

# UniSketch TCP Client (aka "Raw Panel")


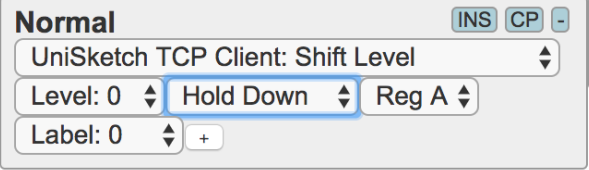
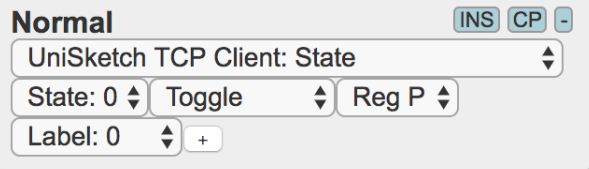
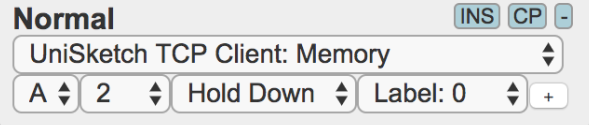
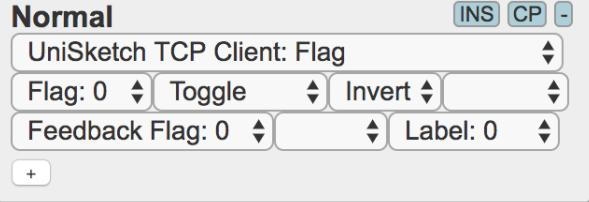
The UniSketch TCP Client device core (also known as "Raw Panel") connects to an external server on port 9923. This server would typically be a third party host system implementing the logic behind the commands sent from the "SKAARHOJ Raw Panel", but it could even be another SKAARHOJ panel with the "TCP Server" device core active in which case the client can "remote control" the server panel. The actions in the table below all relates to such a remote control scenario.

*Raw Panel mode* essentially is to let the server be a software application written to support the UniSketch TCP Client protocol and thus use a SKAARHOJ panel to simply send triggers such as keypresses, pulses and analog values over to the server which in turn maps them to actions in its domain. This has also historically been referred to as "dumb panel" since the panel does not know anything about the application it's being used in. In case of Raw Panel implementations, only the action "Tie to Remote HWC" is likely to be relevant.

## TCP Server Mode

The device core actually does have an option to create a TCP server instead of being a TCP client. This has been implemented due to popular request. However, just keep in mind that most of this manual is written as if the SKAARHOJ panel is always the TCP client connecting to a server, but its actually possible to inverse this. Read more about it further down in the documentation.

This is a table of UniSketch actions for UniSketch TCP Client:

|   |  |
|---|--|
| <p><b>Tie to Remote HWC</b></p>  | <p>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 TCP Server we are connected to. Likewise, the return value of this element will be the return value retrieved from the remote UniSketch controller.</p> <p>Button colors: Responds to the return value of "HWC#xx="</p> <p>Displays: Responds to "HWCg#xx" and "HWCt#xx" which lets the server send text and formatting or graphics to the client.</p> |
| <p><b>Shift Level</b></p>      | <p>See description for "Shift Level" from the System Device Core - only this all applies to shift levels on a remote UniSketch controller, not the local.</p>  |
| <p><b>State</b></p>            | <p>See description for "Shift Level" from the System Device Core - only this all applies to shift levels on a remote UniSketch controller, not the local.</p>  |
| <p><b>Memory</b></p>           | <p>See description for "Memory" from the System Device Core - only this all applies to memories on a remote UniSketch controller, not the local. Notice that "Persist" is not implemented either.</p>  |
| <p><b>Flags</b></p>            | <p>See description for "Flags" from the System Device Core - only this all applies to flags on a remote UniSketch controller, not the local.</p>   |



## API for “Raw Panels”

UniSketch TCP Client can be used to set up an essentially “dumb panel” that only sends action triggers such as keypresses to the server and receives color values for a button and text or graphics for displays. This method is used when UniSketch TCP Client connects to a TCP Server on another SKAARHOJ controller. Likewise any other piece of broadcast hardware can implement a TCP Server that simply uses the same API to exchange information. The function of a “dumb panel” is implemented by using the “Tie to Remote HWC” action on any hardware interface component that is intended to work as such. Thus a “dumb panel” is only dumb to the extent that hardware interface components are consistently mapped to this action - in other words, a configuration mixed with other device cores or system actions is perfectly possible although that introduces more autonomy and “intelligence” in the panel itself.

### Notes Clients and Servers

In the following the term “client” is used for the SKAARHOJ panel with the UniSketch TCP Client device core running and the term “server” is used to indicate the broadcast device or software to which the client connects. However, as it has already been noted and will be documented further down, it's possible to reverse this so the panel hosts a TCP server and any external system can connect to it as a TCP client instead. If that is done, the in- and outbound commands between the panel and external system will of course still be the exact same, it's only the method of connection that changes.

## TCP settings

A SKAARHOJ 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**. (See further down for how to change this port number.)

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

## Handshaking

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.

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!

Notice: In server mode the “UniSketch TCP Client” device core does not require any handshaking. This will be up to the connecting client to implement as it sees fit.

## Inbound TCP commands - from external system to SKAARHOJ panel

In general, see the documentation for the “TCP Server” device core which lists the basic command set supported. However, for the application of “Dumb panels” we will not use any of the registers

(shift, state, flags, mem etc.) and focus only on exchange of information in relation to hardware interface components (HWCs).

The basic incoming commands to the panel that the external system could send are listed in this table:

| Command           | Description   |
|-------------------|---|
| <b>HWC#xx=yy</b>  | <p>Status On/Off/Dimmed<br/>xx is the HWC number, yy is a byte defining the state of the component.</p> <p>The state, "yy", often translates into a color such as off / dimmed / on, but may also contain simple on/off binary information:<br/>Bit 0-3 forms a number from 0-15:</p> <ul style="list-style-type: none"> <li>• 0 = Off</li> <li>• 2,3,4 = On (where 2=red, 3=green, 4="On" color (white or yellow by default))</li> <li>• 5 = dimmed "On" color.</li> </ul> <p>Bit 4: Blink flag for monocolour buttons. If set, a monocolour 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".<br/>Bit 5: Output bit (32); If this is set, a binary output will be set if coupled with this hwc. Generally: Let bit 5 follow whether the "On" color (2,3 or 4) is commanded and let it be off if 0 or 5 is commanded.<br/>Bit8-11: Blink bits: If set 0001, the button will blink with a frequency of about 4 Hz, If set to 1000, 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 millisecond clock and other combinations can create other blinking patterns.<br/>Most typically you would send these values back: 0 ("Off"), 36 (32+4 for "On") and 5 (for "Dimmed")</p>                   |
| <b>HWCx#xx=yy</b> | <p>Extended return values<br/>xx is the HWC number, yy is a 16 bit word defining the extended output of the component.</p> <p>The rightmost 10 bits of this word is the value.<br/>Bits 11 and 12 are reserved for the individual output types to define.<br/>The leftmost 4 bits of this word is the output type:<br/>0=none<br/>1=Output Strength: Value from 0-1000, used to set a strength indication on and LED bar or position of a motorized fader.<br/>2=Directional Output Strength: <i>[future, todo]</i></p> <p>3=Shows steps 0 (no LEDs, off), 1=first, 2=second, 3=third etc. If beyond the number of LEDs, the full bar will light up dimmed.<br/>4=VU metering for audio (values 0-1000)<br/>5=Fader move to position. Like output type 1, but a one time setting that clears itself afterwards. This is useful for faders in many cases: When you receive inputs from a motorized fader, you should either 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 using output type 1. Alternatively, if you use output type 5, the faders previously set position from the external system was a one-off event and the panel is not trying to maintain this position - and therefore won't care if you move the slider to somewhere else.</p> |
| <b>HWCc#xx=yy</b> | <p>Externally imposed button color: index or rrggbb<br/>xx is the HWC number, yy is a byte defining the color of the component if it is supposed to be set externally and not reflect the panel default</p> <p>Bit 7: If set, the color of the component is defined by this value, otherwise the panel default will be used (it's an enable-bit)<br/>Bit 6: Defines the interpretation of bits 5-0; If set, bits 5-0 represents the component color with "rrggb". 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):</p> <ul style="list-style-type: none"> <li>• 0: DEFAULT_COLOR, // Default (+bit 7 on = 128)</li> <li>• 1: 0, // Off (+bit 7 on = 129)</li> </ul>  |

| Command               | Description  |
|-----------------------|--|
|                       | <ul style="list-style-type: none"> <li>• 2: 0b111111, // White (+bit 7 on = 130)</li> <li>• 3: 0b111101, // Warm White (+bit 7 on = 131)</li> <li>• 4: 0b110000, // Red (Bicolor) (+bit 7 on = 132)</li> <li>• 5: 0b110101, // Rose (+bit 7 on = 133)</li> <li>• 6: 0b110011, // Pink (+bit 7 on = 134)</li> <li>• 7: 0b010011, // Purple (+bit 7 on = 135)</li> <li>• 8: 0b110100, // Amber (Bicolor) (+bit 7 on = 136)</li> <li>• 9: 0b111100, // Yellow (Bicolor) (+bit 7 on = 137)</li> <li>• 10: 0b000011, // Dark blue (+bit 7 on = 138)</li> <li>• 11: 0b000111, // Blue (+bit 7 on = 139)</li> <li>• 12: 0b011011, // Ice (+bit 7 on = 140)</li> <li>• 13: 0b001111, // Cyan (+bit 7 on = 141)</li> <li>• 14: 0b011100, // Spring (Bicolor) (+bit 7 on = 142)</li> <li>• 15: 0b001100, // Green (Bicolor) (+bit 7 on = 143)</li> <li>• 16: 0b001101, // Mint (+bit 7 on = 144)</li> </ul> <p>The colors marked “(Bicolor)” are the only ones recommended for use with red/green bicolor buttons on panels.</p>   |
| <b>HWCT#xx=string</b> | <p>Display text, tokenized string</p> <p>xx is the HWc number, <i>string</i> is a string tokenized by a vertical pipe character, “ ”, where each position represents a given parameter being either an integer, boolean or string.</p> <p>The format of <i>string</i> follows this:</p> <p>[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]</p> <p><i>string</i> may not be longer than 63 chars</p> <ul style="list-style-type: none"> <li>• [value] is a 32 bit integer representing the numerical value to be shown. If empty, it will not render at all (like format=7).</li> <li>• [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 Textline (Title &amp; value=size 1-4), 11=2 Textlines (Label 1, Label 2 &amp; value=size 1-2). Default if empty is Integer.</li> <li>• [fine] is used to set various icons</li> <li>• [Title] defines the title string shown in the top of the display. Up to 10 chars long.</li> <li>• [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.</li> <li>• [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.</li> <li>• [label 2] Second text line under title. If not empty, both [label 1] and [label 2] will print in small letters.</li> <li>• [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]</li> <li>• [values pair] ranges from 1-4 and indicates 4 variations of boxing of value pairs.</li> <li>• [scale][scale range low][scale range high][scale limit low][scale limit high] in-</li> </ul> |

| Command                  | Description   |
|--------------------------|---|
|                          | <p>icates different types of scales in the bottom of the graphic that can show a range of a given value.</p> <ul style="list-style-type: none"> <li>• [img] is an index to a system stored media graphic file.</li> <li>• [font] is font face (0-2)</li> <li>• [font size] is font sizes horizontal and vertical for both content and title</li> <li>• [advanced] is some other settings</li> </ul> <p>Please check out the section later in this document for examples and a table with a better overview.</p> <p>Caching:<br/>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 a string should be cached, simply prefix it with "{CS:xxxxx}" where xxxxx is a 15 bit decimal unique non-zero identification number of your choice. To later set the same content again, but using the cached content, simply send a string with "{CR:xxxxx}" and nothing more.<br/>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.</p> <p>Caching has to be enabled and there is a limited number of slots available.... (more to come)</p>   |
| <b>HWCg#xx=yy:string</b> | <p>Display graphic, monochrome<br/>xx is the HWC number, yy is an index 0-2 and <i>string</i> is 1/3 of the image data encoded in base64 (true for 64x32 images, legacy format)</p> <p>Please check out the section later for more information on sending image data to Raw Panel.</p> <p><i>Description of the 64x32 legacy format:</i><br/>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, totalling 256 bytes. The 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.<br/>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).<br/>SKAARHOJ has a helpful tool to convert images to 64x32 monochrome images:<br/><a href="http://skaarhoj.com/FreeStuff/GraphicDisplayImageConverter.php">http://skaarhoj.com/FreeStuff/GraphicDisplayImageConverter.php</a></p> <p>Caching:<br/>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.<br/>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.</p> <p><u><a href="#">Caching has to be enabled and there is a limited number of slots available.... (more to come)</a></u></p> |
| Clear                    | Clears all values sent by HWC, HWCx, HWCc   |
| Reboot                   | Reboots the panel (returns text "Rebooting...\n")   |
| ActivePanel=1            | <p>Activates panel<br/>Send this to activate the panel when "list" is received from the panel.<br/>It's recommended to prepend ActivePanel with &lt;NL&gt; in order to make</p>   |

| Command                     | Description  |
|-----------------------------|--|
|                             | 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                        | Asks panel to reveal some information about itself<br>Returns <code>_serial</code> and <code>_model</code> and <code>_version</code>   |
| ping                        | Keep-alive, return "ack<NL>"   |
| map                         | Produces the map again (which is also sent when initially connecting as a client)  |
| PanelTopology?              | Asks panel to send SVG and JSON data for topology  |
| Mem, Flag#, Shift, State... | 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               | Sets the global sleep timer in milliseconds: This is the number of milliseconds that shall pass before the panel will enter sleep. If zero, sleep is disabled.                   |
| SleepTimer?                 | Will request the global sleep timer value from the panel.  |
| WakeUp!                     | Will wake up the panel if it was asleep.   |
| encoderPressMode=xx         | In xx:<br>bit 0: If set, encoders will return "Press" on "act down" (as well as press after holding down for 1 second). Default is 1.  |
| (Unknown command)           | returns "nack<NL>"   |

## Outbound TCP commands - from panel to external system

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

| Command                     | Description  |
|-----------------------------|--|
| <b>HWC#xx[.mask]=string</b> | <p>Trigger action from hardware component<br/>xx is the HWC number, <i>string</i> contains information about the trigger.</p> <p>Fourway buttons will also add the <i>mask</i>, which is a period followed by a number 1,2,4, or 8 indicating which edge was pressed on the button, respectively Up, Left, Down, and Right.</p> <p><i>string</i> can have any of these forms:</p> <ul style="list-style-type: none"> <li>• "Down" : the component (typically a button or a GPI trigger or encoder knob) is pressed down (or held down for one second with encoders)</li> <li>• "Up" : the component is released again</li> <li>• "Press" : represents that Down and Up happened essentially simultaneously - a pulse</li> <li>• "Abs:yy" : A change, yy, to an absolute position (for example a T-bar). yy ranges 0 to 1000</li> <li>• "Speed:yy" : A change, yy, to a speed (for example a spring loaded joystick). yy ranges -500 to 500</li> <li>• "Enc:yy" : Pulses, yy, from an encoder. The sign indicates direction.</li> </ul> |



| Command                | Description   |
|------------------------|---|
| map=zz:xx              | <b>Local HWc to External HWc mapping information</b><br>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.<br><br>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. |
| BSY                    | Busy message  |
| RDY                    | Ready message   |
| list                   | Initialization status request, return "<NL>ActivePanel=1<NL>"   |
| ack                    | Acknowledgement to "ping" command   |
| nack                   | Unknown incoming command  |
| _model                 | _model=[Model / Product Key]  |
| _serial                | _serial=[Serial number]   |
| _version               | Software version from panel.  |
| _panelTopology_svgbase | Panel base SVG  |
| _panelTopology_HWC     | JSON with HWC (HardWare Component) data   |
| _state:[reg]=xx        | Informs about the panels state register value (sent when changed)   |
| _shift:[reg]=xx        | Informs about the panels shift register value (sent when changed)   |
| _isSleeping=[0/1]      | Informs about the panels whether the panel sleeps or not (sent when changed)  |
| _sleepTimer=xx         | 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.  |

## Server Mode

Using a device core option (see further down, compile time setting) you can enable the device core as a TCP server instead. This turns out to be more convenient for some external systems, sometimes due to redundancy considerations. It also turns out to be fairly nice to test, since the server mode implements no handshake requirements from the panel, so you can basically telnet to the panel and have your fun.

You may want to make sure your telnet application sends \n at the end of a string. "telnet" seems not to do this, but netcat is more compliant:

```
nc 192.168.10.99 9923
```

This would connect to a SKAARHOJ panel with UniSketch TCP Client in server mode on IP 192.168.10.99 and the default port (9923).

After connecting, try to write "ping" + return and you should see "ack" coming back at you. From that point you can try working through the various commands listed above in the section about In-bound commands.

## Get started with test servers written in Python 3

We have written a few Python 3 scripts that will help you to get started quickly implementing support for SKAARHOJ panels in your software application. They can be downloaded from GitHub: <https://github.com/SKAARHOJ/Support/tree/master/Files/UniSketchTCPClient>

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 and trying to connect to the IP address of the host computer will interact with the scripts.

They are a great resource to learn from and experiment with to get up to speed with integrating SKAARHOJ panels with your broadcast software or hardware solution.



We have put videos on YouTube as well that demonstrates these scripts with panels.  
See <https://www.skaarhoj.com/support/raw-panel/>

## Text based graphics

The displays on SKAARHOJ controllers are generally 64x32 pixel graphical displays - "tiles". Sometimes many of them are pooled together on a single, larger display, other times they are individual LCDs on a SmartSwitch. 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 "HWCtrl#xx=string" as documented above. 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  | -9999   | 32767 1  Float2  | 299 2  Percent   | 999 3  dB  | 1234 4  Frames  |
| 16 bit integer   | 16 bit integer, negative                                | Float with 2 decimal points  | Integer value in Percent   | Integer value in dB  | Integer in frames                                       |
|  |   |  |  |  |   |
| 999 5  Reciprocal  | 9999 6  Kelvin  | 9999 7  [Empty!]   | -3276 8  Float3  | 1 [Fine] 1   | 1 Title String  |
| Reciprocal value of integer  | Integer formatted as Kelvin                             | format 7 = empty!  | Float with 3 decimal points, optimized for 5 char wide space. Op to +/-9999            | Fine marker set (the curvy thing on the right of the line), title as "label" | no value, just title string (and with "fine" indicator) |
|  |   |  |  |  |   |
| Title String 1   | Title string 1 Text1Label                               | Title string 1 Text1Label  0   | Title string 1 Text1Label Text2Label   | Title string 1 Text2Label  | Text1Label  |
| Title string as label (no "bar" in title)  | Text1label - 5 chars in big font                        | Adding the zero (value 2) means we will print two lines and the text label will be in smaller printing | Printing two labels of 10 chars - automatically the size is reduced                    | Printing only the second line - automatically the size is reduced            | Text1label - 5 chars in big font, no title bar.         |
|  |   |  |  |  |   |
| Text1Label  0  | Text1Label Text2Label                                   | Text2Label   | 123  Title string 1 Val1: Val2: 456  | -1234 1 Coords: x: y: 4567 2   | -1234 1 Coords: x: y: 4567 3                            |
| Adding the zero (value 2) means we will print two lines and the text label will be in smaller printing | Printing two labels - automatically the size is reduced | Printing only the second line - automatically the size is reduced                                      | First and second value is printed in small characters with prefix labels Val1 and Val2 | A box around the first label/value line                                      | A box around the second label/value line                |
|  |   |  |  |  |   |
| -1234 1 Coords: x: y: 4567 4   | -500 1 Coords:     1 -1000 1000 -700 700 1              | -500 1 Coords:     2 -1000 1000 -700 700 2   |  |  |   |
| A box around the both label/value lines  | A solid bar scale added below value                     | A moving dot scale added below value   |  |  |   |

These graphics are generated from the test Python scripts. They can be very useful to experiment with other combinations. A good script to use for testing would be the script "TCPserver\_colorAnd-

DisplayTestByButtonPress.py" as it has a large number of text and image combinations to learn from.

For format for setting text in the displays is based on tokenizing 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.

| In<br>de<br>x | internal<br>name                          | Name                                 | Description   |
|---------------|---|--------------------------------------|---|
| 0             | _extRetVal[0]                             | Value, 32 bit integer                | Integer value to show in the display. Subject to formatting options in index 1.<br>If empty string, the display format will be set to 7 (value not printed)   |
| 1             | _extRetFormat<br>(bit 0-3)                | Formatting type                      | Determines how the integer value from index 0 (as well as index 7) is formatted in the display:<br><br>0 = as a signed integer<br>1 = float from 10^3 (X.XX). Deprecated, use format 9<br>2 = XX%<br>3 = XXdb<br>4 = XXf<br>5 = 1/XX<br>6 = XXK<br>7 = Blank (not printed)<br>8 = float from 10^3 (X.XXX)<br>9 = float from 10^2 (XX.XX)<br>10 = one text line (index 0 will be your forsize, 1-4) rendered from index 3 (title).<br>11 = two text lines (index 0 will be your forsize, 1-2) rendered from index 3 (title).<br>12 = float from 10^1 (XXX.X) |
| 2             | _extRetFormat<br>(bit 4-5)                | Icon                                 | <b>Bit 0-1 value (0-3):</b><br>0 = No icon<br>1 = Fine-flag (speedy wave lines under title bar, right)<br>2 = Lock icon (in title bar, right)<br>3 = No Access icon (lower right)<br><br><b>Bit 3-5: (Corner icons below title bar in right side, 8x8 pixels) value (0-7)</b><br>(0 : No icon)<br>1 (8) : Cycle icon (return arrow)<br>2 (16): Down (down arrow)<br>3 (24): Up (Up arrow)<br>4 (32): Hold (Down arrow pointing to line)<br>5 (40): Toggle (zig-zag)<br>6 (48): OK (check mark)<br>7 (56): Question mark                                     |
| 3             | _extRetShort<br>_extRetLong<br>(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)<br>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<br>(enables it also)      | Second line of text, string 24 chars |   |
| 7             | _extRetVal[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<br>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<br>2 = The upper label/value pair is marked<br>3 = The lower label/value pair is marked<br>4 = Both label/value pairs are marked  |
| 9             | _extRetScale-Type                         | Scale type                           | 1 = strength bar (from left)<br>2 = centered marker<br>3 = centered bar (from center of range)  |

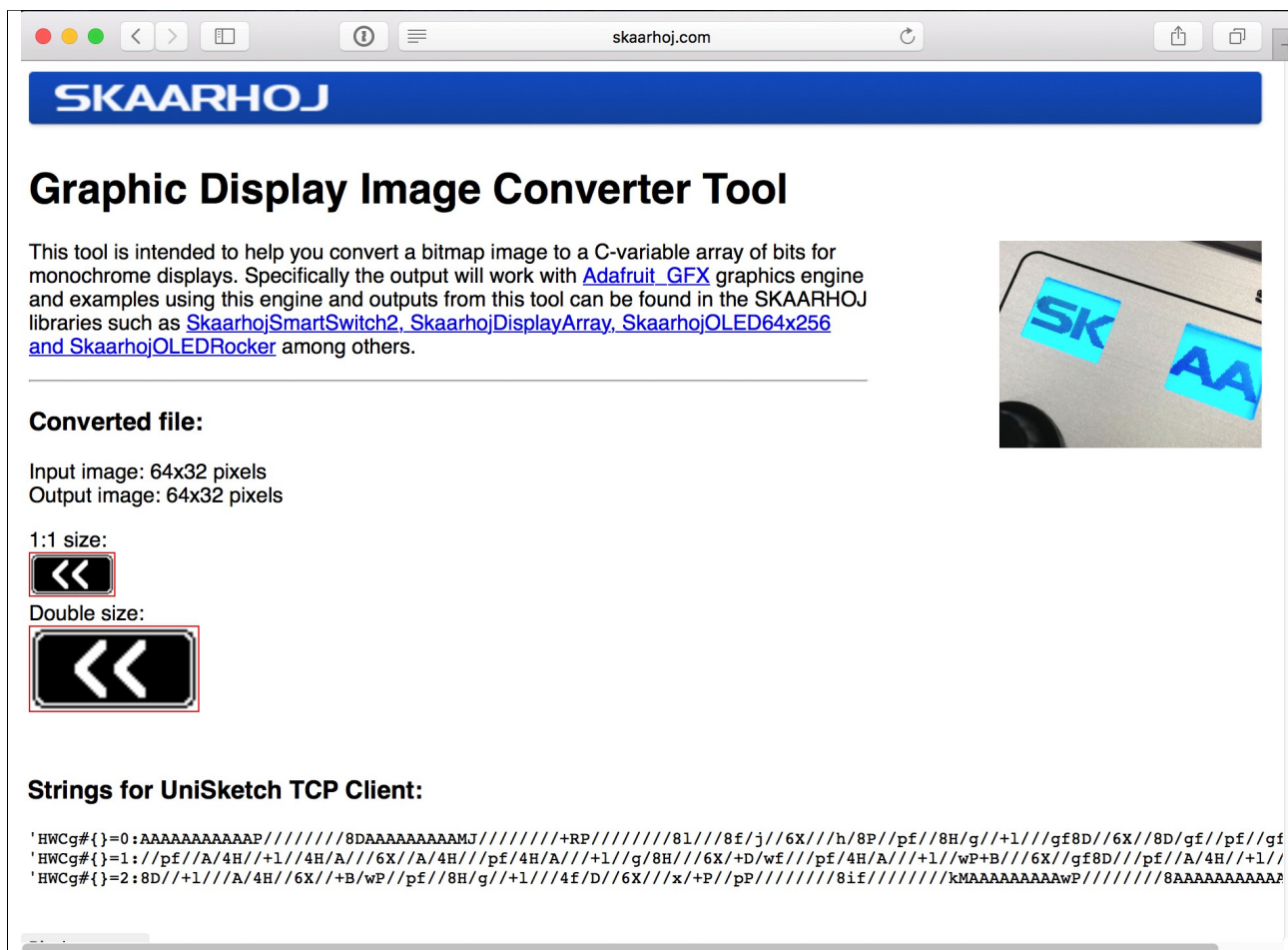
|    |   |                                      |  |
|----|---|--------------------------------------|--|
| 10 | <code>_extRetRangeLow</code> (integer)  | Range low                            | Low range value  |
| 11 | <code>_extRetRangeHigh</code> (integer) | Range high                           | High range value   |
| 12 | <code>_extRetLimitLow</code> (integer)  | Limit low                            | Limit low marker (set to same as range low if you don't want it. Must be set!)                     |
| 13 | <code>_extRetLimitHigh</code> (integer) | Limit high                           | Limit high marker (set to same as range high if you don't want it. Must be set!)                   |
| 14 | <code>extRetVallImage</code> , integer  | Image reference, zero is first image | Reference to an internally stored image (compiled into the firmware from cores.skaarhoj.com)       |
| 15 | <code>_extRetAdvancedFontFace</code>    |                                      | Bit 0-2: General font face, Bit 3-5: Title font face, Bit 6: 1=Fixed Width                         |
| 16 | <code>_extRetAdvancedFontSizes</code>   |                                      | 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 | <code>_extRetAdvancedSettings</code>    |                                      | Bit 0-1: Title bar padding, Bit 2-4: Extra Character spacing (pixels)                              |

## Pixel 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](https://github.com/SKAARHOJ/Support/tree/master/64x32_Graphics)

To assist you in converting such graphics to the code you can send, we have provided a graphic conversion tool online which also outputs the 3 lines of consecutive commands, "HWCg#xx=", that you need to send them to the panel:



You will recognize the three lines in this screenshot as being the commands needed to send a graphic to the client panel. The “{}” is substituted with the HWC number. There are ample examples of this in the Python 3 test scripts.

Find the graphical conversion tool here: <http://skaarhoj.com/FreeStuff/GraphicDisplayImageConverter.php>

## Format

To facilitate images in varying sizes, a new format for base64 encoding the images has been introduced. Instead of three lines of fixed lengths, you can encode the image over an arbitrary amount of lines, although they have a practical limit of around 250 bytes length (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],[x-dimension]x[y-dimension]:[base64 encoded data]`

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/BsPg004Cw==',
'HWCg#{8:+P4zD4fB8P/AxncHAG04C8DGfw==',
'HWCg#{9:hWBjGAvAxcPgMYC8TuAx3P4w==',
'HWCg#{10:uQv8fAMYz8Hx/8QAAAAAAAAELxA==',
'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.

## 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.

## 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 new 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   |
| 96x48  | Color OLED 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 (hShrink and wShrink). 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.

## Device Configurations

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".

## Changelog

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 vertical 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