISP connector should swing between $+0.28 \ldots +2.76$ V, ± 100 mV or so. This pin is what drives the ADCs' clocks. The pin further away from the ISP connector should have a larger swing, between $-0.32 \ldots +3.72$ V, ± 100 mV or so. Both waveforms should be roughly sinusodial.

1.2.2 Serial programming

The next step is to program the application via the RS-485 bus. To do this, mount the power4 and cpu4 modules to the skeleton and connect the instrument converter cable to the instrument. Connect the USB to RS-485 converter between the computer and the RS-485 end of the instrument cable. Set the power supply to somewhere between $18 \ldots 30$ V, disable its output and connect the banana plugs to it (mind the polarity). When you enable the lab supply's output, you have three seconds before the bootloader enters the application automatically. Type *make programserial* into a terminal in the code directory, but don't hit enter yet. When you are ready, enable the lab supply's output and then hit enter in the terminal. If everything went OK then the instrument should now have the application loaded into it. You can now proceed to checking system voltages!

1.2.3 Checking system voltages

After programming the application, the MCU will enable the ± 5 V and 24 V switching regulators after booting (3 seconds after power-up). Once it has done so, you should measure all system voltages to make sure they are reasonable. See table 2.

Voltage bus	**Min**	**Max**
+3.3V	+3.2 V	+3.4 V
+24V	+23 V	+25 V
+5V	+4.9 V	+5.1 V
-5V	−5.1 V	−4.9 V

Table 2: System voltage ranges.

You should also measure the resistances and voltages across the system voltage dividers (R20 through R27, R37 and R38), for calibrating the application. After such calibration, issue a 'v'

command (table 4), to ensure that the MCU's idea of what the system voltages are, is in line with the actual measured voltages.

Next up is distinguishing the two temperature sensors now in the system.

1.2.4 Distinguishing temperature sensors

Distinguish the two DS18B20Z temperature sensors now in the instrument by touching one of them, thus warming it, then issue a 't' command (table 4). Write down which ID corresponds to which of the two sensors.

1.3 fieldmill10

Parts needed:

- One (1) fieldmill10 PCB
- One (1) fieldmill aluminium top plate
- One (1) fieldmill shutter plate
- One (1) Maxon 349694 EC motor
- Stencil with a 15x5 mm hole
- One (1) ITR20001/T IR reflex coupler
- One (1) 0805 4.7 $k\Omega$ resistor
- Three (3) M2 screws, 6 mm length including head (Phillips PH1)
- Four (4) silver plated M3 screws, 10 mm length including head (Torx T10)
- Sixteen (16) silver plated M3 screws. Same as above??
- Four (4) M3 washers
- Four (4) M3 hex nuts
- One (1) M4 X 8/8 hex screw with 2 mm hole drilled through, preferably silver plated or silver painted if *someone* forgot to order silver plated M4 screws
- One (1) silver plated M4 flange nut

Before doing anything else, perform the following electrical tests using a multimeter

- Measure resistance from GND to every rail; +3.3V, +24V, +5V, -5V, +2.5V and -2.5V. Measure in both directions. Values should all be at least 1 kΩ
- Measure resistance from either side of R56 to GND. Measure in both directions. Value should be at least 8 kΩ

Figure 2: Motor wires soldered to their pads. Wire and insulation lengths marked.

Figure 3: Reflex coupler and motor can. Black paint also visible on motor can.

Applying power while these values are out of spec may cause PERMANENT DAMAGE to op-amps, DACs and/or linear regulators. In the worst case the +-5V regulator (TMR 6-2421WI) may also become damaged. Once checked and within spec the assembly process may continue.

fm10 voltage bus	**Min**	**Max**
+2.5V	+2.45 V	+2.55 V
-2.5V	−2.55 V	−2.45 V

Table 3: fieldmill10 voltage ranges.

Cut motor wires to 6-7 mm length. Strip so that 3 mm of insulation remains, with the wire stripper set to 0.4 mm. Twist and tin the ends of the wires. See figure 2.

Put the stencil on the rotor can, tape it down using masking tape and paint the area matt black. One way to do this is to spray primer into a glass or glazed ceramic cup then dip a fine paintbrush into the primer and paint it onto the can. DO NOT USE A PLASTIC CUP TO HOLD THE PRIMER - IT WILL MELT. Wait at least 30 minutes for the primer to dry, then apply the matt black paint on top of it. The black painted area should cover roughly a 90 arc of the edge of the can. See figure 3.

If the motor doesn't fit in the central hole in the PCB, use a small semi-round file to grind away some of the PCB material until the motor fits.

Put the Maxon EC motor in the motor hole, then shim the PCB + motor combination up from the bottom so that the motor can lie flush with the PCB while working on it. Solder the five motor wires to the motor wire pads on the PCB.

Bend and cut the IR reflex coupler leads so that the reflex coupler looks directly at the EC motor's rotor can when inserted into its 2x3 socket (IR2 reference on PCB). See figure 3. Bending then trimming all leads to the same length as the shortest lead will accomplish this, The resulting lead length should be 7 mm from the bottom of the reflex coupler. Insert the reflex coupler into its socket. The gap between the top of the 2x3 socket and the bottom of the reflex coupler should be 2 - 3 mm. Solder the reflex coupler into the socket. Use plenty of flux so the leads and socket are guaranteed to be soldered together. Some extra solder can be applied to the 2x3 socket pads at this point. Carefully test that pulling on the reflex coupler doesn't cause it to come out. Check continuity with a multimeter, and that adjacent pins aren't shorted.

Replace R56 with a 4.7 $k\Omega$ 0805 resistor.

Screw the motor + PCB assembly to the aluminium plate using the M2 screws for the motor and silver plated M3 screws, washers and nuts for the PCB. First screw in the screws lightly, then tighten using the torque wrench. Use 0.3 Nm for the M2 screws and 0.8 Nm for the M3 nuts.

Use a toothpick to paint the motor bearing with SCP03G silver paint. Be careful not to get silver paint on the motor axis. There should be a slight resistance in the bearings which will go away after the first initial minutes of operation.

Use Loctite 638 to glue the M4 screw to the motor axis. Mount the screw so that the head points downward, toward the instrument, and so that it is flush with the inner ring of the motor bearing.

When the glue has dried, put the motor can in the specially shaped vice. Put the shutter plate on the M4 screw and fasten it using the M4 flange nut. Torque to 2.0 Nm. Drop some SCP03G silver paint into the top of the screw so that it gets a good electrical connection to the motor shaft.

Screw each module to the skeleton using M3 screws, 0.8 Nm.

Perform a power-on test. Check that the +5V, -5V, +3.3V, +2.5V and -2.5V rails are within spec (tables 2 and 3). Measure the output of every channel while VGND = 0 V. The average voltage of each output should be within ±200 mV of VGND.

Take a temperature reading after each fieldmill10 module has been inserted, and make a note of the new sensor ID each time and what module it corresponds to.

Start the motors and run them for at least five minutes. Measure the resistance between motor axis and instrument ground. The resistance should be less than 10 Ω.

1.4 credits

- One (1) credits PCB, soldered and cleaned
- One (1) silver plated credits aluminium plate
- Two (2) cpu4/credits aluminium standoffs
- Four (4) PEEK washers (top kind)
- Four (4) PEEK washers (bottom kind)
- Twentyfour (24) silver plated M3 screws, ?? length inluding head (Torx T10)

Be careful to use the aluminium plate with holes countersunk in the correct direction. The credits plates are mirror images of the cpu4 plates.

Screw the standoffs to the plate using M3 screws, 0.8 Nm.

Places bottom PEEK washers, board and top PEEK washers on the standoffs. Screw in place using M3 screws, 0.8 Nm.

Screw the finished module to the skeleton using M3 screws, 0.8 Nm.

Perform a power-on test and a temperature test. Take note of the new temperature sensor ID.

2 Command and response summary

All commands are human-readable, start with a single ASCII character and are terminated with a line ending. Comments may be added by inserting a hash sign ('#'). This causes the rest of the line to be ignored (characters are consumed and discarded until end-of-line). Mistakes can be corrected with backspace (BS, ASCII code 8) or delete (DEL, ASCII code 127), both of which are treated as a backspace. Line reading also understands escape (ESC, ASCII code 27), which aborts the current command.

Parameter parsing is handled by sscanf(), which allows for the same command character to take a varying number of parameters. An example of this is the 'M' command which exists in two-parameter and three-parameter forms:

```
M0 10        # Set speed of motor 0 to 10
M10 10 10    # Set speed of all motors to 10
```

Line endings can either be carriage return ('\r', ASCII code 13) or linefeed ('\n', ASCII code 10), but never both in the same line. In other words both Unix and Mac line endings are OK, but Windows line endings ("\r\n") are not. This ensures that *echo*, *minicom* and *picocom* work as expected. Output from the instrument is however terminated by Windows line endings ("\r\n"), in order to play nice with *minicom* and *picocom*. An ESC ('\x1B', ASCII code 27) anywhere in a line aborts that command. The instrument will respond to ESC immediately. There is no need to terminate ESC with newline.

Output from the instrument is delimited into frames. A frame starts with the string "BUSY\r\n" and ends with the string "READY\r\n". Each frame contains one or more sections, each of which is started by a star (*) followed by the name of the section terminated by \r\n. For example, the reply to the "M10 10 10" command example given earlier is of type MTR_PWM and would therefore be:

```
BUSY
*MTR_PWM
10 10 10
READY
```

Tables 4 and 5 summarize all commands and their parameters. More detailed descriptions of each command and response are given in the sections that follow.

Char	Description	Response
v	Measure system voltages	VOLTAGES
V	Enable 24V and +-5V	VOLTAGES
B	Disable 24V and +-5V	VOLTAGES
m	Read motor PWMs	MTR_PWM
K	Set motor PWMs to 50%	MTR_PWM
o	Read VGND DAC values	VGNDS
r	Measure motor speeds in RPM	MTR_SPD
1	List 1-wire device ROMs	ONEWIRE
!	Search for 1-wire devices	ONEWIRE
t	Measure temperatures	TEMPS
c	Read clock in cycles	CLOCK
U	UNLOCK + STANDBY all ADS131A04	ADC_REGS
q	Read ADS131A04 registers	ADC_REGS
e	Read measurement config	CONFIG
W	WAKEUP + LOCK, start measuring	SAMPLES [...]
S	Reboot	*N*one
?	Print help	INFO
\x1B (ESC)	Stop measurement, abort current recvline(). Can occur anywhere in a line, does not have to be terminated with newline.	ESC

Table 4: Command table, parameterless commands

Char	Parameter syntax	Description	Response
M	id pwm	Set motor PWM	MTR_PWM
M	pwm0 pwm1 pwm2	Set all motor PWMs	MTR_PWM
O	id vgnd	Set VGND DAC value	VGNDS
O	vgnd0 vgnd1 vgnd2	Set all VGND DAC values	VGNDS
Q	id addr val	Set ADS131A04 register	ADC_REGS
E	frame_size gap [num_packets]	Configure measurement	CONFIG
		Read CPU regs ($0000 - $00FF)	
		Write CPU regs ($0000 - $00FF)	
		Read RAM ($0100 - $FFFF)	
		Write RAM ($0100 - $FFFF)	
		Read EEPROM ($000 - $FFF)	
		Write EEPROM ($000 - $FFF)	
		Read ROM ($00000 - $1FFFF)	
		Read fuses	
C	cycles	Set clock in cycles	CLOCK
w	cycles	Wait given number of cycles	INFO
		Start measurement	

Table 5: Command table, commands with parameters

2.1 Short example

To get the instrument into a useable state at least the following commands have to be issued:

- Start motors ('M')
- RESET+UNLOCK+STANDBY ADS131A04s ('U')
- Configure ADS131A04s ('Q')
- Configure measurement ('E')
- WAKEUP+LOCK ADS131A04s ('W')

What follows is a slightly edited session (for brevity) with only one fieldmill mounted. First just the commands are listed, then the full session (input and output):

```
M1 1023      # Motor 1 at full speed
U            # UNLOCK ADCs
Q1 0F 01     # Activate channel 1 on ADC 1
E100 0 3     # 100 frames per packet, no gap, 3 SAMPLES
W            # WAKEUP ADCs, output SAMPLES
```

Full session:

```
BUSY
*INFO
Hello, Earth!
READY
M1 1023     # Motor 1 at full speed
BUSY
*MTR_PWM
0 1023 0
READY
U           # UNLOCK ADCs
BUSY
[...]
*ERROR
ADC 0 seems to be offline
*INFO
ADC 1 up
[...]
*ERROR
ADC 2 seems to be offline
*ADC_REGS
0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
1 04 03 00 00 00 00 00 01 00 00 00 60 3c 08 86 00 00 00 00 00 00
2 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
READY
Q1 0F 01    # Activate channel 1 on ADC 1
BUSY
*ADC_REGS
0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
1 04 03 20 00 00 01 00 01 00 00 00 60 3c 08 86 01 00 00 00 00 00
2 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
READY
E100 0 3    # 100 frames per packet, no gap, 3 SAMPLES
BUSY
*INFO
bytes = 420, cpc = 25600, pc = 1
cycles_out =       276172
 cycles_in =      2560000 (OK)
*CONFIG
100 0 3
READY
W           # WAKEUP ADCs, output SAMPLES
BUSY
*INFO
Measurement started
READY
BUSY
```

```
*SAMPLES
[...]
```

3 Responses

Responses given before commands since many commands share response format.

3.1 INFO

Informative text. Display contents in text window or terminal. Should be displayed with normal colors. Example:

```
BUSY
*INFO
Starting temperature conversion...
READY
```

3.2 WARNING

Warning. Should be displayed with a yellowish or maybe orange color.

```
BUSY
*WARNING
Instrument issues no warnings currently,
but may in the future.
READY
```

3.3 ERROR

Some kind of error. Should be displayed with a reddish color.

```
BUSY
*ERROR
One or more of PWMS 1111, 2222, and 3333
is greater than MOTOR_TOP = 1023
READY
```

3.4 VOLTAGES

TODO

3.5 MTR_PWM

PWM value for each of the three motors, as integers between 0 - 1023. These correspond roughly to 0 - 6000 RPM, but not exactly. For more information see EC20 datasheet. Example response after issuing a "K" command:

```
BUSY
*MTR_PWM
511 511 511
READY
```

3.6 VGNDS

10-bit DAC values for each MAX504 (0 - 1023), which are used for the input stages' virtual grounds (VGND). The MAX504s are in bipolar configuration, so a value of 0 gives -2.048 V out, 512 gives 0 V out and 1023 gives 2.044 V out. In other words the output voltage is $-2.048 + value * 0.004$. See also Table 3 in the MAX504 datasheet.

The instrument is initialized so that all MAX504s output zero volts. Example response to an 'o' command after initialization:

```
BUSY
*VGNDS
512 512 512
READY
```

3.7 MTR_SPD

Motor speeds in RPM, and the number of tachometer impulses detected. For a sensible output at least two tachs must have been seen for each motor. Exact format isn't finalized, so display output as if it were INFO for now.

```
BUSY
*MTR_SPD
Motor 0: 0 tachs (0 RPM)
Motor 1: 1 tachs (0 RPM)
Motor 2: 2 tachs (1234 RPM)
READY
```

3.8 ONEWIRE

1-Wire devices. One line per device containing a 64-bit hexadecimal ROM string (16 characters).

```
BUSY
*ONEWIRE
28d09948090000ec
286a1a690900005e
28ad7548090000c5
READY
```

3.9 TEMPS

Temperature readings. One line per DS18B20Z device with 64-bit hexadecimal ROM and decimal temperature values in degrees Celsius. Range is -40..+125 with a resolution of 1/16 degrees Celsius.

```
BUSY
*TEMPS
28d09948090000ec 24.12
286a1a690900005e 24.62
28ad7548090000c5 -18.56
READY
```

3.10 CLOCK

Current time in CPU cycles as a 64-bit decimal value. May roll over in theory, but it would take 79,000 years unless the user specifically set the clock to something close to the maximum value (18446744073709551615).

```
BUSY
*CLOCK
3702994144
READY
```

3.11 SAMPLES

Sample data. This response is binary. It is possible, but very unlikely, that the binary data contains the string "READY\r\n". To avoid trouble it is best to compute the exact number of bytes in the sample data by following listing 5 on page 21.

```
BUSY
*SAMPLES
[sample_packet_s]
READY
```

3.12 ADC_REGS

The contents of all registers in all ADCs. Three lines, each beginning with the integer ID of the ADC. After this there are twenty-one (21) hexadecimal values corresponding to registers 00h - 14h. See ADS131A04 datasheet for more information.

This example has one fieldmill installed in position 1. Therefore only the middle line (ID=1) has valid data:

```
BUSY
*ADC_REGS
0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
1 04 03 00 00 00 00 00 01 00 00 00 60 3c 08 86 00 00 00 00 00 00
2 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
READY
```

The first two registers (ID_MSB and ID_LSB) can be used to detect that an ADC is working, since they are read-only and of known value. All our chips have ID_MSB = 04h (for ADS131A*04*) and ID_LSB = 03h (revision 3).

3.13 CONFIG

Measurement configuration. Three integers. The first integer is the number of frames in each SAMPLES packet. Zero means no configuration yet, or there was some error with the attempted configuration. In short, if the first integer is zero then WAKEUP ('W') cannot be issued. The second iteger is the gap in frames between SAMPLES packets. It may be zero, and must be non-zero if the total sample bitrate is higher than the UART bitrate. So a non-zero value provides a longer pause between SAMPLES packets. The third integer is the number of SAMPLES packets that will be output. A value of 65535 means SAMPLES will be output indefinitely.

The following example has 100 frames per packet, no gap between packets and will run indefinitely:

```
BUSY
*CONFIG
100 0 1000
READY
```

4 Commands

4.1 Reboot ('S')

Takes no parameters and no prisoners. Hangs the user program, inducing the WDT to cause a software reset. After the reset the CPU enters the bootloader. This allows a serial programmer to keep sending 'S' to get the device into a known state, since the AVR109 bootloader merely responds to 'S' with 'AVRBOOT'. If no programming is performed then the user program is re-entered after the bootloader wait time has elapsed (typically three seconds).

```
S
S
AVRBOOT[three second delay]BUSY
*INFO
Hello, Earth!
READY
```

4.2 Set motor PWMs ('M')

Set one or more OCR1x. PWM values must be between 0 - 1023.

4.2.1 Two parameter form (id pwm)

Set PWM duty rate for motor with given ID (0..2). Example:

```
M1 800
BUSY
*MTR_PWM
0 800 0
READY
```

4.2.2 Three parameter form (pwm0 pwm1 pwm2)

Set PWM duty rate for all three motors in one go.

```
M200 400 600
BUSY
*MTR_PWM
200 400 600
READY
```

4.3 Set VGND DAC values ('O')

Set one or more inputs to the MAX504s. Values must be between 0 - 1023.

4.3.1 Two parameter form (id vgnd)

Set DAC value for MAX504 with given ID (0..2). Example:

```
O1 900
BUSY
*VGNDS
512 900 512
READY
```

4.3.2 Three parameter form (vgnd0 vgnd1 vgnd2)

Set DAC value for all MAX504s:

```
M300 400 500
BUSY
*VGNDS
300 400 500
READY
```

4.4 UNLOCK + STANDBY all ADS131A04 ('U')

Bring ADCs out of reset, halting any ongoing conversions and allowing registers to be modified.

4.5 Set ADS131A04 registers ('Q')

The 'Q' command sets the value of one register in one of the ADCs. After the 'Q the line must contain three values: an integer specifying the ADC ID, a hexadecimal value specifying the register address and a hexadecimal value specifying the value. In C formatting parlance it should be "%i %x %x".

The following example sets COMP_TH = 111b and enables fixed word size (6 device words per frame regardless of ADC_ENA):

```
Q1 0B 67
BUSY
*ADC_REGS
0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
1 04 03 00 00 00 00 00 01 00 00 00 67 3c 08 86 00 00 00 00 00 00
2 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
READY
Q1 0C 3E
BUSY
*ADC_REGS
0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
1 04 03 00 00 00 00 00 01 00 00 00 67 3e 08 86 00 00 00 00 00 00
2 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
READY
```

4.6 Configure measurement ('E')

Takes between two and four integer arguments. The first integer sets the number of frames per packet. The second integer sets the gap between packets. The third integer, if present, sets the number of SAMPLES packets that will be captured and output (0..65534). The fourth integer, if present, sets the sample format to one of the following:

- 0: Signed 24-bit samples
- 1: Signed 8-bit samples. Full (quantized) 24-bit samples can be recovered by multiplying values by 2^{sample_shift}

Given values will be sanity checked.

```
E100 0 1000
BUSY
*INFO
bytes = 420, cpc = 25600, pc = 1
cycles_out =       276172
 cycles_in =      2560000 (OK)
*CONFIG
100 0 1000
READY
E10000 0
```

```
BUSY
*ERROR
sample_data_size = 30000 larger than maximum 4096
READY
e
BUSY
*CONFIG
0 0 65535
READY
```

4.7 Wakeup + LOCK, start measuring ('W')

Sends WAKEUP and LOCK to all ADCs configured with ADC_ENA != 0, then outputs SAMPLES until the user sends ESC or 'U'.

```
W
BUSY
*SAMPLES
[sample_packet_s]
READY
BUSY
*SAMPLES
[sample_packet_s]
READY
[...]
^[
BUSY
*ESC
READY
```

5 Listing

```
// Overview of the IQ demodulated packet format:
//
// +---------------------------------------+
// | Header                (6  bytes)      |
// +---------------------------------------+
// | "TEMP"                (4 bytes)       |
// | Temperature values    (variable size) |
// +---------------------------------------+
// | "VOLT"                (4 bytes)       |
// | Voltage ADC values    (variable size) |
// +---------------------------------------+
// | "FMIQ"                (4 bytes)       |
// | Demodulated FM data   (variable size) |
// +---------------------------------------+
//
// TEMP, VOLT and FMIQ are convenient markers and
// always present. The amount of data between markers
// depends on values in the header.
// The size of the packet can be summarized as:
//
//   packet_size =
//     6 +
//     4 + num_temps*4 +
//     4 + popcount(adc_mask)*2 +
//     4 + popcount(fm_mask)*57
//
// The largest this will typically get (6 temperature
// readings, 5 voltages, 3 FM IQ data structs) is 145 B

typedef struct square_demod_header_s {
  uint8_t version;      // format version (5)
  uint16_t num_frames;  // number of frames sampled

  // Number of DS18B20Z outputs (0..6)
  // Each output is a temperature_s (4 bytes)
  uint8_t num_temps;

  // Voltage ADC channel bitmask.
  // Used for reporting system voltages.
  // Bit n -> ADCn value present (2 bytes).
  // Currently only channels ADC0..4 are used.
  // All values are 10-bit single-ended conversions
  // with 2.56 V reference, transmitted in 16-bit ints.
  // Values are ADCL + 256*ADCH.
```

```
  // See the implementation of the 'v' command in app.c
  // for how to convert these values.
  uint8_t volt_mask;

  // Fieldmill bitmask
  // Bits 0..2 -> corresponding FM is enabled
  // See fm_s struct
  uint8_t fm_mask;
} square_demod_header_s;

// TEMP

// Temperature reading structure.
// Since temperature conversions take around 750 ms
// not every packet will have temperatures.
typedef struct temperature_s {
  // Bytes 1-2 of DS18B20Z ROM is enough to uniquely
  // identify the ones we have:
  //
  //  286a1a690900005e -> 6a 1a
  //  28f72a6909000021 -> f7 2a
  uint8_t   rom12[2];

  // Temperature in degrees Celsius * 16
  int16_t   temp;
} temperature_s;

// VOLT

// Inbetween VOLT and TEMP are the system voltages
// 10-bit values contained in 16-bit integers

// FMIQ

typedef struct {
  uint8_t stat_1;   // STAT_1 (or'd during capture)
  uint8_t stat_p;   // STAT_P
  uint8_t stat_n;   // STAT_N
  uint8_t stat_s;   // STAT_S
} stat_1pns_s;

// There are popcount(fm_mask) instances of this
typedef struct fm_s {
  uint16_t discard; // how many samples were discarded
                    // before the first tach
  uint8_t num_tachs;// number of tachometer impulses
```

```
  uint16_t NQ[4];   // number of frames collected in
                    // each quadrant
  int16_t IQ[3][2]; // IQ data (IQIQIQ)

  // ADS131A04 registers:
  stat_1pns_s stat;

  // min/max of all samples in each channel
  int16_t minmax[4][2];

  // mean value of samples within tachometer interval
  int16_t mean[4];

  // mean absolute value of samples within tachometer
  // channel 4 excluded since it would equal mean[3],
  // due to wiring
  uint16_t mean_abs[3];

  // MAX504 setting (0..1023)
  uint16_t vgnd;
} fm_s;


// Overview of the sample packet format:
//
// +---------------------------------------+
// | Header                (21 bytes)      |
// +---------------------------------------+
// | "TEMP"                (4 bytes)       |
// | Temperature values    (variable size) |
// +---------------------------------------+
// | "TACH"                (4 bytes)       |
// | Tachometer timestamps (variable size) |
// +---------------------------------------+
// | "SAMP"                (4 bytes)       |
// | Sample data           (variable size) |
// +---------------------------------------+
//
// The size of the packet can be summarized as:
//
//   packet_size = 21 +
//     4 + num_temps*4 +
//     4 + sum(num_tachs)*3 +
//     4 + num_frames*popcount(channel_conf)*
//         bytes_per_sample(sample_fmt)
//
```

```
// A more detailed view follows, in the form of C
// structs which are shared between code and manual.

// Sample packet header. Fixed size.
typedef struct sample_packet_header_s {
  uint8_t   version;      // format version (4)
  __uint24  first_frame;  // timestamp of first frame
  uint8_t   num_temps;    // DS18B20Z outputs (0..6)
  uint16_t  num_tachs[3]; // tach impulses per channel
  uint16_t  num_frames;   // number of frames
  uint16_t  gap;          // gap between packets
  uint16_t  channel_conf; // channel bitmap. 3 nybbles

  // Sample format
#define SAMPLE_FMT_S24 0  // raw 24-bit
#define SAMPLE_FMT_S8  1  // signed 8-bit with shift
  uint8_t   sample_fmt;

  // If SAMPLE_FMT_S8, multiply each sample by
  // (1<<sample_shift)
  uint8_t   sample_shift;

  // If gap was insufficient than overflow says many
  // frames had to be thrown away. In other words
  // overflow represents "unplanned" gap.
  // Value is capped at 255, which may mean "255 or more".
  uint8_t   overflow;

  // Prescaler used for timers. Multiply timers
  // by this value to get time in clock cycles.
  // Typical values: 1 or 8.
  uint8_t   prescaler;
} sample_packet_header_s;

// Sample packet itself is variable size.
typedef struct sample_packet_s {
  // Header defined above
  sample_packet_header_s header;

  // Temperatures with structures defined above.
  // A reading like:
  //
  //  286a1a690900005e 23.12
  //  28f72a6909000021 -3.87
  //
  // will be encoded as:
```

```
  //
  //  6a 1a 71 01 f7 2a c2 ff
  //
  char temp_marker[4];  //TEMP
  temperature_s *temps;

  // Tachometer timestamps.
  // Number of entries is sum(num_tachs).
  // Values are stored one channel after the other,
  // NOT interleaved. If num_tachs = {3, 5, 4} then
  // the order will be like this:
  //
  //   0 0 0 1 1 1 1 1 2 2 2 2
  //
  // Keep in mind num_tachs can be zero for one or more
  // channel. num_tachs = {3, 0, 4} would look like:
  //
  //   0 0 0 2 2 2 2
  //
  char tach_marker[4];  //TACH
  __uint24  *tachs;

  // Sample data is stored as a series of frames.
  // Each frame is built up of samples, and the number
  // of samples is the same as the number of bits in
  // channel_conf. Or: popcount(channel_conf).
  // The order of the samples is the same as the order
  // of ones in channel_conf.
  //
  // If all three ADCs are used, but only the first
  // three channels in each ADC, then channel_conf will
  // be "0000 0111 0111 0111" (most significant bit
  // first). Each frame will consist of 9 samples.
  //
  // The size of each sample depends on sample_fmt.
  // If 24-bit samples are used then the total amount
  // of sample data is:
  //
  //   num_frames * popcount(channel_conf) * 3  (bytes)
  //
  // In the example above, if we have 1000 frames then
  // the size of the sample data is 1000*9*3 = 27000 B.
  char samp_marker[4];  //SAMP
  uint8_t   *sample_data;

} sample_packet_s;
```

```
typedef struct {
  __uint24 t;       // timestamp, in cycles (mod 2^24)
  uint8_t num_tachs;// number of tachometer impulses
  uint16_t N;       // number of frames in tach interval
  int16_t IQ[3][2]; // IQ data (IQIQIQ)
} capture_entry_s;


typedef struct {
  // ADS131A04 registers ($00 .. $14)
  // Collected at the end of each block so that STAT_* ($02 .. $05) applies to
  // the block just captured.
  uint8_t ads131a04_regs[21];

  // Number of frames collected in each quadrant, across the entire block
  uint32_t NQ[4];

  // Min/max of all samples in each channel across the entire block
  int16_t minmax[4][2]; //min,max,min,max,...

  // Mean value of samples within tachometer interval
  // Only computed for the last capture of each FM
  int16_t mean[4];

  // Mean absolute value of samples within tachometer intervals
  // Channel 4 is excluded since its voltages are always positive, thus its
  // values would equal mean[3]
  // Like mean, only computed for the last capture of each FM
  uint16_t mean_abs[3];

  // Motor OCR1A/B/C (PWM) settings
  uint16_t OCR1n;
} fm_stat_s;

typedef struct {
  uint8_t version;  // version (0)
  uint32_t f_cpu;   // CPU clock in Hz
  uint64_t t;       // full timestamp at start of block
  uint8_t fm_mask;  // bits 0..2 = corresponding FM enabled

  // Number of frames collected for each entry
  // entris[x].N <= num_frames
  uint16_t num_frames;
```

```
  //TODO: temperatures and voltages

  // Statistics for each FM. Zeroed for all disabled FMs.
  fm_stat_s stats[3];

  // How many FM capture rounds there are for each VGND setting
  // Unbiased data starts at round 2*vgnd_rounds*popcount(fm_mask)
  uint8_t vgnd_rounds;

  // MAX504 settings (0..1023)
  // Each VGND is toggled negative and positive in turn, meaning that for
  // fm_mask = 7 each VGND is set thusly:
  //
  // VGND0 VGND1 VGND2
  // -     0     0    For vgnd_rounds rounds for each of these
  // +     0     0
  // 0     -     0
  // 0     +     0
  // 0     0     -
  // 0     0     +
  // 0     0     0   For the rest of the block
  uint16_t vgnd_zero;   // 0 level
  uint16_t vgnd_minus;  // - level
  uint16_t vgnd_plus;   // + level

  // If instrument is not interrupted, the following should hold:
  //   nentries > 2*vgnd_rounds*popcount(fm_mask)^2
  //   nentries = 0 (mod popcount(fm_mask))
  uint8_t nentries;

  // Entries for each FM in each round are stored in round-robin order
  // For example, if fm_mask = 7:
  //   0, 1, 2, 0, 1, 2, etc.
  // Truncated to nentries*sizeof(capture_entry_s) bytes
  capture_entry_s entries[255];
} capture_block_s;
```