Raw Panel Protocol V2

Revised and expanded by Kasper Skårhøj, July 2022

Introduction

SKAARHOJ ApS is a manufacturer of control panels for the media production world, especially the broadcast and AV markets for live production. Our control panels are characterized by their intelligent nature: they are able to speak the protocols of the devices they connect to. However, they are also able to act as *passive* panels that rely on being connected to a host system that drives the application logic. For this purpose, Raw Panel Protocol was developed in 2017 and has grown to become a significant factor in how our panels are being used. Raw Panel is often a relevant choice for third party integrators.

Here are some facts about Raw Panel Protocol:

- IP Network based communication between panel and host system
- Event based - a panel sends triggers from hardware components when activated
- Feedback for LEDs and displays are returned from the system
- The panel can be a TCP client or server (preferred)
- Messages can be ASCII or binary encoded
- Panels are self-describing by offering their topology in JSON and SVG formats
- Discoverable on the network via mDNS/zeroconf

Today Raw Panel is in fact the internal exchange protocol for SKAARHOJ's two platforms, UniSketch and Blue Pill. UniSketch panels support V1 (ASCII only) when they are running a configuration that maps Raw Panel actions to the hardware components. Blue Pill panels use Raw Panel natively and offers a Raw Panel server on a port or internal socket. On Blue Pill panels the Raw Panel Protocol is V2 and the exchange format is preferred to be via binary encoded protobuf messages although ASCII is supported too.

Examples

Here are some examples of how communication on the Raw Panel Protocol works in ASCII mode. In ASCII mode, the messages are human readable text delimited by line breaks.

Examples (in bold) sent from a panel to a host system:

- **HWC#35.4=Down** - Button number 35 was pressed down on the bottom edge (4)
- **HWC#42=Speed:136** - A joystick with number 42 was moved to the right, position 136
- **HWC#25=Abs:768** - Fader number 25 was moved to about 3/4 from the top
- **HWC#38=Enc:2** - Encoder number 38 was moved two pulses clock wise

Examples (in bold) sent from a host system to a panel:

- **HWC#38=4** - Turn on the LED for hardware component number 38
- **HWCc#38=135** - Set purple color for the LED of component number 38
- **HWCt#38=-3276|8||Float3** - Set title in the display to "Float3" and render the integer -3267 as a floating pointer number with 3 digits:
  

These examples outline how easy it is to work with Raw Panel for the most central communication between a panel and a host system. There are many more commands available and they are documented on the next pages.

Server and Client mode

Originally, Raw Panel supported only a client mode where the panel would be a TCP client that would try to connect to a server system on a specific port, typically 9923. This is reflected in how the Raw Panel device core on UniSketch works by default.
Today, the preferred mode of operation is server mode where the panel is a TCP server and the client is the host system that connects to the panel. This is the only mode supported on Blue Pill panels while on UniSketch panels you can use both modes.
The advantage of server mode is that the panel can accept connections from multiple systems which - assuming they are coordinated - provides possibility for redundancy. Furthermore, on panels using ASCII format it's easy to connect with a simple Telnet/nc/PuTTY application and work with the panel "by hand".

ASCII and Binary formats

As mentioned, you can encode Raw Panel messages as both ASCII and in a binary format.

- **ASCII** is the native - and only - format supported on UniSketch. It's pleasant to work with since it's human readable - at least until you get to setting text and graphics in displays. On UniSketch the protocol version is always V1 for ASCII, while using any of the translation tools below such as "ServerPanel2ClientSystem" provides a V2 interface. The added benefit of V2 can be usage of JSON for state messages as well as comma separated lists of hardware component numbers which partially addresses the complexity of working with text and graphics.
- **Binary** is the native and preferred format on Blue Pill panels. The encoding/decoding itself is done using Protocol Buffers and can be done in basically any programming language using the support libraries auto generated from the .proto file.
  If you want to use ASCII with a Blue Pill panel, it may have its Protocol Mode set to auto detecting ASCII when you connect and send the first message. Alternatively, you use the utility "ServerPanel2ClientSystem" to create an ASCII (V2) interface for it.

Conversion, Code examples, Utilities

In the repository raw-panel-utils (https://github.com/SKAARHOJ/raw-panel-utils) you can find utilities as binaries as well as Go source code. Here are the most significant utilities:

- **ColorDisplayButtonTest** - Test and debugging: Cycles colors and display content, responds to triggers, offers verbose views into the messages. A great all-round debugging utility.

- **ServerPanel2ClientSystem** - Binary/ASCII format converter: Connects to a panel (server) in binary or ASCII mode and creates another TCP server on localhost in binary or ASCII mode.
- **ServerPanel2ServerSystem** - Binary/ASCII format converter: Connects to a panel (server) in binary or ASCII mode and connects to a host system in binary or ASCII mode.

In addition, **Raw Panel Explorer** - the main interactive panel exploration utility - can be found at https://github.com/SKAARHOJ/raw-panel-explorer (download unsigned binaries for Mac and Windows). It will show an index of panels on the network (discovered by zeroconf), let you connect to them, retrieve the topology, render the panel, let you interact with the panel (set colors/display content) and receive triggers from panels. You can inspect basically every relevant aspect of the panel and learn quite a lot to get started quickly on your integration. Indispensable tool.

Protocol Buffers

Protocol Buffers (aka protobufs) is a free and open-source cross-platform data format used to serialize structured data. It is useful in developing programs to communicate with each other over a network or for storing data. It's been developed by Google since 2001 and is supported by many languages including C++, C#, Java, Python, JavaScript, Ruby, Go, PHP and Dart. See more here: https://developers.google.com/protocol-buffers

In the repository "rawpanel-lib" (https://github.com/SKAARHOJ/rawpanel-lib) and you can find the ibeam-rawpanel.proto description of the Raw Panel Protocol's usage of protocol buffers to represent messages internally. In the same repository you also find a number of functions that represents the authoritative way to translate between the ASCII format and the protobuf message structures.

Internally in all our code we use the protobuf message structures exclusively to relate to triggers and generate panel feedback. This ensures 100% compliance as well as a lot of help from code editors such as Visual Studio Code. For example, after having built a feedback message, all you need to do is to encode it in ASCII or Binary form using helper functions readily available from the "raw-panel-lib" repository. Please refer to the source code of the utilities in raw-panel-utils for examples.

Here is an example (in JSON format) of how an event from a panel is represented in a data structure:

```json
[
    {
        "Events": [
            {
                "HWCID": 34,
                "Timestamp": 833857293,
                "Binary": {
                    "Pressed": true,
                    "Edge": 4
                }
            }
        ]
    }
]
```

In ASCII format this looks like "HWC#34.4=Down" which is obviously much shorter, but from a computational point of view a structured message is usually preferred. Plus as you may be able to spot, it supports carrying multiple events and other objects in a single message body. For your information, the binary encoded version of the message above is this series of bytes: [66 14 8 34 18 4 8 1 16 4 48 208 161 225 141 3]

The Raw Panel Explorer tool is helpful in revealing the triggers from a panel.

When working with the protobuf formats, one thing to keep in mind is this: Zero values, false booleans and empty strings in protobuf messages are always omitted (implicit).

Handshaking

Handshaking in Raw Panel mode depends on whether you are in Client mode or Server mode.

In server mode, which is recommended and preferred by most and used exclusively on Blue Pill panels, any handshaking is optional. Usually you would immediately ask a panel to reveal information about itself using the "list" command, the "PanelTopology?" command etc. In terms of keep-alive signals, it's recommended that you send a "HeartBeatTimer" command with a value in milliseconds by which the panel should send a "ping" to the system. You must respond to this ping with any command. The reason why this is recommended is that the panel may not discover a broken network connection unless it proactively sends out a message to the system. If you enable the HeartBeatTimer the panel will disconnect itself unless it receives a response in 2 seconds.

In client mode - which only concerns UniSketch panels - there is a more elaborate handshake, but involving the same ideas: You must send the "list" command and an "ActivatePanel" command and you must respond to ping messages from the panel. The heart beat is not optional in client mode. See more details in the section on "Client Mode Handshaking" later in this document.

Inbound TCP commands
- from external system to SKAARHOJ panel

Introduction

In the **Command column** you will see the format for the ASCII command in plainly formatted text and after that you will see a mention of which objects from the *protobuf* definition you can use for the binary format. The idea here is to make a pairing for successful self exploration.
In the **V1** and **V2** columns you find information about which version of the protocol supports the command.

- V1 is the protocol used by all UniSketch panels (ASCII only)
- V2 is the protocol version used on all Blue Pill panels (Binary and ASCII)

The contents in the version columns is color coded:

- Green means "supported"
- Blue means "supported, incl. JSON format" which in turn means that alternatively to supplying a (series of) tokenized ASCII strings you can also supply it as a single line JSON block which should match the structure of the similar protobuf message. This clearly needs some more explanation and examples, please see later (and the Raw Panel Explorer tool)
- Red means "Not supported"

Legend example:

Command	Description	V1/V2
ASCII Command `Binary Protobuf Objects`	(Descriptive text)	√ = Supported √ = Supported+JSON formatting - = Not supported

A great place to study the alignment between the ASCII and binary versions of the commands would be in the *rawpanel-lib* repository. Check out the source code of the conversion functions:

- RawPanelASCIIstringsToInboundMessages()
- InboundMessagesToRawPanelASCIIstrings()

Commands

The basic incoming commands to the panel that an external system could send are listed in the table below.

Command	Description	V1	V2
HWC#*xx*=*yy* `States:` `HWCIDs` `HWCMode.State` `HWCMode.BlinkPattern` `HWCMode.Output`	**Status On/Off/Dimmed, Blinking and some fixed colors** *xx* is the HWC number, *yy* is a word defining the state of the component. The state, "yy", often translates into a light intensity state such as off / dimmed / on, but may also contain simple on/off binary and color information: Bit 0-3 forms a number from 0-15 and sets the behavior of an LED	√	√*

Command	Description	V1	V2
	• 0 = Off • 1,2,3 = On with fixed color (1=amber, 2=red, 3=green) [Legacy] • 4 = On with the color set by HWCc command (white is default) • 5 = Dimmed with the color set by HWCc command (white is default) Bit 4: Blink flag for mono color buttons. If set, a mono color button will blink. This is to provide a way to indicate a different “on” value like a red (2) or green (3) but for a button that can otherwise just show “on”. [Legacy, Deprecated, not used on Blue Pill] Bit 5: Output bit (32); If this is set, a binary output (like a GPO pin / relay) will be set if coupled with this HWC. Generally: Let bit 5 follow whether the “On” color (1,2,3 or 4) is commanded and let it be off if 0 or 5 is commanded. Bit8-11: Blink bits: If set 0b0001, the button will blink with a frequency of about 4 Hz, If set to 0b1000, the button will blink with a frequency of about 0.5Hz, if set to 1100 it will blink with a 0.5Hz frequency and a 75% duty cycle. The bits are a simple enabling mask against the systems bit-shifted millisecond clock and other combinations can create other blinking patterns. Below you find an overview of 4 second patterns for the values 1-15 for the 4 blinking bits: `Blinking= 1: *_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_` `Blinking= 2: **__**__**__**__**__**__**__**__` `Blinking= 4: ****____****____****____****____` `Blinking= 8: ********________********________` `Blinking= 3: ***_***_***_***_***_***_***_***_` `Blinking= 6: ******__******__******__******__` `Blinking= 7: *******_*******_*******_*******_` `Blinking= 5: *****_*_*****_*_*****_*_*****_*_` `Blinking= 9: *********_*_*_*_*********_*_*_*_` `Blinking=10: **********__**__**********__**__` `Blinking=11: ***********_***_***********_***_` `Blinking=13: *************_*_*************_*_` `Blinking=12: ************____************____` `Blinking=14: **************__**************__` `Blinking=15: ***************_***************_` Typically you would send these values back: 0 (“Off”), 36 (32+4 for “On”) and 5 (for “Dimmed”). Notice that this only sets the highlight state and most likely you will want to combine this command with "HWCc" in order to also set the RGB color of the button. *) In RWPv2.0 xx can be a comma separated list of integers to address multiple HWCs with the same content.		
HWCc#*xx*=*yy* `States:` `HWCIDs` `HWCColor.ColorRGB` `HWCColor.ColorIndex` *ColorRGB has priority over ColorIndex*	**Button color: index or rrggbb** *xx* is the HWC number, *yy* is a byte defining the color of the component in “On” and “Dimmed” state. Bit 7: Enable bit. If set, the color of the component is defined by this value, otherwise the panel default will be used. Most likely you will always want to set this. Bit 6: Defines the interpretation of bits 5-0; If set, bits 5-0 represents the component color with “rrggbb”. If clear, bits 5-0 represents an index from 0-16 pointing to a preset color from this list (all of which are selected to be visually distinct from each other): • 0: DEFAULT_COLOR, // Default (+bit 7 on = 128)	√	√

Command	Description	V1	V2
	• 1: 0, // Off (+bit 7 on = 129) • 2: 0b111111, // White (+bit 7 on = 130) • 3: 0b111101, // Warm White (+bit 7 on = 131) • 4: 0b110000, // Red (Bicolor) (+bit 7 on = 132) • 5: 0b110101, // Rose (+bit 7 on = 133) • 6: 0b110011, // Pink (+bit 7 on = 134) • 7: 0b010011, // Purple (+bit 7 on = 135) • 8: 0b110100, // Amber (Bicolor) (+bit 7 on = 136) • 9: 0b111100, // Yellow (Bicolor) (+bit 7 on = 137) • 10: 0b000011, // Dark blue (+bit 7 on = 138) • 11: 0b000111, // Blue (+bit 7 on = 139) • 12: 0b011011, // Ice (+bit 7 on = 140) • 13: 0b001111, // Cyan (+bit 7 on = 141) • 14: 0b011100, // Spring (Bicolor) (+bit 7 on = 142) • 15: 0b001100, // Green (Bicolor) (+bit 7 on = 143) • 16: 0b001101, // Mint (+bit 7 on = 144) • *17: 0b101010, // Light Gray (For Color/Graytone displays only)* • *18: 0b010101, // Dark Gray (For Color/Graytone displays only)* The colors marked "(Bicolor)" are the only ones recommended for use with red/green bicolor buttons on panels.		
HWCx#*xx*=*yy* `States:` `HWCIDs` `HWCExtended.Interpreta-tion` `HWCExtended.Value`	**Extended return values** *xx* is the HWC number, *yy* is a 16 bit unsigned integer defining the extended output of the component. The rightmost 10 bits of this integer is the value. Bits 11 and 12 are reserved for the individual output types to define. The leftmost 4 bits of this word is the output type: **0: None** Disable extended return values. This must be used to disable extended mode for LEDs that respond to extended return values. **1: Output Strength** Value from 0-1000, used to set a strength indication on an LED bar. *Legacy:* On UniSketch it will also set *and maintain* a position of a motorized fader. When you receive inputs from a motorized fader, you should acknowledge the new value by returning it immediately with HWCx#xx=(4096+value) (=type 1) so the fader knows it should stay in this position. If not, then you will experience that the fader moves back to the last position it was set to. [Deprecated use for faders, use type 5 instead] **2: Directional Output Strength** *Not available on any platform* **3: Shows steps (LED bars)** Values: 0 = All LEDs are off 1 = first LED is on 2 = second LED is on ... n = n'th LED is on n+1 and above = the full bar will light up dimmed. n represents the number of LEDs in the LED bar **4: VU metering (LED bars)**	√	√

Command	Description	V1	V2
	Used to display audio level (values 0-1000) (See separate section for details on the color patterns) **5: Fader move to position.** Moves the motorized fader into position and leaves it there. This is the most useful way to use motorized faders in many cases: If you use output type 5, the faders previously set position from the external system was a one-off event. **6: Buzzer (proposal juli 2021):** Bit0-3: Time of buzzing, 0-4 seconds approx. Bit4-7: Buzzing frequency: A low value is a low (slow) frequency of buzzing and a high value is a higher (faster) frequency of buzzing. Bit8-11: Pattern bits: If set 0b0001, the buzzer will be active with an active frequency of about 4 Hz, If set to 0b1000, the activity frequency is about 0.5Hz, if set to 1100 it will be active with a 0.5Hz frequency and a 75% duty cycle. The bits are a simple enabling mask against the systems millisecond clock and other combinations can create other blinking patterns.		
HWCt#*xx*=*string* `States:` `HWCIDs` `HWCText.BackgroundColor` `HWCText.PixelColor` `HWCText.Inverted` `HWCText.TextStyling` `HWCText.Scale` `HWCText.PairMode` `HWCText.IntegerValue2` `HWCText.Textline1` `HWCText.Textline2` `HWCText.SolidHeaderBar` `HWCText.Title` `HWCText.StateIcon` `HWCText.ModifierIcon` `HWCText.Formatting` `HWCText.IntegerValue`	**Display text** *xx* is the HWC number, *string* is a string tokenized by a vertical pipe character, "|", where each position represents a given parameter being either an integer, boolean or string. The format of *string* follows this: [value]|[format]|[fine]|[Title]|[isLabel]|[label 1]|[label 2]|[value2]|[values pair]|[scale]|[scale range low]|[scale range high]|[scale limit low]|[scale limit high]|[img]|[font]|[font size]|[advanced settings] *string* may not be longer than 63 chars • [value] is a 32 bit integer representing the numerical value to be shown. If empty, it will not render at all (like format=7). • [format] defines how [value] is formatted: 0=Integer, 1=10e-3 Float w/2 dec. points, 2=Percent, 3=dB, 4=Frames, 5=1/[value], 6=Kelvin, 7=Hidden, 8=10e-3 Float w/3 dec., 9=10e-2 Float w/2 dec., 10=1 Text line (Title & value=size 1-4), 11=2 Text lines (Label 1, Label 2 & value=size 1-2). Default if empty is Integer. • [fine] is used to set various icons • [Title] defines the title string shown in the top of the display. Up to 10 chars long. • [isLabel] is a boolean (0/1) that sets if the title bar should be rendered as a "label". This is a convention used on SKAARHOJ controllers to indicate whether the content of a display shows the state of a given parameter (the current value) or if the display shows a label that indicates what will happen if the associated control component is triggered. Default is to show "state" which is indicated by a solid bar underlying the text. In "label" mode the title is rendered with only a thin line underneath. • [label 1] First text line under title. If [label 2] is omitted it will be printed in large font. Up to 25 chars long. If small text is preferred without invoking [label 2], please set [value2] to something. • [label 2] Second text line under title. If not empty, both [label 1] and [label 2] will print in small letters. • [value2] Represents a second value. This is used if you use [label 1] and [label 2] as prefixes for [value] and [value2] along with settings for[values pair] • [values pair] ranges from 1-4 and indicates 4 variations of boxing of value pairs. • [scale][scale range low]|[scale range high]|[scale limit low]|[scale limit	√	√

Command	Description	V1	V2
	high] indicates different types of scales in the bottom of the graphic that can show a range of a given value. • [img] is an index to a system stored media graphic file. • [font] is font face (0-2) • [font size] is font sizes horizontal and vertical for both content and title • [advanced] is some other settings *Please check out the section later in this document for examples and a table with a better overview.*		
HWCg#*xx*= *header*:*base64* `States:` `HWCIDs` `HWCGfx.ImageType == rwp.HWCGfx_MONO`	**Display Monochrome bitmaps** *xx* is the HWC number, *header* is a header string and *base64* is a part of the image data encoded in base64 *header* is on the form [sequence number, zero is first]/[Last number in sequence],[width, pixels]x[height, pixels],[x-coordinate from upper left],[y-coordinate from upper left] Please check out the section later for more information on sending image data to Raw Panel. Specifying the x and y coordinates are optional. **Depreciated legacy features (V1, UniSketch):** On UniSketch, sending an image that only partially fills the display tile would only write to that portion of the tile. This is also not supported on Blue Pill panels. *Description of a special 64x32 legacy format on UniSketch:* Sending a 64x32 monochrome images to the panel is done by sending three consecutive lines, each representing 86, 86 and 84 bytes of the image data respectively, totaling 256 bytes. The header being an index from 0-2 is used to indicate which part of the image is represented in the line. Always send them in this order. When index 2 reaches the client it will assume that all image data has been received and write it to the display. The 256 byte monochrome image data itself represents the image starting with bit 7 in the first byte being the upper left pixel (1=on, 0=off) and then progressing to the right and down (reading direction). Caching (future, not yet implemented): To speed up repeated usage of the same content, you can assign a 15 bit hash number which can be used to recall it again. To indicate that an image should be cached, simply send "{CS:xxxxx}" where xxxxx is a 15 bit non-zero decimal unique identification number of your choice before sending the three parts of the graphic content. For example "HWCg#xx={CS:xxxxx}". To later set the same content again, but using the cached content, simply send a string with "HWCg#xx={CRxxxxx}" and nothing more. Notice that a successful recall may only work for the same display type, otherwise it can appear scrambled. Also caching will only work for 64x32 displays.	√	√
HWCgRGB#*xx*= *header*:*base64* `States:` `HWCIDs` `HWCGfx.ImageType == rwp.HWCGfx_RGB16bit`	**Display RGB bitmaps** Format is similar to sending monochrome data with "HWCg". See also the dedicated section later.	√	√
HWCgGray#*xx*= *header*:*base64*	**Display Grayscale bitmaps**	√	√

Command	Description	V1	V2
States: *HWCIDs* *HWCGfx.ImageType == rw-p.HWCGfx_Gray4bit*	Format is similar to sending monochrome data with "HWCg". See also the dedicated section later.		
HWCrawADCValues#*xx*=[0/1] *States:* *HWCIDs* PublishRawADCValues	Enable / Disable continuous reporting of raw analog values from ADCs on the panel (for service a profiling)	-	√
Clear `Command:` `ClearAll_connections`	Clears all values sent by HWC, HWCx, HWCc (LED colors) and Display content	√	√
ClearLEDs `Command:` `ClearLEDs`	Clears only LED content (HWC, HWCx, HWCc)	√	√
ClearDisplays `Command:` `ClearDisplays`	Clears display content	√	√
Reboot `Command:` `Reboot`	Reboots the panel (returns text "Msg=Rebooting...\n")	√	√
ActivePanel=1 `Command:` `ActivatePanel`	Activates panel (Client Mode) Send this to activate the panel when “list” is received from the panel. It's recommended to append ActivePanel with <NL> in order to make sure, the full command gets noticed. Cases with short disconnects of the connection has proven to be vulnerable to missing this command which results in no initialization.	√	√
list `Command:` `SendPanelInfo`	Returns information about the panel: _serial: Serial number (string with alphanumeric content) _model : SKAARHOJ model SK_[Panel ID]. **Notice**, no indication of options on V1 / UniSketch. Other prefixes may exist, for example "XP_" is a namespace for various non-SKAARHOJ hardware panels. _version: Version (branch) _bluePillReady: 1 if Blue Pill Ready is enabled. In this mode, press on encoders act like buttons (Down/Up events), the LED ring responds like a button's LED and the panel will dim itself when unconnected and set a 2 minute sleep timer. _platform: Platform (ibeam (codename) = Blue Pill platform) _name: Friendly Name of the panel *In Server mode only:* _serverModeLockToIP: If the panel is locked to one or more IP addresses, it's provided as a comma separated list with this command. _serverModeMaxClients: Returns the limits to number of simultaneous clients. Zero means no cap is set. On UniSketch you can have at most 7-8 clients.	√	√
ping `FlowMessage:` `InboundMessage_PING`	Keep-alive, panel returns “ack<NL>”	√	√
ack `FlowMessage:` `InboundMessage_ACK`	Accepted, but no action taken and nothing returned.	√	√
nack	Accepted, but no action taken and nothing returned.	√	√

Command	Description	V1	V2
FlowMessage: InboundMessage_NACK			
map Command: ReportHWCavailability	Sends the map. The map is sent to all connected endpoints on UniSketch. The map is also often sent proactively upon connection from UniSketch panels, but while a connected system may sometimes experience that, it cannot be assumed and therefore a connecting system should always ask specifically for the map initially. Changes to the map during the connection should be sent over by the panel automatically (mostly something that happens on UniSketch panels where HWCs can be dynamically included/excluded from Raw Panel operation).	√	√
PanelTopology? Command: SendPanelTopology	Asks panel to send SVG and JSON data for topology	√	√
BurninProfile? Command: SendBurninProfile	Asks panel to send the burnin profile JSON. The burnin profile is data used to run a manually assisted test sequence on a panel to verify its functionality. This is used at the factory and potentially for remote validation.	-	√
Connections? Command: GetConnections	Returns IP addresses of the connected clients.	√	√
RunTimeStats? Command: GetRunTimeStats	Returns statistics for uptime and boot count.	√	√
Mem, Flag#, Shift, State	Legacy. See TCP Server device core (as mentioned previously in this document) for these commands – they are typically not relevant for Raw Panel implementations in third party systems.	√	-
SleepTimer=xx Command: SetSleepTimeout	Sets the panel sleep timer in milliseconds: This is the number of milliseconds that shall pass before the panel will enter sleep. If zero, sleep is disabled. Generally, GPI inputs and outputs should not be affected by sleep mode.	√	√
SleepTimer? Command: GetSleepTimeout	Will return the global sleep timer value from the panel. (milliseconds)	√	√
SleepMode=xx Command: SetSleepMode	Sets the panel sleep mode 0 = FireWorks (Default) - LEDs will animate on the panel 1 = Buttons Off	√	√
SleepScreenSaver=xx Command: SetSleepScreenSaver	0 = "Wake Up On Key Press" message (default) 1 = "Sheep And Goats" - Classic UniSketch funtime screen saver 2 = "Save The Oleds" message 3 = Just Dimmed - keeps content, just dims the panel	√	√
HeartBeatTimer=xx Command: SetHeartBeatTimer	*For panels in server mode only:* Instructs the panel to send a heart beat "ping<NL>" with a period of xx ms (xx must be > 200). If HeartBeatTimer is enabled, the panel will also monitor that it receives an answer back from the client within 2 seconds of sending the heart beat ping. Any answer counts to satisfy the heart beat timer. If no answer is received, the panel will disconnect the client. The heart beat timer is individual for each connected client.	√	√

Command	Description	V1	V2
DimmedGain=xx `Command:` `SetDimmedGain`	Sets the gain level (0, 1-64) for the dimmed button state on the panel. Full gain is 64, a good dimmed value is around 4-16. Setting DimmedGain to zero will reset it to the panel defaults (10-16).	√	√
WakeUp! `Command:` `WakeUp`	Will wake up the panel if it was asleep.	√	√
encoderPressMode=xx	Depreciated Legacy, use Blue Pill Ready option instead. In xx: bit 0: If set, encoders will return "Press" on "act down" (as well as press after holding down for 1 second). Default is 1.	√	-
PanelBrightness=x,y `Command:` `PanelBrightness.LEDs` `PanelBrightness.OLEDs`	Brightness for LEDs (x) and OLEDs (y). x and y goes from 0-8. If ",y" is omitted from the command, the x-value will apply to both LEDs and OLEDs (displays). On panels that uses only one brightness value per HWC, they may interpret two values by taking their average.	√	√
Webserver=x `Command:` `SetWebserverEnabled`	*UniSketch only:* Webserver on (x=1) and off (x=0)	√	√
PublishSystemStat=xx `Command:` `PublishSystemStat`	Interval (seconds) by which to send out system stats. If zero, it's disabled.	-	√
LoadCPU=xx `Command:` `LoadCPU`	Number of CPU cores to stress (0-4). Zero is off.	-	√
(Unknown command)	returns “nack<NL>”	√	*

VU meter LED bars

When using extended return values for VU metering, these are the limits that drive the various sized LED bars on SKAARHOJ panels:

6 segment bar (RCP) value thresholds (values 0-1000):

>50	>250	>450	>650	>800	>950

7 segment bar (Mini Fly) value thresholds (values 0-1000):

>50	>200	>350	>500	>650	>800	>950

10 segment bar value thresholds (values 0-1000):

>50	>150	>250	>350	>450	>550	>650	>750	>850	>950

15 segment bar value thresholds (values 0-1000):

>50	>116	>182	>250	>316	>382	>450	>516	>582	>650	>710	>770	>830	>890	>950

3 segment led bars has global pattern for all 3 LEDs (values 0-1000):

JSON formatted states

The HWC-series of commands from the table above are marked to also support JSON (blue fields). The allowed JSON structure corresponds to the State field inside the protobuf message. While is this theoretically all you need to know, here is a an example and a way to get more.

The three commands below

```
HWC#38=4
HWCc#38=137
HWCt#38=9999|2|40|Value
```

are represented by the following protocol buffer structure (JSON):

```
[
    {
        "States": [
            {
                "HWCIDs": [
                    38
                ],
                "HWCMode": {
                    "State": 4
                },
                "HWCColor": {
                    "ColorIndex": {
                        "Index": 9
                    }
                },
                "HWCText": {
                    "IntegerValue": 9999,
                    "Formatting": 2,
                    "ModifierIcon": 5,
                    "Title": "Value:",
                    "SolidHeaderBar": true
                }
            }
        ]
    }
]
```

Taking the part marked in bold, stripping the line breaks and tabulation and sending it to a panel will work the same as the individual three lines:

```
{"HWCIDs":[38],"HWCMode": {"State": 4},"HWCColor": {"ColorIndex": {"Index": 9}},"HWCText": {"IntegerValue": 9999,"Formatting": 2,"ModifierIcon": 5,"Title": "Value:","SolidHeaderBar": true}}
```

The string is longer, but the advantage is mainly the structured way display content is represented in JSON rather than the |-delimited format.

If you are interested in how hardware component states can be represented as JSON, there is the implicit documentation of the protobuffer structures available, but the most fruitful way to discover structures for what you need to do is likely to use the utility below from raw-panel-utils:

```
ColorButtonDisplayTest -verboseIncoming 2 [panel-id]:9923
```

As you work with the application and the panel it will show console output of the incoming commands to the panel and you should be able to spot the correlation between the feedback you see on the panel, the ASCII commands in their standard form and the JSON structures you could extract and use. Here is how the console output would look like:

```
...
INFO[0070] System -> Panel: HWC#40=4                   module=main
INFO[0070] System -> Panel: HWCc#40=137                module=main
INFO[0070] System -> Panel: HWCt#40=9999|2|56|C.Icon=7   module=main
INFO[0070] [
 {
  "States": [
   {
    "HWCIDs": [
     41
    ],
    "HWCMode": {
     "State": 4
    },
    "HWCColor": {
     "ColorIndex": {
      "Index": 9
     }
    },
    "HWCText": {
     "Formatting": 7,
     "Textline1": "Floating",
     "Textline2": "Point",
     "PairMode": 1,
     "Scale": {},
     "TextStyling": {
      "TitleFont": {},
      "TextFont": {
       "FontFace": 1,
       "TextHeight": 2,
       "TextWidth": 1
      },
      "ExtraCharacterSpacing": 1
     }
    },
    "HWCGfx": {}
   }
  ]
 }
]  module=main
INFO[0070] System -> Panel: HWC#41=4                   module=main
INFO[0070] System -> Panel: HWCc#41=137                module=main
INFO[0070] System -> Panel: HWCt#41=||||1|Floating|Point||1||||||||1|9|4  module=main
...
```

Notice that the tool **Raw Panel Explorer** from our repository raw-panel-explorer is very helpful in creating feedback JSON strings for use int your applications and for learning the structure.

Outbound TCP commands - from panel to external system

Introduction

Please see the introduction written for the Inbound TCP commands section above. It's the same that applies here.

A great place to study the alignment between the ASCII and binary versions of the commands would be in the *raw-panel-lib* repository. Check out the source code of the conversion functions:

- RawPanelASCIIstringsToOutboundMessages()
- OutboundMessagesToRawPanelASCIIstrings()

Commands

This lists the outgoing commands from the SKAARHOJ panel and which the external system should understand and respond to.

Command	**Description**	**V1**	**V2**
HWC#*xx[.mask]*=*string* `Events:` `HWCID:` `Binary:` `Pulsed:` `Absolute:` `Speed:`	**Trigger action from hardware component** *xx* is the HWC number, *string* contains information about the trigger. Buttons may also add the *mask*, which is a period followed by a number 1,2,4,8 or 16 indicating which edge was pressed on the button: • Two and Four-way buttons: Up=1, Left=2, Down=4, Right=8 • Encoders button press: Returns with edge 16 (if Blue Pill Ready is enabled) *string* can have any of these forms: • "Down" : the component (typically a button or a GPI trigger or encoder knob) is pressed down • "Up" : the component is released again • "Abs:*yy*" : A change, *yy*, to an absolute position (for example a T-bar). yy ranges 0 to 1000 • "Speed:*yy*" : A change, *yy*, to a speed (for example a spring loaded joystick). yy ranges -500 to 500 • "Enc:*yy*" : Pulses, yy, from an encoder. The sign indicates direction. • "Press" : represents that Down and Up happened essentially simultaneously - a pulse (Legacy)	√	√
HWCmsg#xx=CN:content	**Change notification** (Legacy)	√	-

Command	Description	V1	V2
map=zz:xx `HWCavailability`	**Local HWC to External HWC mapping information** *zz* is the native HWC number on the client panel and *xx* is the external HWC number used in communication with the server (the xx found in any other HWC command in this API). The command is issued initially and when changes in this mapping appears. It can be helpful for the server to know which HWCs are actually active on the panel. The information about the native HWC number can be of interest in relation to servers which use the topology information. Notice how an external HWC may be associated with multiple native HWCs. Changes in the map can be used to track if a display may need update. For instance, the map is zeroed out in case of a sleep timeout on the panel and regains its values when it returns from sleep, thus giving the server a chance to repopulate the displays of the hardware components. Please check comments made on the "map" incoming command too. Proactive delivery of the map cannot be assumed upon connection although it may often be experienced (from UniSketch panels).	√	√
BSY `FlowMessage: OutboundMessage_BSY`	Busy message. Hold back with sending new data until RDY. Typically invoked when sending images to UniSketch panels.	√	-
RDY `FlowMessage: OutboundMessage_RDY`	Ready message. You can send data to the panel again now.	√	-
list `FlowMessage: OutboundMessage_HELLO`	Initialization status request, return "<NL>ActivePanel=1<NL>". Used only on UniSketch Client mode panels.	√	√
ping `FlowMessage: OutboundMessage_PING`	Sends "ping<NL>"	√	√
ack `FlowMessage: OutboundMessage_ACK`	Acknowledgement to "ping" command Variant: "ack: image WxH received" (Legacy, UniSketch)	√	√
nack `FlowMessage: OutboundMessage_NACK`	Unknown incoming command	√	√
_model `PanelInfo Model`	_model=[Model / Product Key]. String delimited by "_". The first part is a name space, typically "SK" for SKAARHOJ original panels, while "XP" is seen for various external panels. Other namespaces may be invoked in the future. Notice that *no options* are included in product keys returned from UniSketch panels but some options are returned on Blue Pill Inside panels. Example: SK_RACKFUSIONLIVE - On UniSketch this could be with or without NKK option. On Blue Pill it would be without NKK option. SK_RACKFUSIONLIVE_NKK - On Blue PIll panels this would be the NKK option.	√	√
_serial `PanelInfo Serial`	_serial=[Serial number, Alpha-numeric string]	√	√

Command	Description	V1	V2
_version `PanelInfo` `SoftwareVersion`	Software version from panel.	√	√
_bluePillReady `PanelInfo` `BluePillReady`	Returns 1 if Blue Pill Ready is set (device core option on UniSketch). Blue Pill ready means the panel generally complies with the standards for sending and responding to commands with raw panel.	√	-
_platform `PanelInfo:` `Platform`	Platform. Shared from Blue Pill panels, where it will be "ibeam" (early codename for Blue Pill)	--	√
_name `PanelInfo:` `Name`	Panel friendly name	-	√
_serverModeLockToIP `PanelInfo` `LockedToIPs`	Returns a semi-colon separated list of IP addresses the panel is locked to (Server mode)	√	√
_serverModeMaxClients `PanelInfo` `MaxClients`	Returns max number of clients (Server mode)	√	√
_panelTopology_svgbase `PanelTopology`	Panel base SVG	√	√
_panelTopology_HWC `PanelTopology`	JSON with HWC (HardWare Component) data	√	√
_burninProfile `BurninProfile`	JSON with burnin profile sequence data (for service and support)	-	√
_state:[reg]=xx	Informs about the panels state register value (sent when changed) (UniSketch only)	√	-
_shift:[reg]=xx	Informs about the panels shift register value (sent when changed) (UniSketch only)	√	-
_isSleeping=[0/1] `SleepState`	Informs about the panels whether the panel sleeps or not (sent proactively when changed)	√	√
_sleepTimer=xx `SleepTimeout`	Returns the sleep timer in milliseconds: This is the number of milliseconds that shall pass before the panel will enter sleep. If zero, sleep is disabled.	√	√
_sleepMode=xx	Returns Sleep Mode in response to setting it.	√	-
_sleepScreenSaver=xx	Returns Sleep Screen Saver in response to setting it.	√	-
_heartBeatTimer=xx `HeartBeatTimer`	Returns the heart beat timer value in response to setting it. If larger than zero this is the time between the panel sending out heart beat pings.	√	√
PanelBrightness=x,x	Returns panel brightness,oled brightness in response to setting it.	√	-
DimmedGain=xx `DimmedGain`	Returns dimmed gain in response to setting it.	√	√
Webserver=x	Returns webserver status in response to setting it.	√	-
_connections=[IP];[IP];...;[IP];	Returning connected IPs	√	√

Command	Description	V1	V2
Connections			
_bootsCount RunTimeStats BootsCount	How many times the panel has experienced a reboot	√	√
_totalUptimeMin RunTimeStats TotalUptime	Total uptime in minutes during the lifetime of the panel	√	√
_sessionUptimeMin RunTimeStats SessionUptime	Total uptime in minutes since boot.	√	√
_screenSaverOnMin RunTimeStats ScreenSaveOnTime	Total time in minutes out of the total uptime time where the panel has been running in SleepMode	√	√
ErrorMsg=*string* ErrorMessage	Error message string (like when panel connection is refused)	√	√
Msg=*string* *Message*	Message string (like when rebooting)	√	√
SysStat=CPUUsage:xx:CPU Temp:yy:ExtTemp:zz SystemStat CPUUsage CPUTemp ExtTemp	Sends System statistics in response to setting period reporting with PublishSystemStat. xx: CPU load in percent (integer) yy: CPU temperature (float) zz: External board temperature (float), -100 if not available.	-	√

Code examples

To help you get started with Raw Panel we have provided a few repositories with scripts, utilities and source code examples.

Conversion, Code examples, Utilities

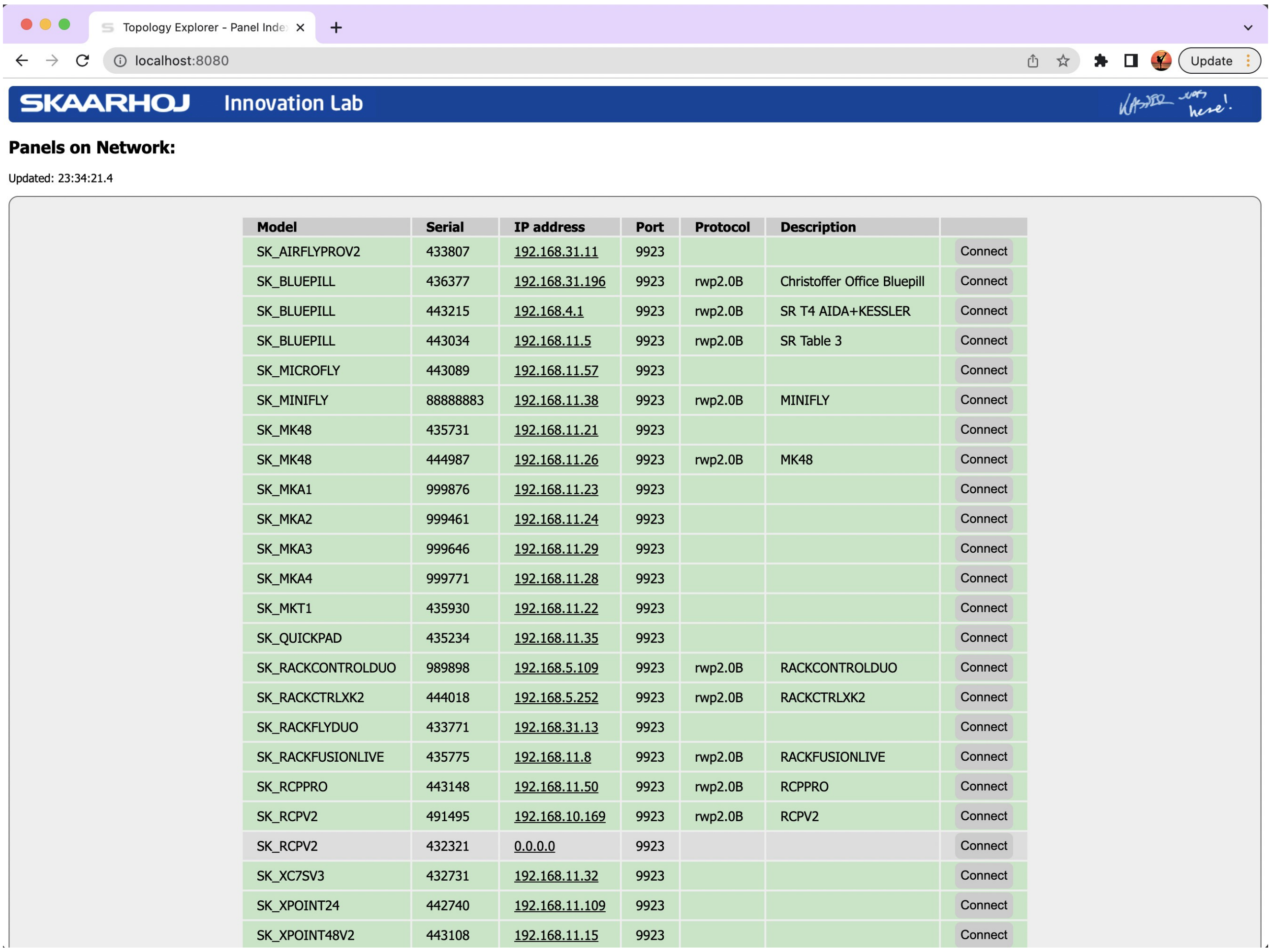
In the repository raw-panel-utils (https://github.com/SKAARHOJ/raw-panel-utils) you can find utilities as binaries as well as Go source code. These are all written in Go, which is a statically typed, compiled programming language designed at Google. It is syntactically similar to C, but with memory safety, garbage collection, structural typing, and CSP-style concurrency. It's highly recommended if you don't know Go already.

- **ColorDisplayButtonTest** - Test and debugging: Cycles colors and display content, responds to triggers, offers verbose views into the messages. A great all-round debugging utility.
- **ServerPanel2ClientSystem** - Binary/ASCII format converter: Connects to a panel (server) in binary or ASCII mode and creates another TCP server on localhost in binary or ASCII mode.
- **ServerPanel2ServerSystem** - Binary/ASCII format converter: Connects to a panel (server) in binary or ASCII mode and connects to a host system in binary or ASCII mode.
- **Burnin** - Reads the burn-in profile from a Blue Pill panel (V2) and manages a test sequence for the panel.
- **ImageConverter** - Can help convert png/jpeg files on your system to raw panel commands

In addition, **Raw Panel Explorer** - the main interactive panel exploration utility - can be found at https://github.com/SKAARHOJ/raw-panel-explorer (download unsigned binaries for Mac and Windows). It will show an index of panels on the network (discovered by zeroconf), let you connect to them, retrieve the topology, render the panel, let you interact with the panel (set colors/display content) and receive triggers from panels. You can inspect basically every relevant aspect of the panel and learn quite a lot to get started quickly on your integration. Indispensable tool.

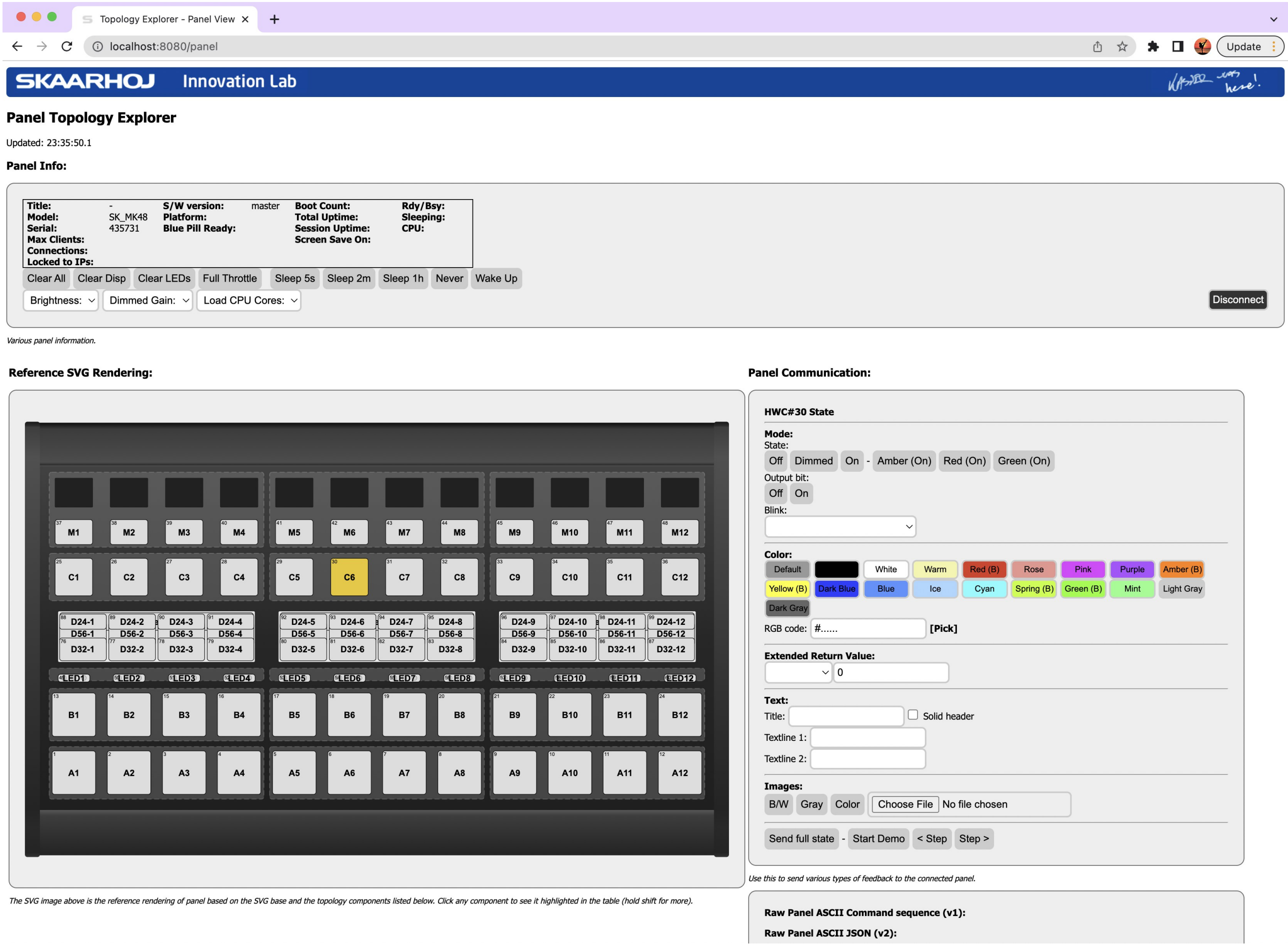
Raw Panel Explorer

Just start it from the command line and open a web browser to localhost:8080. You should see this with your panel(s) listed (if they are on the same network that is):



Model	Serial	IP address	Port	Protocol	Description	
SK_AIRFLYPROV2	433807	192.168.31.11	9923			Connect
SK_BLUEPILL	436377	192.168.31.196	9923	rwp2.0B	Christoffer Office Bluepill	Connect
SK_BLUEPILL	443215	192.168.4.1	9923	rwp2.0B	SR T4 AIDA+KESSLER	Connect
SK_BLUEPILL	443034	192.168.11.5	9923	rwp2.0B	SR Table 3	Connect
SK_MICROFLY	443089	192.168.11.57	9923			Connect
SK_MINIFLY	88888883	192.168.11.38	9923	rwp2.0B	MINIFLY	Connect
SK_MK48	435731	192.168.11.21	9923			Connect
SK_MK48	444987	192.168.11.26	9923	rwp2.0B	MK48	Connect
SK_MKA1	999876	192.168.11.23	9923			Connect
SK_MKA2	999461	192.168.11.24	9923			Connect
SK_MKA3	999646	192.168.11.29	9923			Connect
SK_MKA4	999771	192.168.11.28	9923			Connect
SK_MKT1	435930	192.168.11.22	9923			Connect
SK_QUICKPAD	435234	192.168.11.35	9923			Connect
SK_RACKCONTROLDUO	989898	192.168.5.109	9923	rwp2.0B	RACKCONTROLDUO	Connect
SK_RACKCTRLXK2	444018	192.168.5.252	9923	rwp2.0B	RACKCTRLXK2	Connect
SK_RACKFLYDUO	433771	192.168.31.13	9923			Connect
SK_RACKFUSIONLIVE	435775	192.168.11.8	9923	rwp2.0B	RACKFUSIONLIVE	Connect
SK_RCPPRO	443148	192.168.11.50	9923	rwp2.0B	RCPPRO	Connect
SK_RCPV2	491495	192.168.10.169	9923	rwp2.0B	RCPV2	Connect
SK_RCPV2	432321	0.0.0.0	9923			Connect
SK_XC7SV3	432731	192.168.11.32	9923			Connect
SK_XPOINT24	442740	192.168.11.109	9923			Connect
SK_XPOINT48V2	443108	192.168.11.15	9923			Connect

After selecting a panel you will see a view with it's topology listed in a table as well as the panel shown rendered as an SVG. It's all interactive and relies only on the topology received from the panel:



ServerPanel2ClientSystem

This tool is basically a converter between the ASCII and binary versions of the Raw Panel Protocol. It may also serve to bring in support for V2 when working with UniSketch panels.

Here is an example from a terminal:

```
ServerPanel2ClientSystem -binPanel 192.168.11.8:9923 9924
```

The line above will try to connect to a Blue Pill panel on IP address 192.168.11.8, port 9923. The -binPanel option means the utility will try to send binary encoded Raw Panel messages to the panel. On localhost port 9924 it will also open a TCP server to which you can connect using ASCII (implicit when -binSystem is not set). So, in another terminal you can use telnet, nc or PuTTY to open that port:

```
nc localhost 9924
```

When connected, you can type in "list" and ented, and you should see a response like this:

```
list
_model=SK_RACKFUSIONLIVE
_serial=435775
_version=v0.0.9
_name=Unknown
_platform=ibeam
```

This is the ASCII version of the binary encoded data delivered from the Blue Pill based panel on IP 192.168.11.8. If you study the console output of ServerPanel2ClientSystem you will see this in total:

```
Welcome to Raw Panel - Server Panel 2 Client System! Made by Kasper Skaarhoj (c) 2020-
2022
Configuration:
  binPanel:   true
  binSystem:  false
  system port:  9924
Ready to accept TCP connections on port 9924 and facilitate communication to panel on
192.168.11.8:9923...

Trying to connect to panel on 192.168.11.8:9923...
Success - Connected to panel
Success - TCP Connection from a system at [::1]:53006...
System -> Panel:  [18 2 16 1]
Panel -> System: _model=SK_RACKFUSIONLIVE
Panel -> System: _serial=435775
Panel -> System: _version=v0.0.9
Panel -> System: _name=Unknown
Panel -> System: _platform=ibeam
System: [::1]:53006 disconnected
```

This code shows the list command in binary encoding ([18 2 16 1]) and the response from the binary panel which is then forwarded.

ServerPanel2ServerSystem

This tool is also a ASCII/binary converter, but at the same time it makes two north poles meet: It facilitates that at system can be a TCP server, which is how many systems are created for using Raw Panel in client mode. In other words: The panel can be a TCP server, and the system can be a TCP server. The utility ServerPanel2ServerSystem is therefore a double-TCP client that bridges the two.

An example you can try is this: from SKAARHOJs Support repository in path Files/UniSketchTCPClient/, start this Python script:

```
./TCPserver_colorAndDisplayTest.py
```

This will create a TCP server on localhost port 9923. You will see no output until a panel connects to it. If you have a UniSketch, you could try to point it to the IP address of your computer with Raw Panel and you should see color and display content cycle. For now, we will not see anything until ServerPanel2ServerSystem connects and makes a bridge for us. So in another terminal, you start the utility:

```
ServerPanel2ServerSystem.go -binPanel 192.168.11.8:9923 localhost:9923

Welcome to Raw Panel - Server Panel 2 Server System! Made by Kasper Skaarhoj (c) 2020-
2022
Configuration:
  binPanel:   true
  binSystem:  false
Ready to facilitate communication between a panel and system, both in server mode. Start-
ing to connect...

Trying to connect to panel on 192.168.11.8:9923...
Trying to connect to system on localhost:9923...
Success - Connected to panel
Success - Connected to system
System -> Panel:  [18 2 8 1]
System -> Panel:  [18 2 32 1]
System -> Panel:  [18 2 16 1]
Panel -> System: RDY
Panel -> System: map=4:4
Panel -> System: map=8:8
Panel -> System: map=9:9
Panel -> System: map=45:45
Panel -> System: map=24:24
```

```
Panel -> System: map=16:16
....
```

This tells you that the utility successfully connected to a binary Blue Pill panel on 192.168.11.8, port 9923 and also to the server system on localhost port 9923. You can see how the system sends out the list command and a few other things and the panel responds back and after the handshake is done, the Python script will run a colorful animation of displays and buttons on the panel. At this point the Python script will have shown something in it's console output:

```
Client 127.0.0.1 sent: 'list<NL>'
- Returned state and assumes panel is now ready
Client 127.0.0.1 sent: 'RDY<NL>'
Client 127.0.0.1 sent: 'map=4:4<NL>'
Client 127.0.0.1 sent: 'map=8:8<NL>'
Client 127.0.0.1 sent: 'map=9:9<NL>'
Client 127.0.0.1 sent: 'map=45:45<NL>'
Client 127.0.0.1 sent: 'map=24:24<NL>'
Client 127.0.0.1 sent: 'map=16:16<NL>'
...
```

ColorDisplayButtonTest

This utility is useful to test your panels capabilities with respect to displays, LED colors as well as triggers returned. The utility has a fair bit of options you can set to achieve various behaviors, but for the most immediate use you just need the same as above: Point it to an IP address and port and mention if the panel is a binary or ASCII panel:

```
ColorDisplayButtonTest -binPanel 192.168.11.8:9923

Welcome to Raw Panel - Server Panel Color/Display/Button test! Made by Kasper Skaarhoj, (c) 2020-22
INFO[0000] Trying to connect to panel 1 on 192.168.11.8:9923 ...  module=main
INFO[0000] Success - Connected to panel                  module=main

   0: .......... ........INFO[0018] Panel 1 -> System: HWC#42=Speed:-12          module=main
INFO[0018] Panel 1 -> System: HWC#42=Speed:-38          module=main
INFO[0018] Panel 1 -> System: HWC#42=Speed:-58          module=main
INFO[0018] Panel 1 -> System: HWC#42=Speed:-79          module=main
INFO[0018] Panel 1 -> System: HWC#42=Speed:-99          module=main
INFO[0018] Panel 1 -> System: HWC#42=Speed:-120         module=main
.INFO[0019] Panel 1 -> System: HWC#42=Speed:-99          module=main
INFO[0019] Panel 1 -> System: HWC#42=Speed:-73          module=main
INFO[0019] Panel 1 -> System: HWC#42=Speed:-43          module=main
INFO[0019] Panel 1 -> System: HWC#42=Speed:-12          module=main
INFO[0019] Panel 1 -> System: HWC#42=Speed:0            module=main
.INFO[0020] Panel 1 -> System: HWC#26.4=Down             module=main
INFO[0020] Panel 1 -> System: HWC#26.4=Up               module=main
 .INFO[0021] Panel 1 -> System: HWC#26.1=Down             module=main
INFO[0021] Panel 1 -> System: HWC#26.1=Up               module=main
.INFO[0022] Panel 1 -> System: HWC#23=Enc:1              module=main
INFO[0022] Panel 1 -> System: HWC#23=Enc:2              module=main
INFO[0022] Panel 1 -> System: HWC#23=Enc:1              module=main
INFO[0022] Panel 1 -> System: HWC#23=Enc:-2             module=main
INFO[0022] Panel 1 -> System: HWC#23=Enc:-1             module=main
INFO[0022] Panel 1 -> System: HWC#23=Enc:-1             module=main
........ .......... .......... ..........
  60: .......... .......... .......... .......... .......... ..........
 120: .......... .......... .......... ......
```

After about 10 seconds the utility will start to progressively write content in the displays and change colors of buttons on the panel. While that happens you see a dot in the console every second. An unbroken number of dots shows that no triggers were sent from the panel. If you operate hardware components on the panel (in the example a joystick, button and encoder was operated) it will be outputted in the console.

With this utility you can actually connect multiple panels by listing a series of IP addresses/ports as arguments.

Try the -h option to see what the utility offers:

```
  -autoInterval int
        Interval in ms for demo engine sending out content (default 100)
  -binPanel
        Connects to the panels in binary mode
  -demoModeDelay int
        Seconds before demo mode starts after having manually operated a panel (default
10)
  -exclusiveHWClist string
        Comma separated list of HWC numbers to test exclusively
  -invertCallAll
        Inverts which button edges that triggers 'call all' change of button colors and
display contents. False=Left+Right edge, True=Up+Down+Encoder+None
  -verboseIncoming int
        Verbose input messages to panel (default is none shown). 1=Low intensity, 2=High-
er intensity (protobuf messages as JSON)
  -verboseOutgoing int
        Verbose output from panel, otherwise only events are shown. 1=Low intensity,
2=Higher intensity (protobuf messages as JSON)
```

Test servers for Client Mode written in Python 3

We have written a few Python 3 scripts as well that will help you to get started quickly implementing support for SKAARHOJ panels using Client mode. They can be downloaded from GitHub: https://github.com/SKAARHOJ/Support/tree/master/Files/UniSketchTCPClient

One of these scripts is "TCPserver_colorAndDisplayTest.py" used in the example above. Generally, these scripts are from a time when Client mode was the preferred way to work with Raw Panel, so they are considered legacy, but they are functional.

When you run any of the scripts they will set up a TCP server on the host computer and listen on port 9923. A SKAARHOJ panel working as a UniSketch TCP Client (Raw Panel in client mode) and trying to connect to the IP address of the host computer will interact with the scripts.

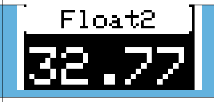
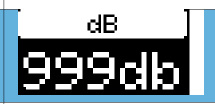
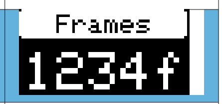
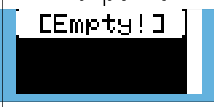
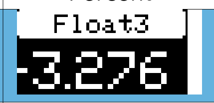
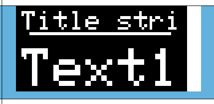
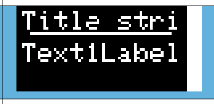
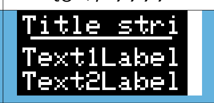
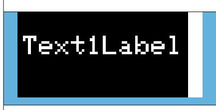
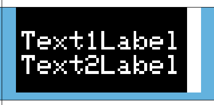
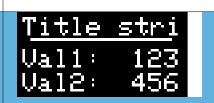
We have put videos on YouTube as well that demonstrates these scripts with panels.

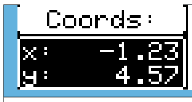
See https://www.skaarhoj.com/support/raw-panel/

Text

Text based graphics

The displays on SKAARHOJ controllers are graphical displays in varying resolutions (see later) - we call them “tiles”. Sometimes many of them are pooled together on a single, larger display, other times they are individual displays.
The easiest way to leverage the displays is to send a string with text / value content to the display. This is done with the command “HWCt#xx=string” as documented in the table of inbound commands. This section lists a number of example strings along with their rendered result. In the table you will find the string that resulted in a given graphic just below the graphic itself. The string is in italics and a comment is given below the string as well:

					
32767 16 bit integer	*-9999* 16 bit integer, negative	*32767|1||Float2* Float with 2 decimal points	*299|2||Percent* Integer value in Percent	*999|3||dB* Integer value in dB	*1234|4||Frames* Integer in frames
					
999|5||Reciproc Reciprocal value of integer	*9999|6||Kelvin* Integer formatted as Kelvin	*9999|7||[Empty!* format 7 = empty!	*-3276|8||Float3* Float with 3 decimal points, optimized for 5 char wide space. Op to +/-9999	*||1|[Fine]|1* Fine marker set (the curvy thing on the right of the line), title as "label"	*||1|Title String* no value, just title string (and with "fine" indicator)
					
|||Title String|1 Title string as label (no "bar" in title)	*|||Title string|1|Text1Label* Text1label - 5 chars in big font	*|||Title string|1|Text1Label||0* Adding the zero (value 2) means we will print two lines and the text label will be in smaller printing	*|||Title string|1|Text1Label|Text2Label* Printing two labels of 10 chars - automatically the size is reduced	*|||Title string|1||Text2Label* Printing only the second line - automatically the size is reduced	*||||Text1Label* Text1label - 5 chars in big font, no title bar.
					
||||Text1Label||0 Adding the zero (value 2) means we will print two lines and the text label will be in smaller printing	*||||Text1Label|Text2Label* Printing two labels - automatically the size is reduced	*|||||Text2Label* Printing only the second line - automatically the size is reduced	*123|||Title string|1|Val1:|Val2:|456* First and second value is printed in small characters with prefix labels Val1 and Val2	*-1234|1||Coords:||x:|y:|4567|2* A box around the first label/value line	*-1234|1||Coords:||x:|y:|4567|3* A box around the second label/value line

					
-1234|1||Coords:||x:|y:|4567|4 A box around the both label/value lines	*-500|1||Coords:||||||1|-1000|1000|-700|700|1* A solid bar scale added below value	*-500|1||Coords:||||||2|-1000|1000|-700|700|2* A moving dot scale added below value			

These graphics are generated from the test utilities or the Python scripts. They can be very useful to experiment with other combinations. A good script to use for testing would be the script "TCPserver_colorAndDisplayTestByButtonPress.py" or the ColorDisplayButtonTest utility (see its source code) as it has a large number of text and image combinations to learn from.

Pipe-delimited string format

The formatting of text in the displays is based on tokenizing the string with vertical pipe (|) and each part starting with index 0 is described in this table.
Notice, the total text string itself cannot be longer than 64 characters. Only ASCII from 32-127 is supported.

Index	internal name	Name	Description
0	_extRetValue[0]	Value, 32 bit integer	Integer value to show in the display. Subject to formatting options in index 1. If empty string, the display format will be set to 7 (value not printed)
1	_extRetFormat (bit 0-3)	Formatting type	Determines how the integer value from index 0 (as well as index 7) is formatted in the display: 0 = as a signed integer 1 = float from 10^3 (X.XX). Deprecated, use format 9 2 = XX% 3 = XXdb 4 = XXf 5 = 1/XX 6 = XXK 7 = Blank (not printed) 8 = float from 10^3 (X.XXX) 9 = float from 10^2 (XX.XX) 10 = one text line (index 0 will be your fortsize, 1-4) rendered from index 3 (title). 11 = two text lines (index 0 will be your fortsize, 1-2) rendered from index 3 (title). 12 = float from 10^1 (XXX.X)
2	_extRetFormat (bit 4-5)	Icon	**Bit 0-1 value (0-3):** 0 = No icon 1 = Fine-flag (speedy wave lines under title bar, right) 2 = Lock icon (in title bar, right) 3 = No Access icon (lower right) **Bit 3-5: (Corner icons below title bar in right side, 8x8 pixels) value (0-7)** (0 : No icon)

			1 (8) : Cycle icon (return arrow) 2 (16): Down (down arrow) 3 (24): Up (Up arrow) 4 (32): Hold (Down arrow pointing to line) 5 (40): Toggle (zig-zag) 6 (48): OK (check mark) 7 (56): Question mark
3	_extRetShort _extRetLong (24 chars)	Title	Sets title of the tile. If title is blank, the title area is not rendered.
4	is label	Label (1) or Value (0, default)	0 = Generates bar behind title (shall indicate that the content shows current value / state) 1 = Line under title (shall indicate that the content shows a description of what the function does)
5	extRetValTxt, 0	First line of text, string 24 chars	
6	extRetValTxt, 1 (enables it also)	Second line of text, string 24 chars	
7	_extRetValue[1]	Value of second line, integer 32 bit	Will be subject to formatting from index 1.
8	_extRetPair	Pair mode, 0-4	0 = Not a pair 1 = A label/value pair is shown: On the first line, index 5 and 0 is shown, on the second line index 6 and 7 is shown 2 = The upper label/value pair is marked 3 = The lower label/value pair is marked 4 = Both label/value pairs are marked
9	_extRetScaleType	Scale type	1 = strength bar (from left) 2 = centered marker 3 = centered bar (from center of range)
10	_extRetRangeLow (integer)	Range low	Low range value
11	_extRetRangeHigh (integer)	Range high	High range value
12	_extRetLimitLow (integer)	Limit low	Limit low marker (set to same as range low if you don't want it. Must be set!)
13	extRetLimitHigh (integer)	Limit high	Limit high marker (set to same as range high if you don't want it. Must be set!)
14	extRetValImage (integer)	Image reference, zero is first image	Reference to an internally stored image (compiled into the firmware from cores.skaarhoj.com)
15	_extRetAdvanced-FontFace		Bit 0-2: General font face, Bit 3-5: Title font face, Bit 6: 1=Fixed Width
16	_extRetAdvanced-FontSizes		Bit 0-1: Text Size H, Bit 2-3: Text Size V, Bit 4-5: Title Text Size H, Bit 6-7: Title Text Size V
17	_extRetAdvanced-Settings		Bit 0-1: Title bar padding, Bit 2-4: Extra Character spacing (pixels)
18	_extRetInvert	Various color settings	Bit 1: Rendering is inverted
19	_extRetPixelColor	Pixel color (only for color displays)	1-18: Raw Panel Indexed colors (see HWCc command for table) Bit 6 (64) enables RGB mode where bit 0-5 is 2-bit RGB values
20	_extRetBckgColor	Pixel color (only for color displays)	1-18: Raw Panel Indexed colors (see HWCc command for table)

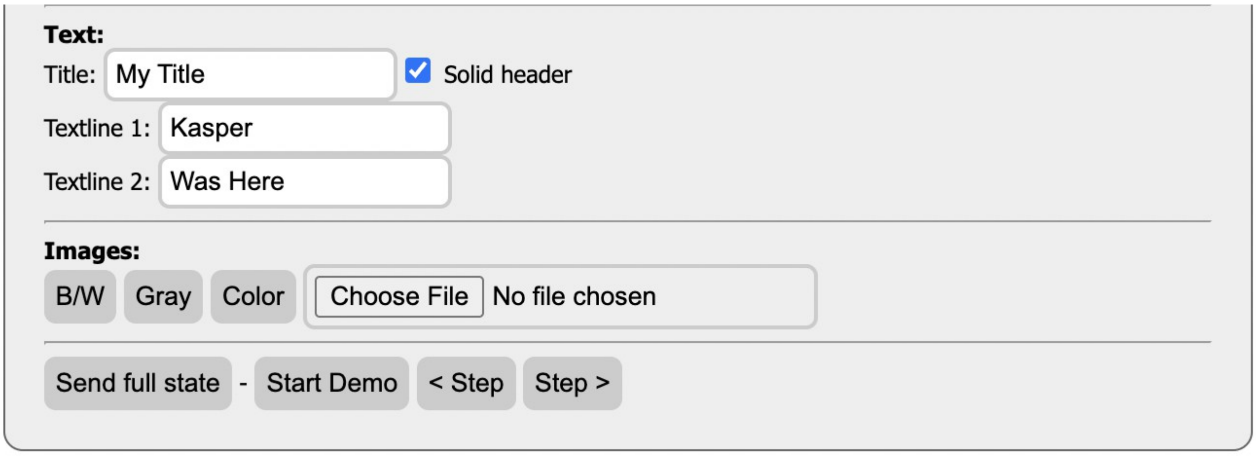
			Bit 6 (64) enables RGB mode where bit 0-5 is 2-bit RGB values

JSON formatted states

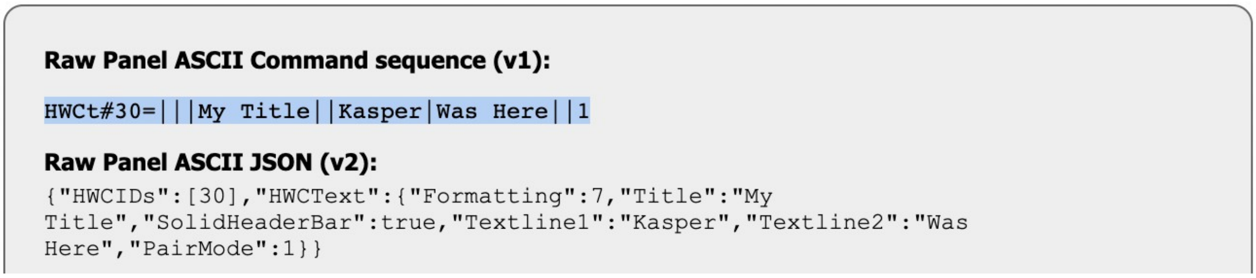
The shortest form of working with text in displays is using the pipe-delimited format, but it may also be the least convenient. It's possible with V2 of Raw Panel to set states using JSON such as the following example:

```
"HWCText": {
    "IntegerValue": 9999,
    "Formatting": 2,
    "ModifierIcon": 5,
    "Title": "Value:",
    "SolidHeaderBar": true
}
```

Also, don't miss out on using the **Raw Panel Explorer** tool provided by SKAARHOJ from the repository raw-panel-explorer. This is also very useful for exploring the text format.



Use this to send various types of feedback to the connected panel.



Protobuf structures in application development

As mentioned elsewhere in this document, the ASCII format shown here has an internal protobuf representation which is far more structured and likely how you could use it in an application. Here is an example of how text for a display is set up in one of our Go applications (Burnin):

```
    txt := rwp.HWCText{}
    txt.Formatting = 7
    txt.Title = fmt.Sprintf("HWc #%d", Event.HWCID)
    txt.SolidHeaderBar = int(Event.HWCID) == displayHWC
    txt.Textline1 = fmt.Sprintf("%s %s", su.Qstr(Event.Binary.Pressed, "Dwn", "Up"),
edgeString)
    txt.Inverted = true
```

If you consider the various formatting types allowed for the text format, you may realize that the format of setting text is aimed at contexts with little or no ability to compile strings with such as Sprintf() like functions. In contexts where your application has ample of room for that, you are likely to not utilize the formatting options a whole lot but rather do it like in the code example above.

Conventions on using displays on SKAARHOJ controllers

SKAARHOJ panels use many different display tile dimensions, and the text based format has been developed to generally adapt to any display associated with a hardware component. It will look great regardless of whether it's a small or large display and theoretically you shouldn't have to know as the sender. However, sometimes you will want to fine tune your content to a given display size. When sending images to a display, knowing its size would be beneficial so you can design the image to the display. The display tile sizes are revealed through the JSON topology data from the panel. Here is a view of the topology as found in the Raw Panel Explorer tool:

61	Title S1	-	-	256x26, Gray, Shrink Bottom	[DISP]	77	OLED Display Tile
62	Title S2	-	-	256x26, Gray, Shrink Bottom	[DISP]	77	OLED Display Tile
63	Title S3	-	-	256x26, Gray, Shrink Bottom	[DISP]	77	OLED Display Tile
64	D56-1	-	-	64x58, Gray, Shrink Right	[DISP]	76	OLED Display Tile
65	D56-2	-	-	64x58, Gray, Shrink Right	[DISP]	76	OLED Display Tile
66	D56-3	-	-	64x58, Gray, Shrink Right	[DISP]	76	OLED Display Tile
67	D56-4	-	-	64x58, Gray	[DISP]	76	OLED Display Tile
68	D56-5	-	-	64x58, Gray, Shrink Right	[DISP]	76	OLED Display Tile
69	D56-6	-	-	64x58, Gray, Shrink Right	[DISP]	76	OLED Display Tile
70	D56-7	-	-	64x58, Gray, Shrink Right	[DISP]	76	OLED Display Tile
71	D56-8	-	-	64x58, Gray	[DISP]	76	OLED Display Tile
72	D56-9	-	-	64x58, Gray, Shrink Right	[DISP]	76	OLED Display Tile
73	D56-10	-	-	64x58, Gray, Shrink Right	[DISP]	76	OLED Display Tile

Tile sizes

The typical tiles you will find on a SKAARHOJ controller are these:

Tile size	Comment
64x32	The most typical tile size you will find on a SKAARHOJ controller! This is the reference and any content should render reasonably on this tile size.
112x32	A wide tile type mostly found as a display for encoder knobs, but also found for some buttons
64x48	A type of tile with larger display area than the classic 64x32 pixel tiles
256x20	Wide title line - on large controllers
52x24	Mini tiles, 24 pixels high and more narrow than the standard tiles. Used on large controllers as labels for buttons.
48x24	Mini tiles, 24 pixels high and more narrow than the standard tiles. Used on large controllers as labels for buttons.
128x32	Tile size seen once in a while. For example on the RCP (ID display)
64x38	Color OLED display (physically 64x48 pixels, active area 64x38) on a NKK SmartSwitch
86x48	Color OLED display (physically 96x64 pixels, active area 86x48) on a NKK SmartSwitch
96x64	Color OLED display 94x64 pixels.
128x36	Tiles on a Blue Pill display
128x72	Tiles on a Blue Pill display

For your information: When text based tiles are placed next to each other on shared displays, the tile is rendered one pixel smaller in the relevant dimension and a blank row or column of pixels is placed between them to the right and bottom (called shrinking). Does not apply to graphics.

"Labels" or "Values"

Another significant convention is how the title bar on a tile should be rendered. The flag "is label" (index 4) is used to determine if the title area is rendered as a solid bar (is label = 0, default) or if it's rendered as a string of text with a line under (is label = 1).

This is how you should use it:

- If the display shows the **current value** of anything - the "status" -, then set "is label" to 0 so it renders as a **solid bar.** An example is if the display shows the current source name on an Aux bus or if it shows the current state "on" for a given feature. Typically this will be the case for cycling buttons, encoders or toggle buttons.
- If the display shows the **label of a function** - what it will "do" - then set "is label" to 1 so it renders with just a line under. An example is if the display shows the source name that you will route to the aux bus if pressed, or shows "on" because a button press will actually turn something "on". Typically this is the case for non-cyclic and non-toggle buttons.

Bitmap Graphics

Totally custom pixel graphics are another format you can use to generate content for the displays. Find sample graphics here:
https://github.com/SKAARHOJ/Support/tree/master/64x32_Graphics

Protobuf format

Working with images is easy with the protobuf structures and even with JSON used to set states via the ASCII V2 Raw Panel format . Here is an example of sending a monochrome 48x24 pixel graphic to hardware component #34:

```
{
    "HWCIDs": [
        34
    ],
    "HWCGfx": {
        "W": 48,
        "H": 24,
        "ImageData": "////////////////0IQhCEIT0IQhCEIT0IQhCEIT/AAf+A+f0AABAAAD-w4/AB+Bzw4zABvBzx5huYDDzz4zG4GHz2Y/DweMz2Y3Dg4Mz/9zjgwf7/9znxgf7wY/G5+AzwYeMZ+Az0AAAAAAD0AAAAAIT0IQhCEIT////////0IQhCEIT////////////////"
    }
}
```

ASCII V1 Format

The ASCII V1 format for sending images is far more cumbersome. Luckily you may not necessary have to deal with this yourself since the library code in raw-panel-lib can handle it for you, but anyway, here is a description:

To facilitate images in varying sizes, you need to encode the image over a given amount of lines, having an individual length no longer than around 250 characters (corresponding to 170 bytes of image data)

The format is this for the starting line (sequence number zero):
HWCg#*[HWC]*=*[sequence number, zero is first]*/*[Last number in sequence]*,*[width, pixels]*x*[height, pixels]*,*[x-coordinate from upper left]*,*[y-coordinate from upper left]*:*[base64 encoded data]*

The x and y coordinates (blue) are optional and if left out (or equal to the default value of -1) the image will be centered in that dimension.

For subsequent lines it looks like this:
HWCg#*[HWC]*=*[sequence number]*:*[base64 encoded data]*

Example:

```
# TEST 64x38
'HWCg#{}=0/15,64x38://///////////////////////8QhCA==',
'HWCg#{}=1:QhCEIQvEIQhCEIQhC////////w==',
'HWCg#{}=2:///EIQhCEIQhC8QhCEIQhCELxA==',
'HWCg#{}=3:IQhCEIQhC8QhCEIQhCEL/////w==',
'HWCg#{}=4://///8QhCEIQhCELxCEIQhCEIQ==',
'HWCg#{}=5:C8QBCEIQAAELxAAAAgAAAQv8fg==',
'HWCg#{}=6:AwAHwfH/xP4HAA/juQvAwA8AAA==',
```

```
'HWCg#{}=7:YxgLwMAfHMBjGAvA/BsPgOO4Cw==',
'HWCg#{}=8:+P4zD4fB8P/AxncHAGO4C8DGfw==',
'HWCg#{}=9:hwBjGAvAxgcPgGMYC8TuAx3P4w==',
'HWCg#{}=10:uQv8fAMYz8Hx/8QAAAAAAAELxA==',
'HWCg#{}=11:AQgAAAABC8QhCEIQhCELxCEIQg==',
'HWCg#{}=12:EIQhC///////////xCEIQhCEIQ==',
'HWCg#{}=13:C8QhCEIQhCELxCEIQhCEIQvEIQ==',
'HWCg#{}=14:CEIQhCEL///////////EIQhCEA==',
'HWCg#{}=15:hCEL/////////////////////w==',
```


where {} is the HWC number.

Please see "TCPserver_colorAndDisplayTestByButtonPress.py" for examples. This is also how a function like InboundMessagesToRawPanelASCIIstrings() will convert a protobuf message to ASCII V1 strings that a UniSketch panel can parse.

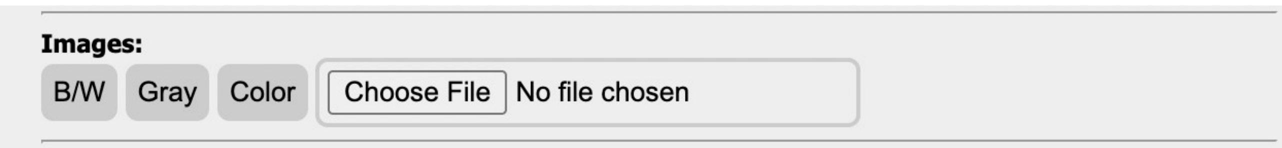
RGB and Grayscale

If your SKAARHOJ controller integrates grayscale or color displays you can send graytone and RGB images to it and set background color and pixel color.
Sending such images is done using the command "HWCgRGB" (5+6+5=16bit => 2 bytes pr. pixel) or "HWCgGray" (4bit => 2 pixels per byte). Please see "TCPserver_colorImages.py" for examples.

Important disclaimer: Color displays and RGB images take up 16 times more space and processing power on the controllers than the standard monochrome displays does, so the practical frame rates are lower. It's therefore important to have realistic expectations to what can be achieved with color displays, especially on UniSketch panels. Anecdotally; you can achieve to send over 6 RGB images of 96x64 pixels a second to a UniSketch panel. That's far from showing moving pictures on a display tile.
On Blue Pill panels the frame rates are much higher but often sharing the same data bus. In reality on Blue Pill panels you can easily show moving pictures on a few tiles at a fair framerate.

Conversion

With the Raw Panel Explorer tool you are able to generate graphics that fits the exact pixel dimensions of any display on your panel! This is simply a matter of uploading the image:



The image will get scaled (stretched) to the exact display dimensions so if you want to avoid that, prepare the graphic dimensions you upload on beforehand. The output is shown under the field in the tool:

Raw Panel ASCII Command sequence (v1):


```
HWCg#46=0/2,64x48://////////////////////////////////////////////+//////////7/////////////////
HWCg#46=1:fnNln/////4eeAFfD////i9gQL9/////dwAAHP/////zAMBZ//////mAAHP//////IAGv//////+QA8
HWCg#46=2:///////////////////////////////////////////////////////////8=
```


Raw Panel ASCII JSON (v2):


```
{"HWCIDs":[46],"HWCGfx":
{"W":64,"H":48,"ImageData":"//////////////////////////////////////////////+//////////7/////
```


Raw Panel Binary Protobuf command:

(not rendered)

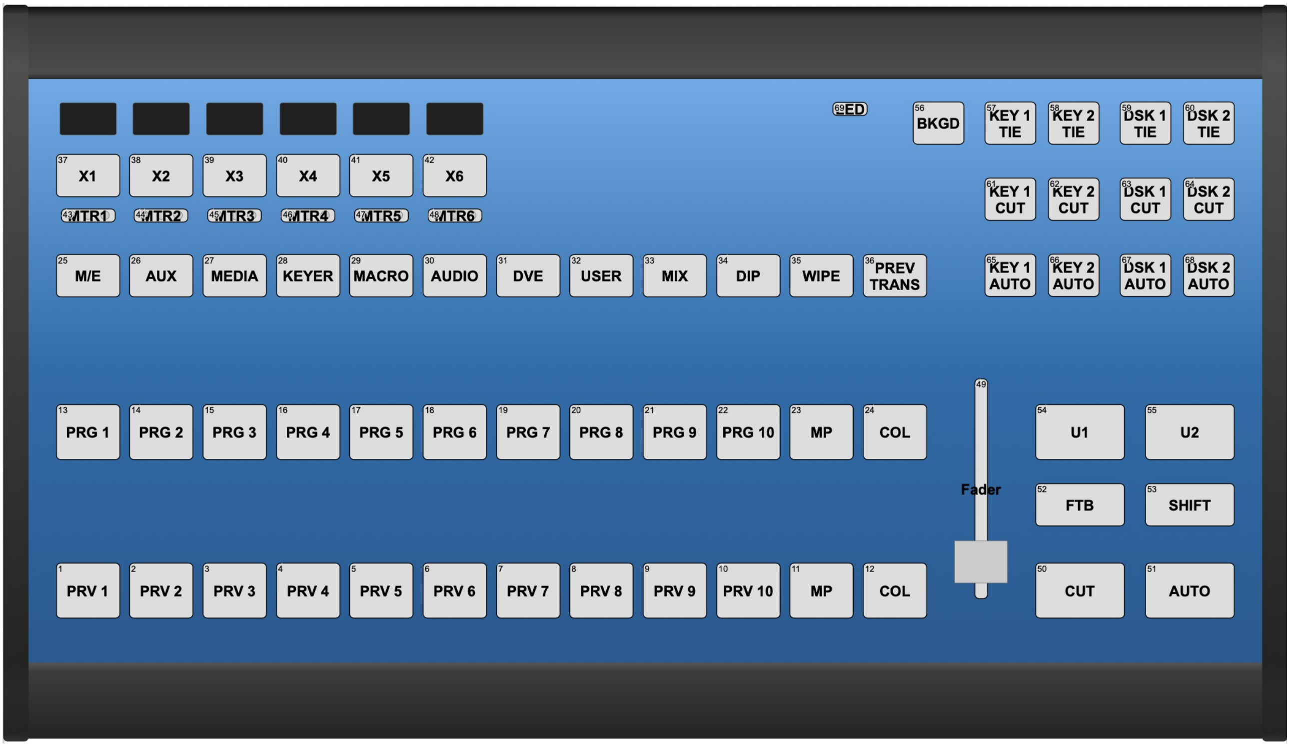
There is also a conversion tool in raw-panel-utils called **ImageConverter**. This command line tool allows a little more control of scaling etc based on some options.

SKAARHOJ once hosted an online tool for image conversion. We don't do that anymore as the above mentioned applications should be sufficient and more powerful even.

Topology

A powerful concept with Raw Panel is the ability to query the panel for its topology. This is delivered as an SVG background graphic and a JSON data structure that documents the hardware components on the controller in a way that will allow you to render a beautiful configuration interface. The JSON holds visual information as well as various properties for each hardware component.

Here is an example of an Air Fly controller:



SVG background

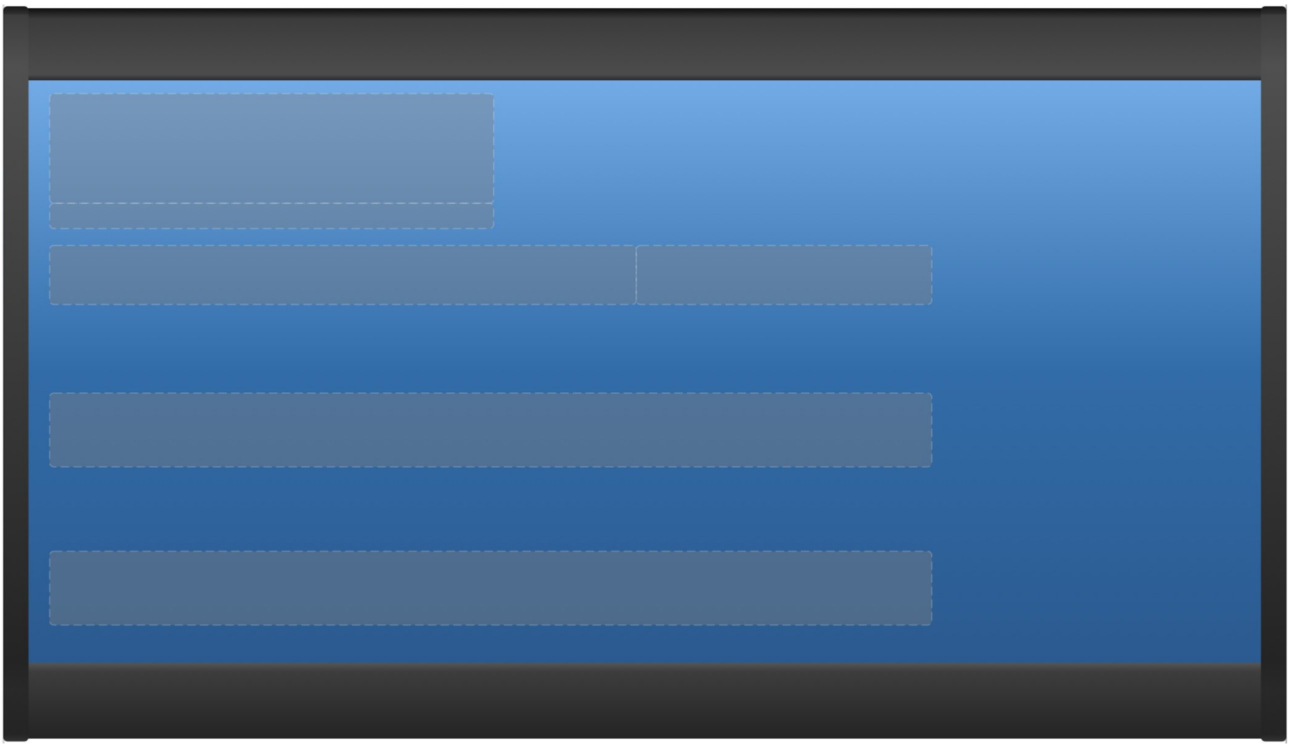
The SVG background for the Air Fly rendered above would look like this:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 3040 1858" width="100%" id="ctrlimg" style="display:block;">
                 <defs>
                     <linearGradient id="frontplate" x1="0%" y1="0%" x2="0%" y2="100%">
                       <stop offset="0%" style="stop-color:rgb(99, 171, 235);stop-opacity:1" />
                       <stop offset="50%" style="stop-color:rgb(25, 108, 173);stop-opacity:1" />
                       <stop offset="100%" style="stop-color:rgb(26, 90, 147);stop-opacity:1" />
                     </linearGradient>
                     <linearGradient id="topedge" x1="0%" y1="0%" x2="0%" y2="100%">
                       <stop offset="0%" style="stop-color:rgb(13,13,13);stop-opacity:1" />
                       <stop offset="8%" style="stop-color:rgb(39,39,39);stop-opacity:1" />
                       <stop offset="15%" style="stop-color:rgb(60,60,60);stop-opacity:1" />
                       <stop offset="92%" style="stop-color:rgb(76,76,76);stop-opacity:1" />
                       <stop offset="100%" style="stop-color:rgb(40,40,40);stop-opacity:1" />
                     </linearGradient>
                     <linearGradient id="bottomedge" x1="0%" y1="0%" x2="0%" y2="100%">
                       <stop offset="0%" style="stop-color:rgb(86,86,86);stop-opacity:1" />
                       <stop offset="12%" style="stop-color:rgb(61,61,61);stop-opacity:1" />
                       <stop offset="92%" style="stop-color:rgb(38,38,38);stop-opacity:1" />
                       <stop offset="100%" style="stop-color:rgb(37,37,37);stop-opacity:1" />
                     </linearGradient>
                     <linearGradient id="sides" x1="0%" y1="0%" x2="0%" y2="100%">
                       <stop offset="0%" style="stop-color:rgb(24,24,24);stop-opacity:1" />
                       <stop offset="1%" style="stop-color:rgb(47,47,47);stop-opacity:1" />
                       <stop offset="1.2%" style="stop-color:rgb(62,62,62);stop-opacity:1" />
                       <stop offset="8%" style="stop-color:rgb(80,80,80);stop-opacity:1" />
                       <stop offset="89%" style="stop-color:rgb(36,36,36);stop-opacity:1" />
                       <stop offset="93%" style="stop-color:rgb(41,41,41);stop-opacity:1" />
                       <stop offset="99%" style="stop-color:rgb(36,36,36);stop-opacity:1" />
                       <stop offset="100%" style="stop-color:rgb(52,52,52);stop-opacity:1" />
```

```
            </linearGradient>
          </defs>
          <style>
                        text {font-family:Sans,Arial;}
            .sections {
                stroke: white;
                stroke-width: 3;
                fill: gray;
                fill-opacity: 0.4;
                stroke-opacity: 0.2;
                stroke-dasharray: 20, 10;
                   }
          </style>
        <rect width="2940" height="172" x="50" y="67" style="fill:url(#topedge);" />
        <rect width="2940" height="1380" x="50" y="239" style="fill:url(#frontplate);" />
        <rect width="2940" height="180" x="50" y="1619" style="fill:url(#bottomedge);" />
        <rect width="60" height="1743" x="0" y="63" style="fill:url(#sides);" rx="10" ry="10" />
        <rect width="60" height="1743" x="2980" y="63" style="fill:url(#sides);" rx="10" ry="10" />

        <rect width="1052" height="260" x="110" y="270" rx="10" ry="10" class="sections"/>
        <rect width="1052" height="60" x="110" y="530" rx="10" ry="10" class="sections"/>
        <rect width="1390" height="140" x="110" y="630" rx="10" ry="10" class="sections"/>
        <rect width="700" height="140" x="1500" y="630" rx="10" ry="10" class="sections"/>
        <rect width="2090" height="175" x="110" y="980" rx="10" ry="10" class="sections"/>
        <rect width="2090" height="175" x="110" y="1355" rx="10" ry="10" class="sections"/>
</svg>
```

It would look like this:



JSON Topology Data

The JSON data for the hardware components has two keys on the first level, "HWc" and "typeIndex". Here a few examples from the Air Fly is shown (first three hardware components):


```
"HWc": [
    {
        "id": 1,
        "x": 201,
        "y": 1449,
        "txt": "PRV 1",
        "type": 132
    },
    {
        "id": 2,
        "x": 374,
        "y": 1449,
        "txt": "PRV 2",
        "type": 132
    },
    {
        "id": 3,
        "x": 548,
        "y": 1449,
```

```
        "txt": "PRV 3",
        "type": 132
    },
    .....
```

Below you will find the reference for the topology:

Key	Description
id	The HWC id of the component. It's important to understand this id in relation to the "map=zz:xx" command: This id - the native ID - will correspond to "zz" in the map command. Keep in mind that strictly, you are not supposed to render a hardware component unless you have received a map= command for it! The map command basically confirms that the hardware component is configured to send you data (it may not always be since it may belong to an option not installed in a given controller). **More background about map=** If you receive a map from the controller like "map=23:23" it means that hardware componet with id 23 will send commands as "id 23". (However, if you receive a map command like "map=23:47" it means that whenever hardware component with id 23 is triggered, the commands will be sent from the controller as if they came from id 47).
x y	These are the x / y coordinates of the hardware component on the controllers SVG graphic. Unit: 1/10th of a millimeter.
txt	This is the default label of the hardware component. You are invited to let users edit it in your application.
type	This number is a reference to a type of component and it's an index into the "typeIndex" part of the JSON. *Notice:* This value shall be considered valid as a reference only within the JSON structure at hand; the value is allowed to be a random integer in the next received topology and any value is not guaranteed to describe the same component in a different topology. In reality, you will find that type numbers are mostly used consistently for the same component, even across model IDs.
typeOverride	If you find keys in here for a given HWC it's meant to override the same key in the typeIndex. This allows you to make customizations and extensions on a per-HWC basis. Example: `{ "id": 69, "x": 1312, "y": 912, "txt": "D58-6", "type": 76, "typeOverride": { "disp": { "w": 64, "h": 58 } } },` In this case, typeOverride is used to indicate the display dimensions is 64x58. It was likely necessary because the type 76 has other pixel dimensions that didn't fit in this case. Example: `{ "id": 9, "x": 180, "y": 909, "txt": "Knob A", "type": 15, "typeOverride": { "disp": { "w": 64, "h": 32, "subidx": 0 },`

<table>
<tr><th>Key</th><th>Description</th></tr>
<tr><td></td><td>
<pre>
                "sub": [
                    {
                        "_": "r",
                        "_x": -55,
                        "_y": -550,
                        "_w": 190,
                        "_h": 95,
                        "rx": 5,
                        "ry": 5,
                        "style": "fill:rgb(33,33,33);"
                    }
                ]
            }
        },
</pre>
In this case, typeOverride is used to add a whole "sub" sections which didn't even exist plus set the display dimensions and refer to the first sub element (index 0) to represent the display in the drawing.
The rules for overriding this:

Keys W, H, and Subidx: If larger than zero, they will override
Keys Out, In, Ext: If not blank, they will override
Keys Rotate: If different from zero (float), it rotates
Keys Disp and Sub: If they contain a configuration with non-zero / non-blank values, they will override.

</td></tr>
</table>

The "typeIndex" part describes each hardware component type used on the controller. The numbers won't change between controllers. All SKAARHOJ controllers would carry the same set of data about a given number (assuming the same firmware version).

The "typeIndex" looks like this (example):

```
    "typeIndex": {
        "15": {
            "w": 160,
            "out": "rgb",
            "in": "pb",
            "desc": "Encoder"
        },
        "28": {
            "w": 30,
            "h": 710,
            "in": "av",
            "ext": "pos",
            "subidx": 0,
            "desc": "Motorized Fader 60mm",
            "sub": [
                {
                    "_": "r",
                    "_x": -63,
                    "_y": 53,
                    "_w": 125,
                    "_h": 250
                }
            ]
        },
        "36": {
            "w": 570,
            "h": 151,
            "disp": {
                "w": 128,
                "h": 32
            },
            "desc": "OLED Display Tile"
        },
        "40": {
            "w": 250,
            "h": 40,
            "in": "gpi",
            "desc": "Opto-isolated Input (to GND)"
        },
....
```

The first type, "15", is an encoder. Since only "w" is given, it must be rendered as a circle with the diameter 160. "out" indicates that it can accept RGB color information (background LED ring most likely) and "in" has the value "pb" which means "pulses + button" which corresponds to an encoder with push function.

The second type, "28" has both "w" and "h" and by convention should be rendered as a rectangle. The "in" (input type) indicates with "av" that it's an absolute component oriented vertically (like a T-bar). "ext" has the value "pos" which indicates that sending the extended return value back will let the component position itself (which makes sense, since the description reveals it's a motorized fader). The "sub" element holds additional data when the component has more visual elements than it's bases circle or rectangle. This is the case with many components that has displays for example. Or sliders that tend to have a rectangle on top to represent the handle/knob. In the "sub" element, "_" tells us it's a rectangle we should draw in position "_x","_y" with "_w" and "_h" for width and height.

Type "36" is just a display. It has no indication of input or output type. It should be rendered as a rectangle and we are told its pixel dimensions is 128x32.

Type "40" is a GPI input. The "in" type is set to "gpi"

Key	Description
in	Input type: • b = Standard button • b4 = Four-way button • b2v = Two-way button, vertical (top/bottom) • b2h = Two-way button, horizontal (left/right) • gpi = GPI trigger • pb = encoders (pulses + button) • p = encoders (pulses, no button) • av = Absolute vertical (Faders) • ah = Absolute horizontal • ar = Absolute rotation (Potentiometers) • a = Absolute, direction unspecified • iv = Intensity vertical (Joysticks) • ih = Intensity horizontal (Joysticks) • ir = Intensity rotation • i = Intensity, direction unspecified Parse input type by splitting with a comma. There may be more parameters to it, for example "ar,steps=16" could indicate an analog component which would have only 16 steps in its input value (like a binary selector)
subidx	A reference to the index of an element in the "sub" element which has a "special" meaning. For analog (av, ah, ar) and intensity (iv, ih, ir) elements, this would be an element suggested for being used as a handle for a fader or joystick. This value is implicitly zero if not found in the JSON data and therefore always pointing to the first sub element (if it exists). A value of -1 explicitly indicates that no sub element is referenced.
out	Output type: • gpo = GPO output • mono = mono LED / LEDBAR • rg = Red/Green LED / LEDBAR • rgb = RGB colored LED / LEDBAR
disp	Indicates display dimensions in keys "w" and "h". Furthermore you can find the "type" key set to a value "gray" (4bit/pixel) or "color" (5-6-5 rgb/pixel) to indicate the display capability. The default is black background with white pixels.

<table>
<tr><th>Key</th><th>Description</th></tr>
<tr><td></td><td>If the type is "text", it means the display has no pixel canvas but takes text strings via the HWCText element, prioritizing Textline1. If w is set (>0), it indicates a limited number of characters shown, if h is set (1-3) it indicates the number of lines supported, prioritized as Textline1, Title, Textline2
Finally, the key "subidx" indicates an index of an element in the "sub" element which shall represent the display of the component instead of the main component. This is used a lot as many components not being displays themselves has their display offset from their center. Notice that implicitly subidx is zero and therefore always pointing to the first sub element if it exists. If you want to explicitly not point to any sub element, the value -1 will indicate that.</td></tr>
<tr><td>ext</td><td>Support for extended return values:
pos - indicates a self-positioning components, like a motorized fader
steps - indicates an element that can respond to steps, like an LED bar. The number of steps shall be determined by iterating over elements inside "sub" and look for their "_idx" property which determines the order they should be used in. The number of steps supported should be determined by finding the highest and lowest value of _idx.</td></tr>
<tr><td>desc</td><td>Description</td></tr>
<tr><td>sub</td><td>Inside "sub" you will find one or more sets of properties, each one representing additional SVG elements to be rendered.

Example:<pre>"sub": [
            {
             "_idx": 1,
                "_": "r",
                "_x": -67,
                "_y": -190,
                "_w": 134,
                "_h": 76,
                "rx": 5,
                "ry": 5,
                "style": "fill:rgb(33,33,33);"
            }
        ]</pre>
"_" - if "r" means render SVG rectangle, if "c" means SVG circle
"_x", "_y" - the x/y offset of this SVG element from the component center
"_w", "_h" - the width/diameter and height of the element
Any key NOT prefixed with "_": Is added as a standard SVG element attribute, hence "rx", "ry" and "style" in the above example would be added as such elements. For some known applications, only "r", "rx", "ry" and "style" is supported here.</td></tr>
</table>

The Dynamics of a Topology

When you retrieve a topology from a panel you must expect that it can change dynamically: a panel may send you a different topology next time you connect, and it may even push you an updated topology as you are connected. You cannot make the assumption that a panels with the same model ID has the same topologies.

Furthermore, changes in the "map" should show/hide topology elements. Such changes may happen as a panel goes to sleep or wakes up or it may be of a more stable nature where a panel includes elements in its topology which are disabled through the map since they are not available on that particular panel (but may be on other panels of the same type for whatever reason).

The *type* (or type index) used as a schema to describe the same type of component within a topology can also change its type number between topologies. Mostly, it will stay stable and in most cases even describe the same component across panel models, but it's allowed to change as long as it functions as an internal reference in a given JSON structure.

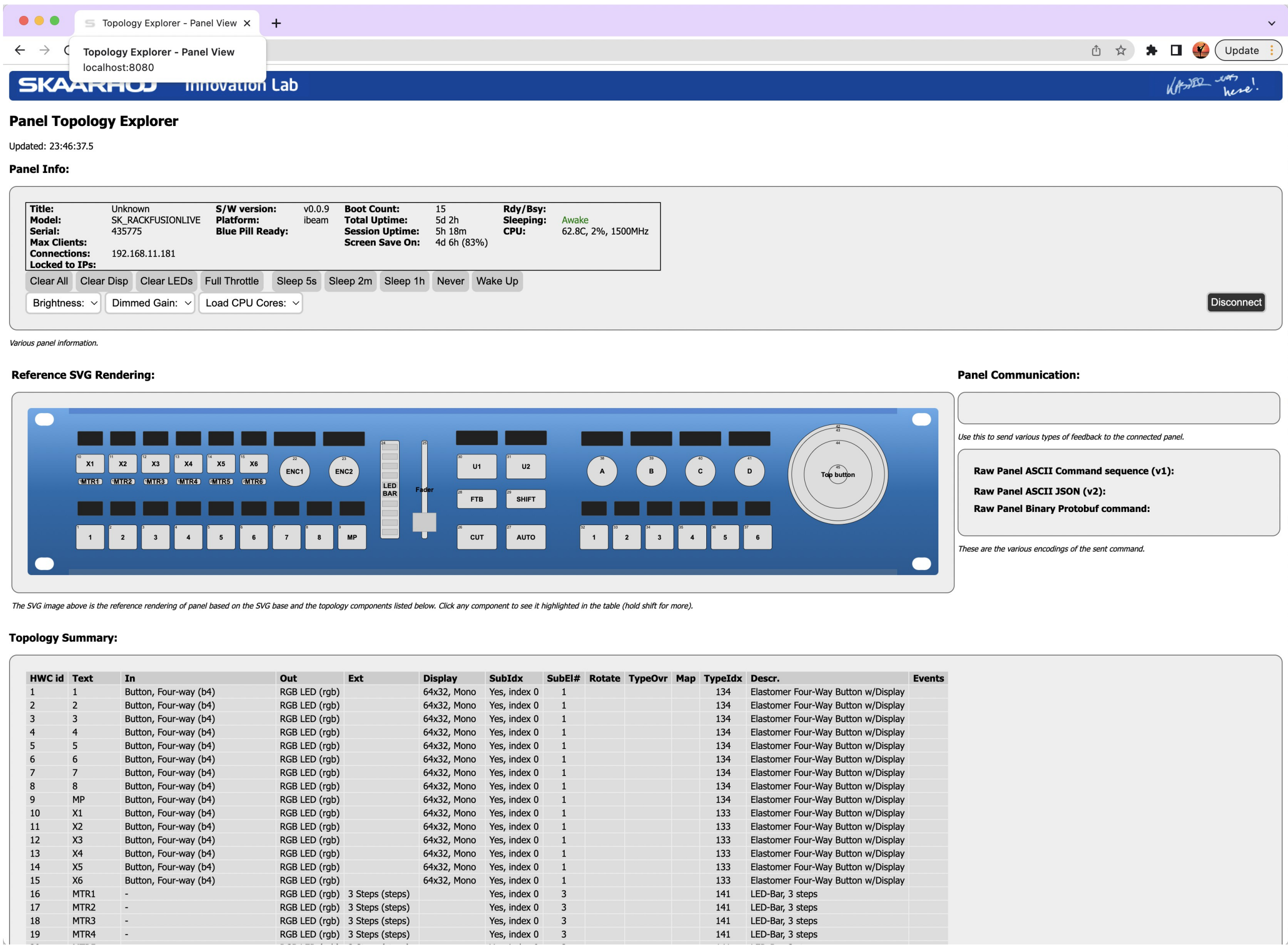
These are suggestions on how to handle dynamic topologies:

- For any currently connected panel, always use the latest topology sent from the panel to you (ask for it on connect and if you receive it during connection at the initiative of the panel, use that going forward)
- For offline panels, you may cache the latest known topology for that panel. It's suggested to bind that to the serial number of the panel rather than the model ID.
- For keeping a library of discovered topologies linked to model IDs, you may decide to store the first-ever or last received topology, but inherently you cannot know for sure that it will even be relevant as a highly dynamic panel topology wouldn't make any sense to store like that. However, the vast majority of Raw Panels will very likely have a stable topology so in reality, it will often (appear to) work fine.
- Updates to the map should also trigger updates to your rendered topology. However, you may consider if a map is changed at the event of a panel going to sleep or if it's happening in response to a configuration change that enables/disables elements. In the case of a sleeping panel you may want to not change the topology.
- Do not render a component with anything but data coming from the JSON structure at hand. If you make assumptions about a type number and start augmenting it, you may run a risk that it changes with no notice and invalidates your rendering.

Reference for Rendering

The repository *rawpanel-lib* from SKAARHOJ contains a module, *topology*, which contains the final reference with respect to properties and rendering. It should be consistent with information above but can be used to also provide more information implicitly on how to interprete the various topology data.

In the repository *raw-panel-explorer* you will find a utility with source code, **Raw Panel Explorer**, which shows how the topology and SVG data shall be rendered into a full controller view. This tool even provides an interactive web based view of the topology of a controller you connect to.



HWC id	Text	In	Out	Ext	Display	SubIdx	SubEl#	Rotate	TypeOvr	Map	TypeIdx	Descr.	Events
1	1	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
2	2	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
3	3	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
4	4	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
5	5	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
6	6	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
7	7	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
8	8	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
9	MP	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				134	Elastomer Four-Way Button w/Display	
10	X1	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				133	Elastomer Four-Way Button w/Display	
11	X2	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				133	Elastomer Four-Way Button w/Display	
12	X3	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				133	Elastomer Four-Way Button w/Display	
13	X4	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				133	Elastomer Four-Way Button w/Display	
14	X5	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				133	Elastomer Four-Way Button w/Display	
15	X6	Button, Four-way (b4)	RGB LED (rgb)		64x32, Mono	Yes, index 0	1				133	Elastomer Four-Way Button w/Display	
16	MTR1	-	RGB LED (rgb)	3 Steps (steps)		Yes, index 0	3				141	LED-Bar, 3 steps	
17	MTR2	-	RGB LED (rgb)	3 Steps (steps)		Yes, index 0	3				141	LED-Bar, 3 steps	
18	MTR3	-	RGB LED (rgb)	3 Steps (steps)		Yes, index 0	3				141	LED-Bar, 3 steps	
19	MTR4	-	RGB LED (rgb)	3 Steps (steps)		Yes, index 0	3				141	LED-Bar, 3 steps	

Discovery

Your SKAARHOJ panels, whether it's on UniSketch or Blue Pill will announce its presence on the network with mDNS (Bonjour/ZeroConf).

Here are two examples from a discovery tool. It finds two Rack Fusion Live controllers, serial numbers 435775 and 491034, on the network. The one with serial 435775 announces that it uses protocol "rwp2.0B" which is Raw Panel protocol V2 binary - hence this would be a Blue Pill panel. The Rack Fusion Live with serial 491034 doesn't reveal protocol information which makes us assume it's ASCII V1 format and thereby a UniSketch panel.

```
v _skaarhoj-rwp._tcp. - 5 items
  > SK_BLUEPILL [443131]
  > SK_INLINE10 [433322]
  v SK_RACKFUSIONLIVE [435775]
      435775.SK_RACKFUSIONLIVE.skaarhoj.local.
      192.168.11.8:9923
      [fe80::dea6:32ff:fedb:76e6]:9923
      protocol=rwp2.0B
  v SK_RACKFUSIONLIVE [491034]
      491034.SK_RACKFUSIONLIVE.skaarhoj.local.
      192.168.11.9:9923
  > SK_RCPV2 [491578]
```

The **protocol** field could be rwp2.0A (ASCII), rwp2.0B (Binary) or rwp2.0A/B (Auto detecting protocol mode)

The PanelTopology tool will show you an index of panels on the network based on mDNS/zero-conf/bonjour.

Auto Detection Handshake

SKAARHOJ produced Raw Panels in protocol auto detect mode will determine the protocol type after receiving four bytes of content from the newly connected system. If those bytes doesn't look like a realistic binary header (package length), it will set ASCII as the protocol mode for the remainder of the TCP session. Until the protocol is determined this way, the panel will assume binary protocol (including sending any triggers that occur as binary). It's therefore recommended to send a command like "list" immediately after connection to force the panel to settle the protocol.
If the connecting system itself is also auto detecting protocol mode, it's *highly* recommended to start probing with a binary command, such as "list". This will secure that auto detecting panels and systems will negotiate to use the most efficient protocol encoding (binary).

UniSketch Panels

UniSketch panels with Raw Panel deserves a special mention. First of all because the Raw Panel Protocol was originally developed as a device core (UniSketch TCP Client) for UniSketch panels and secondly because on UniSketch, Raw Panel can do more and different things than what is the main promoted usage today, where Raw Panel is the universal language that binds all of SKAARHOJ's platforms together.

Take the Blue Pill

Blue Pill Direct Mode is a state where any UniSketch panel regardless of configuration will instead run a full Raw Panel server. The quickest way to get your UniSketch panel into Blue Pill Direct Mode is to press the key in the lower left corner of the panel twice when you see the color animations during boot. This will enable - and disable - Blue Pill Direct Mode on the panel without affecting your configuration on the panel. It's confirmed by a blue/white LED swipe across the panel and a reboot. You can also type "TakeTheBluePill" in the serial monitor (and type or press Reset to exit it again).

The serial monitor will confirm Blue Pill Direct Mode like this during boot:

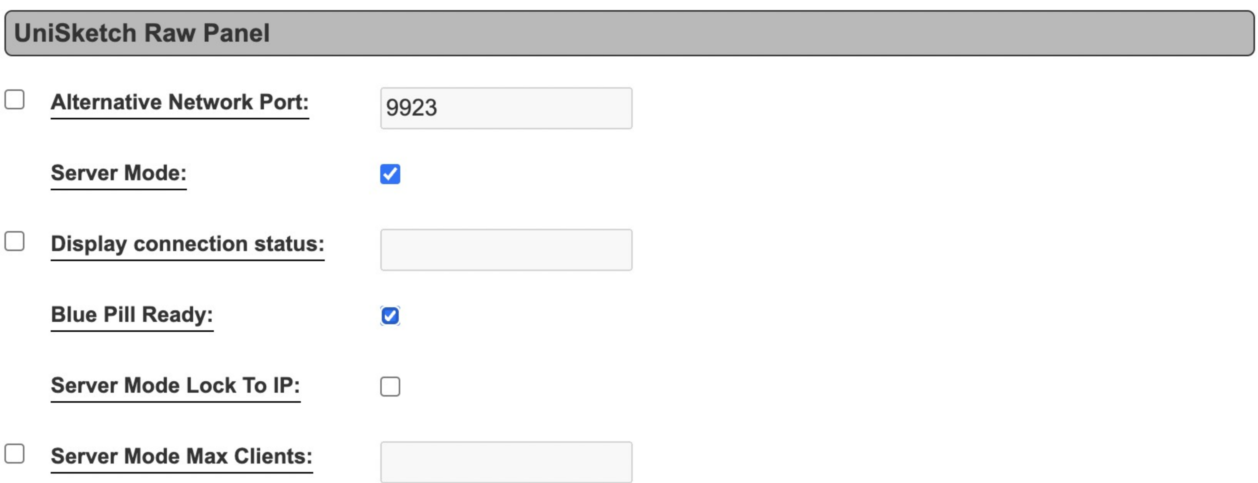
```
*****
Blue Pill DIRECT MODE enabled (DHCP + Raw Panel Server Mode on port 9923)
*****
DeviceCore #1: UniSketch TCP Client1, IP = 0.0.0.0:9923
UNISKETCHTCPCLIENT (RawPanel): Blue Pill Ready: Yes
UNISKETCHTCPCLIENT (RawPanel): Server Mode = ON
```

Notice, in Blue Pill Direct Mode your panel will expect to get an IP address from a DHCP server.

It's highly recommended to *only* use Blue Pill mode for adhoc tests while any stable connection should be achieved with loading a true Raw Panel configuration onto the panel with the appropriate device core options set.

Device Core Options

If you intend to use your UniSketch panel with Blue Pill Reactor you will want it to comply with the applied standards for Raw Panel in this context. When setting up your UniSketch panel, make sure to use these Device Core settings:



- Server Mode is mandatory. You are invited to consider Lock to IP and Max Clients to "protect" your panel from additional connections from other systems. This is a semi security feature as well as a way to keep accidental multi-master scenarios from occurring.
- Blue Pill Ready will adjust how encoder button presses are sent and how they respond to LED color. By default encoders are differently handled on UniSketch than buttons are, but with this set, they will work similarly which is expected from Reactor. Also, other aspects may be adjusted to create compliance.

Legacy Device Core options

On the media page for panel configuration on cores.skaarhoj.com you can also find device core options being set via a legacy format. This is deprecated, but yet, here is how it works:

Device configuration options exist:

- Index 0: **Port number**: If different from 0, then this is the port number the controller will try to connect to on the device core IP (or the port number of the TCP server in server mode)

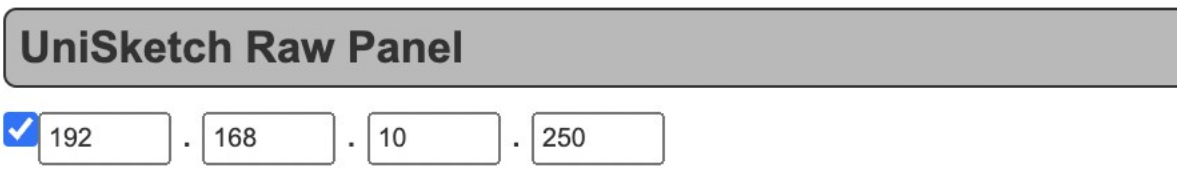
Example: If two UniSketch TCP Client device cores are active on the same IP 192.168.10.250, then setting "D1:0=9234" will mean that the second device core (because of "1") will try to connect on port 9234 instead of port 9923.

- Index 1: **Server Mode**: If set to 1 the device core will not try to connect to as a TCP client but rather set up a TCP server on port 9923 (or the port defined by device core index 0) and allow up to 8 external TCP clients to connect and interact with it. In this case, the IP address of the device core will not matter of course.

Example: Setting up UniSketch TCP Client (assuming it's the first device core (zero)) in server mode, listening on port 9930 "D0:0=9930;D0:1=1".

Handshaking in Client Mode

A SKAARHOJ UniSketch controller with the "UniSketch TCP Client" device core will need to be set up with an IP address and it will attempt a connection to this IP address on port 9923.



All communication forth and back is ASCII and terminated by <NL> (newline, "\n")

After the TCP server responding on port 9923 accepts the connection, it will receive the command "**list<NL>**" from the UniSketch TCP Client. In response to this command, the server must respond with any initial data it wishes to dump followed (or preceded) by "**<NL>ActivePanel=1<NL>**" (Notice: text and graphics must come after "<NL>ActivePanel=1<NL>" is sent, in fact text and graphics should probably respond to the "map" command). This will confirm to the UniSketch TCP Client that it has been initialized and it will start to evaluate actions for the panels hardware interface components.

Periodically (like every 3 seconds) the UniSketch TCP Client will send the command "**ping<NL>**" to which the server must respond in some unspecified way, suggested "ack<NL>" for example. If the server does not respond to pings, the client will disconnect and try to reconnect. Note that the TCP client will wait to send out the 'ping' command while there are incoming commands since incoming commands work as confirmations of connection as well.

Periodically (like every 60 seconds) the UniSketch TCP Client will send the command “**list<NL>**” to which the server can respond with state information (like button colors, including graphics, text). It’s not mandatory, more like a provision to compensate for any lost communication that might have resulted in the panel being out of sync with the server - something that ideally should not happen of course since all state information should have been perfectly shared over time.

The client will send “**BSY<NL>**” to the server if it feels it receives content quicker than it can process it. The server should respond by holding back new content until “**RDY<NL>**” or a “ping<NL>” is received from the client. Generally a whole bunch of data (like graphics and text) can be offloaded at any one time without fear of overload or missing packets since transport layers in TCP will take care of queuing, but the BSY / RDY commands are here to make sure the queue doesn’t grow out of hand. If it does, the panel will keep processing the queue and seem to lag behind in processing new commands.

The server is of course responsible to continuously update the client with new state information as necessary in relation to changes on the server.

Server Mode - no handshake!
In server mode the “UniSketch TCP Client” device core does not require any handshaking unless set up by the command HeartBeatTimer.

Raw Panel aka “UniSketch TCP Client”

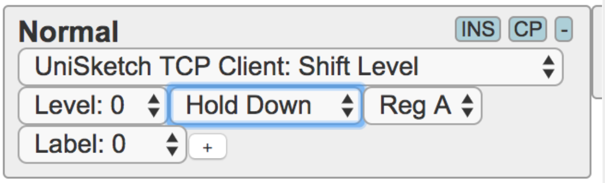
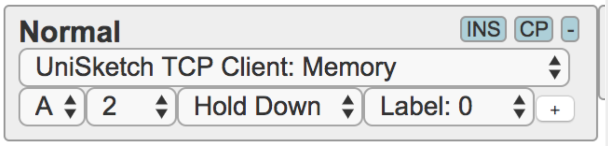
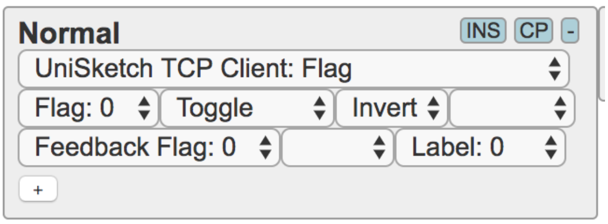
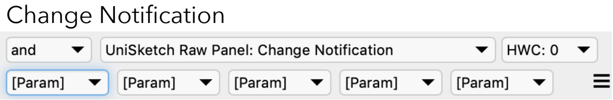
What is today known almost exclusively as Raw Panel was originally born as the UniSketch TCP Client device core. It's purpose was to interconnect UniSketch panels and share various states between them. The device core called "TCP Server" is the opposite end for such a case. The TCP Server device core would create a Raw Panel server that waits for the UniSketch TCP Client to connect. But it would exchange more information than just the triggers and hardware component states. It would share internal memories such as shift levels, states, flags etc. Known applications for UniSketch TCP Client and TCP Server is connecting two ETH-GPI Link boxes over network.
Raw Panel has also historically been referred to as "dumb panels" since the panels don't know anything about the application they are being used in.

Configuration for Raw Panel

A UniSketch panel will only send triggers from hardware components if they are configured with the action "Tie to Remote HWC". If you study configurations for Raw Panel you will see how this action is applied to every single hardware component consistently.
This means the UniSketch panels could also serve two different systems if you add the UniSketch TCP Client device core multiple times and assign actions accordingly. You can also opt to add the "Tie to Remote HWC" to only some hardware components if you want a partial panel to run Raw Panel. The partial application is revealed via the map command to the system in the other end.

Configuration for panel interconnects

When connecting two UniSketch panels or devices, the other actions - Shift level, State, Memory and Flags - are used as well. They are documented in the table below and won't enjoy much more mention here as they are considered legacy that is not found in V2 of the protocol.

Tie to Remote HWC  	Will send down / up / encoder pulses / analog values / speed values to the remote HWC by the number listed (unless zero is selected in which case the current HWC number is used). So for instance, if this is applied to a push button, when that button is pressed down, a Down action for that HWC is sent to the system we are connected to. Likewise, the return value of this element will be the return value retrieved from the remote UniSketch controller. Button colors: Respond to the return value of "HWC#xx=" and "HWCc#xx=" (both are necessary) Displays: Responds to "HWCg#xx" and "HWCt#xx" which lets the server send text and formatting or graphics to the client.
Shift Level  	See description for "Shift Level" from the System Device Core - only this all applies to shift levels on a *remote* UniSketch controller running "TCP Server" device core, not the local.
State  	See description for "State" from the System Device Core - only this all applies to states on a *remote* UniSketch controller running "TCP Server" device core, not the local.
Memory  	See description for "Memory" from the System Device Core - only this all applies to memories on a *remote* UniSketch controller running "TCP Server" device core, not the local. Notice that "Persist" is not implemented either.
Flags  	See description for "Flags" from the System Device Core - only this all applies to flags on a *remote* UniSketch controller running "TCP Server" device core, not the local.
Change Notification  	Inserting this element can lead to automatically sending a message like "HWCmsg#8=CN:0,8,0,0,64" to the server. The message is a change notification which is triggered when any of the five [Param] fields change their value. The values are included in the message as the five comma separated values and can assume any meaning depending on the server side application. On the client side - in the SKAARHOJ Raw Panel - the five parameters can take values like this: - Static value 0-31 - Content of memory A-H - Shift registers (Default, A-D) - State registers (Default, P-S) - Flags (grouped in 8 bits)

	- Time period triggered (1/8s to 64s) The main idea with this element is to create generic notifications to a Raw Panel server which supplies content for displays and/or button colors based on a controller state change such as a camera selection or shift level change which should lead to the external system sending new graphics for displays. The timed element would be a way to make sure a change request is also sent out periodically.

Changelog

September 2017:

- First version

January 2019:

- Pushing an encoder will now send the “Press” action to the server. This was previously done only after holding for 1 second (still does so in any case)
- Added format “9” (XX.XX float) to graphics rendering
- Added format 10 and 11 for graphics rendering
- Added software version output to Raw Panel (UniSketch TCP Client)
- Added commands for handling, changing and reading sleep mode
- Added command (HWCx) for extended return values (like strength, VU meters, setting value of motorised faders).
- Internal changes in State, Shift and sleep mode is reported automatically to host system
- Recommends now to prepend ActivePanel=1 with <NL> to avoid missing initialisation in some cases of disconnect/reconnect

May 2019:

- Added command for receiving panel topology

August 2019:

- Added server mode
- Added "nack" response to unknown commands
- Added "Clear" command
- Added Extended output type 5 useful for faders so they don't need to have their positions updated by the remote system.
- Added "Reboot" command

January 2020:

- Multiple improvements for text rendering in UniSketch has been supported, including:
  - support for 8x8 and 5x5 fonts, separate horizontal and verical text scaling sizes
  - proportional fonts (typically more characters fits in the lines now)
  - Better automatic usage of display tiles regardless of their size (however, it can be overridden by forced values for text sizes, fonts and other things)
  - ASCII range 32-127 supported
  - Increased length of most strings to 24 chars instead of 16/10 etc.
- Support for other image sizes than 64x32, including increased number of image buffers (8)
- Correct centering of images
- Increased number of HWcs from 128 to 255
- Fixed potential bug where if two images was received for the same HWc and buffered at the same time, the first received image would be rendered only (so now we scan the buffered images backwards)
- Added support for 32 bit integers as values in Raw Panel protocol
- Added Raw Panel support for setting icons

May 2020:

- Fixed bug that slowed down sending content in server mode significantly
- Added ClearLEDs and ClearDisplays commands
- Added PanelBrightness and Webserver commands

August 2020:

- Documented quite a bit about topology

April 2022:

- Rearranged and updated manual significantly. Moved to a different location in the repository.

July 2022:

- Blue Pill auto detect ASCII is documented
- More clarification of how systems should interpret topologies was added.
- Tools like PanelTopology and ImageConverter from raw-panel-utils library are described in words and images.
- PanelTopology tool was renamed to Raw Panel Explorer and moved to it's own repo, raw-panel-explorer