ScanaStudio Scripting programmer's manual

Script API version: 3.0

Ibrahim KAMAL

November 7, 2019

Contents

Introduction to ScanaStudio scripts

ScanaStudio scripts are JavaScript (*.js) files that are executed by ScanaStudio. Those scripts have access to the signals captured by logic analyzer devices, and can output data in various ways depending on the task the script has to accomplish. There are various kind of script, the most common type is a protocol decoder script, that is, a script that will decode (interpret) logic signals according to a standard protocol (like I2C, Serial UART, CAN, Etc…). There are also scripts that will allow a logic analyzer device to trigger on a specific word of a protocol, and finally, there are scripts that can build some logic patterns (like a PWM pattern) that can be later generated by devices that support signal generation.

Note: This document is oriented to users with some minimal skills, you are one of those users if:

- You have programmed in any language before, and preferably have some JavaScript background
- You have some minimal experience with logic signals and digital electronics
- You are familiar with one or more serial protocols (like I2C, UART or CAN bus, to name only a few examples)

If you feel you're lacking some of the skills mentioned above, it's probably not a good idea to start writing scripts until you gain more knowledge about digital electronics and programming.

A bit of history

In 2019, Ikalogic has introduced a new version of ScanaStudio (Logic Analyzer software). With this new version, the scripting system was totally rebuilt from the ground up to enhance its operation and broaden the scope of the scripts. It's also worth noting that the previous script system was initially designed in 2015 only to decode signals. Over the years, features and options were gradually added to the scripting system, and it came to a point where it was totally non-harmonized and barely usable for someone outside Ikalogic organization. This new version of ScanaStudio script API addresses this problem. It's designed to be intuitive, future-evolutions ready, easy to appropriate and fast to execute.

Scope of this document

ScanaStudio scripting language is based on JavaScript. This document will not cover the JavaScript basics that someone need to have to write or modify ScanaStudio scripts. Excellent tutorials exist on the internet for that purpose. This document will however cover all the details related to the specifics of writing scripts for ScanaStudio. Example source codes will be provided as often as necessary, so if you're not a JavaScript expert but have general programming knowledge, you should be able to follow fairly easily.

Important note: This documentation covers ScanaStudio V3.1 and beyond. Earlier versions are not covered by this documentation as it is considered too different to be merged in a single document.

Older scripts repository can be found here.

What can a ScanaStudio script do?

We're constantly opening up new parts of ScanaStudio to be "scriptable". At the moment this documentation is written, the following main features are implemented:

- Decode logic signals according to a specific protocol (this include building packet-view and hex-view).
- Build logic patterns to be generated with devices that have this capability.
- Build demo signals for a particular protocol (arbitrary signals that are generated when no device is connected to the software)
- Generate trigger sequence (e.g. trigger on a specific I2C address).

More features are planned, so check back later and/or don't hesitate to share the list of features you think script should be able to handle.

Principle of operation

As stated before, ScanaStudio scripts are based on standard JavaScript. There are two ways ScanaStudio and user script need to interact with each others:

Interaction initiated by ScanaStudio ScanaStudio needs to call one or more special functions (called *entry-point functions* hereafter) to start an operation. For example, if a script needs to be used to decode signals it must at least implement a function called `on_decode_signals(resume)`, something like this:

```
function on_decode_signals(resume)
{
   /*
   Write here the code that will interpret the logic signals
   resume is == true if decoder should resume where it left
   */
}
```

This function will be called by ScanaStudio when logic signals need to be decoded. From a programming point of view, entry point functions can be treated as events handlers.

Interaction initiated by the script The script needs to retrieve information from ScanaStudio, as well as call various functions in ScanaStudio software. An object called `ScanaStudio` is exposed in the script context as a global object, and can be used for all the interactions with ScanaStudio that are initiated by the script. For example, a protocol decoder script can get the sampling rate for the last captured samples via this function call:

`ScanaStudio.get_capture_sample_rate()`

In the same manner, the script can output some console messages via this function:

`ScanaStudio.console_info_msg("Hello world from the script!").`

Below is a more complete example:

```
function on_decode_signals(resume)
{
   ScanaStudio.console_info_msg(
                  "The sample rate is:" +
                  ScanaStudio.get_capture_sample_rate() + " Hz");
}
```

Obviously, the example above is completely useless in terms of "signal decoding", it's just here to illustrate the way the standard JavaScript is enhanced with the `ScanaStudio` global object.

Setting up coding environment

This section presents the coding and debugging environment as used by Ikalogic engineers. Those are only recommendations. You're free to use our tools and methodology and/or inspire from this setup as needed.

To edit the script (or write new ones) we recommend using Atom text editor. It's a free and open source editor that is perfectly adapted to coding and scripting. Also, Ikalogic provides an Atom plugin that adds *text snippets*. After downloading and installing Atom to your computer, you can install ScanaStudio plugin by following those steps:

1. In Atom, go to File > Settings
2. Select "+Install" from the left vertical menu
3. Search for a package names "ScanaStudio-snippets" and install it.
4. Close and restart Atom.

Once the ScanaStudio-snippets package is installed, you will see snippet widgets appear as you type to help you finding the right syntax.

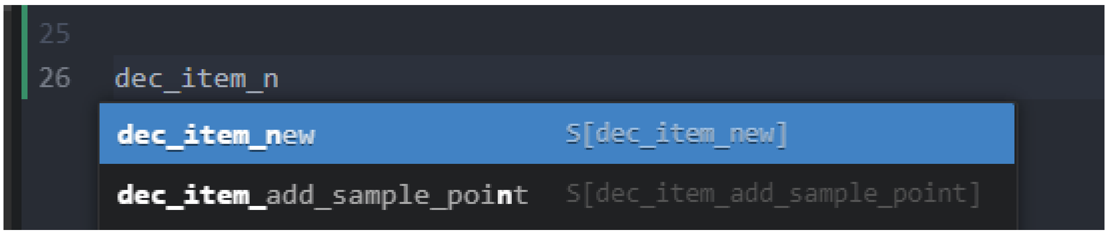

dec_item_n
dec_item_new S[dec_item_new]
dec_item_add_sample_point S[dec_item_add_sample_point]

Figure 1: atom-snippet-example

Some snippets can generate whole blocs of code - a template - like the `meta-info-template` snippet:

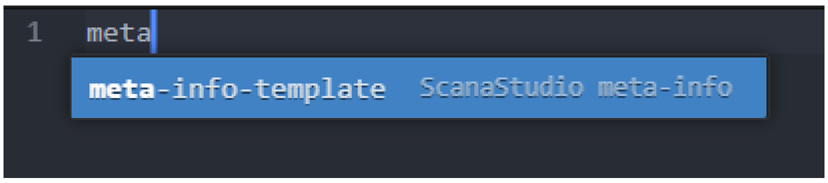


Figure 2: meta-info-snipped

Which will generate this code block:

```
/* Protocol meta info:
<PROTOCOL_NAME> My Protocol </PROTOCOL_NAME>
<DESCRIPTION>
My protocol can decode the s%^* out of any logic signal!
</DESCRIPTION>
<VERSION> 0.0 </VERSION>
<AUTHOR_NAME>   Your name </AUTHOR_NAME>
<AUTHOR_URL> your@email.or.website </AUTHOR_URL>
<COPYRIGHT> Copyright 2018 your name </COPYRIGHT>
<LICENSE>   This code is distributed under the terms of the GNU General
    Public License GPLv3 </LICENSE>
<RELEASE_NOTES>
V0.0:  Initial release.
</RELEASE_NOTES>
*/
```

Hint: Start typing the word “template” and Atom will show a list of all available template snippets that can help you easily get started with the writing of a script.

Architecture of a script

This chapter will give you a more exhaustive view of the base construction blocks of a script. It will address the different kind of interactions that can take place between your script and ScanaStudio software.

A script is composed of three main parts:

- Meta-information
- One or more entry point functions
- User code, that is, all the rest of your script.

Any script must contain some meta-information, which is some text that describes what your script does, what's its version or who is the author. Think of the Meta-information as the ID card of your script.

A script must also contain at least one entry point function. Entry point functions are called by ScanaStudio when it's time for your script to execute the task it was made for.

Both meta-information and entry point functions will be discussed in details in the following sections.

Meta-information

As stated above, meta-information is like an ID card of your script. It can be placed anywhere in the script, but for the sake of harmonization, we recommend that you place it at the very top of your script. Meta-information is composed of a commented bloc of code, containing meta tags. Each tag encapsulates a specific information as shown in the example below:

```
/*
<NAME> My script name </NAME>
<DESCRIPTION>
Description of my script. It can decode my custom protocol,
and it can generate trigger sequences for that specific protocol.
</DESCRIPTION>
<VERSION> 1.1.2 </VERSION>
<AUTHOR_NAME>   My name </AUTHOR_NAME>
<AUTHOR_URL> my_website_or_email.com </AUTHOR_URL>
<HELP_URL> link to a documentation for your script, e.g.:
https://github.com/ikalogic/ScanaStudio-scripts-v3/wiki/SPI-script-
    documentation </HELP_URL>
<COPYRIGHT> Copyright 2018 Myself or my company </COPYRIGHT>
<LICENSE>   This code is distributed under the terms of the
GNU General Public License GPLv3 </LICENSE>
```

```
<RELEASE_NOTES>
V1.1.2:  Fixed anther bug.
V1.1.1:  Fixed some bug.
V1.1.0:  Added some feature.
V1.0.0:  Initial release.
</RELEASE_NOTES>
*/
```

As you may have noticed, this is inspired from the way HTML or XML tags work. It's highly recommended that each script implements this full list of meta-info tags. This will allow ScanaStudio to provide consistent and harmonized information to the user of your script.

Entry points functions

Each and every script (that is to be supported by ScanaStudio) need entry point functions to operate. These are called by ScanaStudio when a specific function is to be carried by your script.

A summary of the entry point functions is listed in the following table:

Function	Script Context	Description
on_decode_signals()	Protocol decoder	Called when ScanaStudio needs to decode captured signals.
on_draw_gui_decoder()	Protocol decoder	Called when ScanaStudio needs to show protocol decoder configuration GUI.
on_eval_gui_decoder()	Protocol decoder	Called when ScanaStudio needs to evaluate if the decoder GUI configuration is valid
on_build_trigger()	Trigger	Called when ScanaStudio need to generate a trigger sequence.
on_draw_gui_trigger()	Trigger	Called when ScanaStudio needs to show the trigger GUI
on_eval_gui_trigger()	Trigger	Called when ScanaStudio needs to evaluate if the Trigger GUI configuration is valid.
on_build_signals()	Signal builder	Called when the user requests signals to be built.
on_build_demo_signals()	Signal builder	Called when the user tries to run a workspace that has your script without any device connected.
on_draw_gui_signal_builder()	Signal builder	Called when ScanaStudio needs to show the signal builder GUI

Function	Script Context	Description
on_eval_gui_signal_builder()	Signal builder	Called when ScanaStudio needs to evaluate if the signal builder GUI configuration is valid.

Note: Each and every one of those functions will be discussed in details in dedicated sections of this document.

Not all of those entry point functions need to be implemented in your script. It all depends on what features you want your script to have. ScanaStudio will parse your script searching for those "special functions", and will automatically detect which features are supported by your script.

Obviously, those entry point function's names are reserved only for this purpose. You should never use the names of those functions as regular user function. Also, you should never call entry point functions from within the script as this can lead to unexpected behavior.

You may notice the word "**Script context**" was mentioned in the table above. The Script context simply defines what your script is allowed to do when a particular entry point function is called. In other words, some of the functions exposed by the `ScanaStudio` object are only available for a particular context.

For instance, if the `on_decode_signals` function is called - in the Protocol decoder context - the `ScanaStudio` object will expose functions that allows fetching captured signals and creating decoded data elements (that appears on the waveform). Example of a function available in this context:

```
ScanaStudio.trs_reset(0); // reset the transitions iterator for channel
     0
```

Similarly, in the protocol decoder context, functions related to building signal *cannot* be called. For example, the following code is invalid if called from protocol decoder context:

```
ScanaStudio.builder_add_samples(0,1,200); //Add 200 HIGH LEVEL samples
    on channel 0
```

However, some functions in the `ScanaStudio` objects are available across all contexts (global context), like the functions used to display console messages:

```
ScanaStudio.console_info_msg("Hello World"); //Valid in all contexts
```

As you may have noticed, the entry point function called to launch your script defines the context during the whole time the script is executed. Context can never be changed unless another entry function is called.

Usually, you don't need to worry about entry points and their contexts: you should never need to use signal building functions when decoding signals, and vice-versa. If at some point you feel limited by the context of the script, that probably means that you're doing something wrong and that it's time to rethink the architecture of your script.

Along this document, all function available through `ScanaStudio` object are described and its context clearly stated.

Anatomy of a simple script

Before we get deeper into the exhaustive list of functions and methods used to build sophisticated scripts, let's analyze a very simple script to get a global view of the architecture of a script and how different parts, work together. Let's imagine we want to build a script that will calculate the frequency of signals, and display the frequency on the waveform as shown below.

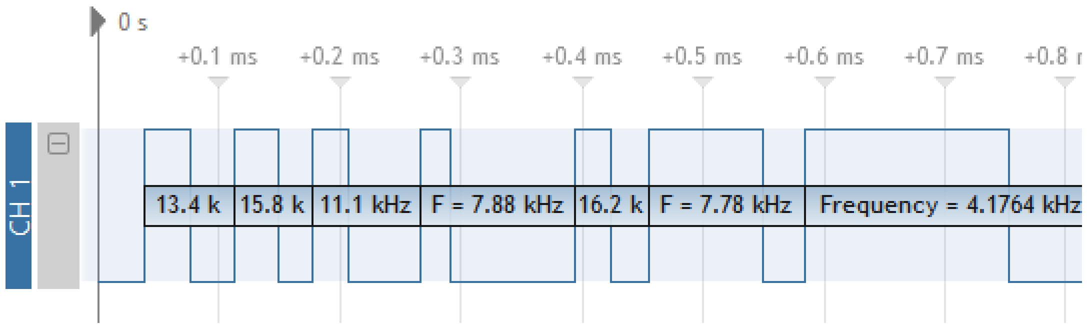


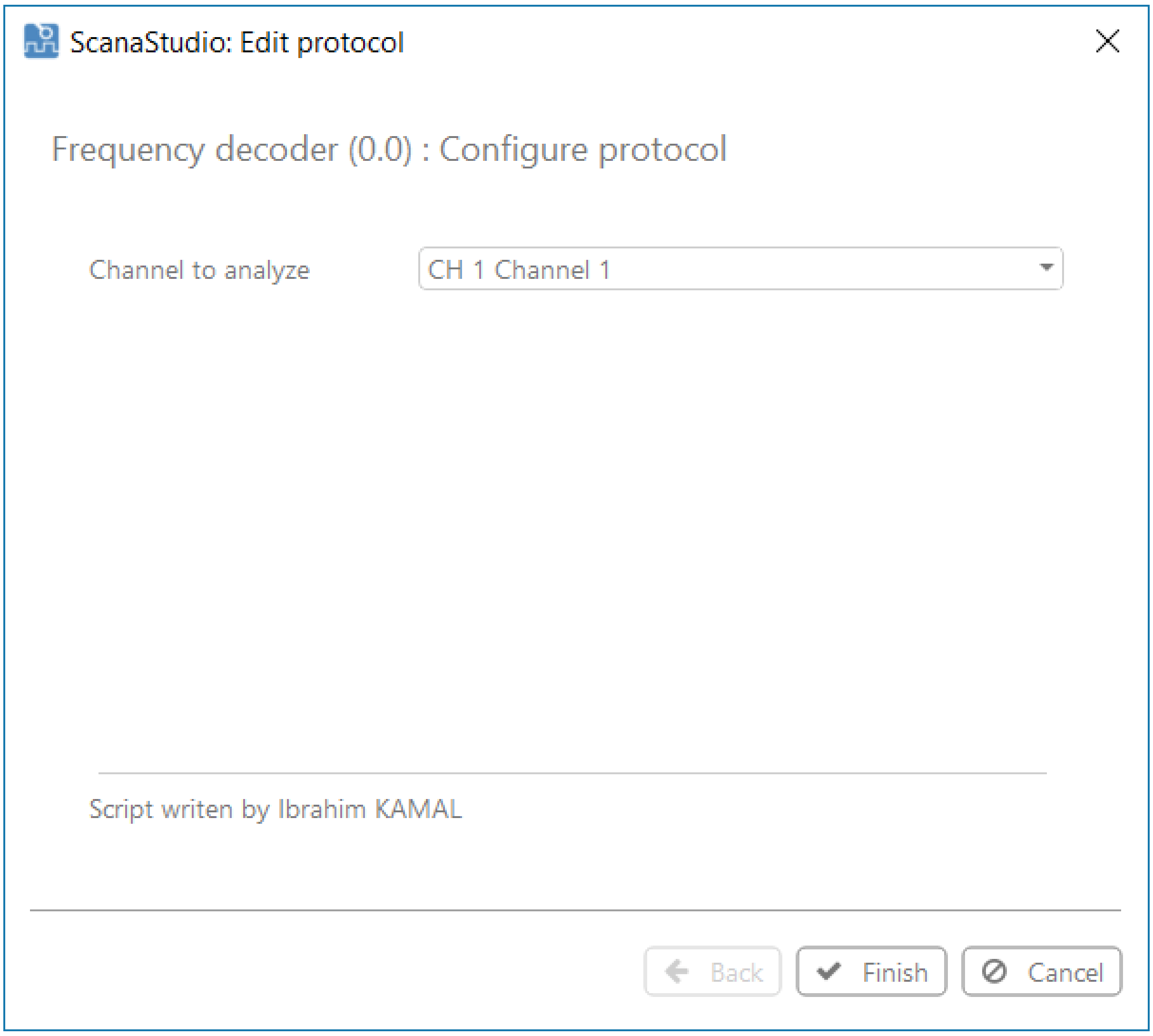
Figure 3: Result of Frequency decoder script

Such a script would have two mandatory entry-point functions:

- A function to draw the GUI that allows the user of the script to select which channel to decode (this entry-point function needs to be named `on_draw_gui_decoder()`)
- A function to decode the logic signals into meaningful, frequency values expressed in Hz, Khe or Mhz. This entry-function, as you may have guessed, needs to be named `on_decode_signals(resume)`

Building the GUI

For that simple script, the GUI is very minimalistic, since there is only one option that needs to be set (or changed) by the user of the script, which is the channel which shall be analyzed by this decoder.



Frequency decoder (0.0) : Configure protocol

Channel to analyze: CH 1 Channel 1

Script writen by Ibrahim KAMAL

Figure 4: Frequency decoder GUI

To create such a GUI, first, we have to create the `on_draw_gui_decoder()` function: the entry-point function which will be called by ScanaStudio when the GUI needs to be redrawn.

```
function on_draw_gui_decoder()
{
  ScanaStudio.gui_add_ch_selector("freq_ch","Channel to analyze","Freq"
    );
}
```

The function `ScanaStudio.gui_add_ch_selector(...)` simply tells ScanaStudio to add a GUI component called "ch_selector" (Channel selector). As the name implies, this adds a combo box with the list of channels supported by the device currently used. This function takes 3 parameters:

1. The ID of the GUI element. This should be a unique text string, and it will be used later to retrieve the value of the combo box (the user's choice). In our example, we used "freq_ch".
2. The text label to show next to the combo box.
3. The default (new) channel name. A decoder can prompt the user to rename the labels of logic analyzer channels to something more meaningful than the standard "Channel n" label. In our case, we can prompt the user to rename the channel to "freq".

Note: The title of the GUI cannot be changed via the `on_draw_gui_decoder()` function. It is automatically generated from the meta-information provided by the script.

Decoding the signals

Another entry-point function needs to be implemented: the `on_decode_signals(resume)` function. This function is called each time ScanaStudio needs to decode logic signals using that script. In some situations, this function may be called repeatedly as new logic signals come in (in case of a logic analyzer that supports live data stream). The function argument "resume" is **`true`** if it's a repeated function call for the same capture. At the very first call to that function, the `resume` parameter is **`false`**. All global variables retain their value between different calls with `resume == true`.

```
function on_decode_signals(resume)
{
  if (!resume) //If resume == false, it's the first call to this
    function.
  {
      //initialization code goes here, ex:
      ScanaStudio.console_info_msg("Frequency analyzer initialized");
      state_machine = 0;
      sampling_rate = ScanaStudio.get_capture_sample_rate();
      freq_ch = ScanaStudio.gui_get_value("freq_ch");
      ScanaStudio.trs_reset(freq_ch); //reset the trs iterator.
      last_falling_edge = last_rising_edge = -1;
  }

  while (ScanaStudio.abort_is_requested() == false)
  {
    if (!ScanaStudio.trs_is_not_last(freq_ch))
    {
      break;
    }
    switch (state_machine)
    {
      case 0: //search for rising edge
        trs = ScanaStudio.trs_get_next(freq_ch); //fetch next transition
        if (trs.value == 1) { //Rising edge found?
          if (last_rising_edge != -1){ //if it's not the very first rising edge
            period = (trs.sample_index - last_rising_edge) / sampling_rate; //period in [s]
            frequency = 1/period; //in [Hz]
            ScanaStudio.dec_item_new(freq_ch,last_rising_edge,trs.sample_index);
            ScanaStudio.dec_item_add_content("F = " + ScanaStudio.engineering_notation(frequency,3)         + "Hz");
          }
          last_rising_edge = trs.sample_index;
          state_machine++;
        }
        break;
      case 1: //search for falling edge
        trs = ScanaStudio.trs_get_next(freq_ch); //fetch next transition
        if (trs.value == 0){ //Falling edge found?
          last_falling_edge = trs.sample_index;
          state_machine = 0;
        }
        break;
      default:
        state_machine = 0;
    }
  }
}
```

We're not going to analyze this script in detail for the time being, but it's worth noting that it's fully functional.

Putting the finishing touches

Finally you should add a meta-information bloc (or update it if you only modified the script). For our example, we'll write the following meta-information bloc:

```
/*
<NAME> Frequency decoder </NAME>
<DESCRIPTION>
Analyze logic signals to shows its frequency. This script's main aim
is to provide a simple example to be inspired from when creating a
new decoder.
</DESCRIPTION>
<VERSION> 0.0 </VERSION>
<AUTHOR_NAME>  Ibrahim KAMAL </AUTHOR_NAME>
<AUTHOR_URL> i.kamal@ikalogic.com </AUTHOR_URL>
<COPYRIGHT> Copyright Ibrahim KAMAL </COPYRIGHT>
<LICENSE>  This code is distributed under the terms of the GNU General
   Public License GPLv3 </LICENSE>
<RELEASE_NOTES>
V0.0:  Initial release.
</RELEASE_NOTES>
*/
```

GUI related functions

This chapter will deal with the different GUIs (Graphical User interfaces) used to interact with the user of your script.

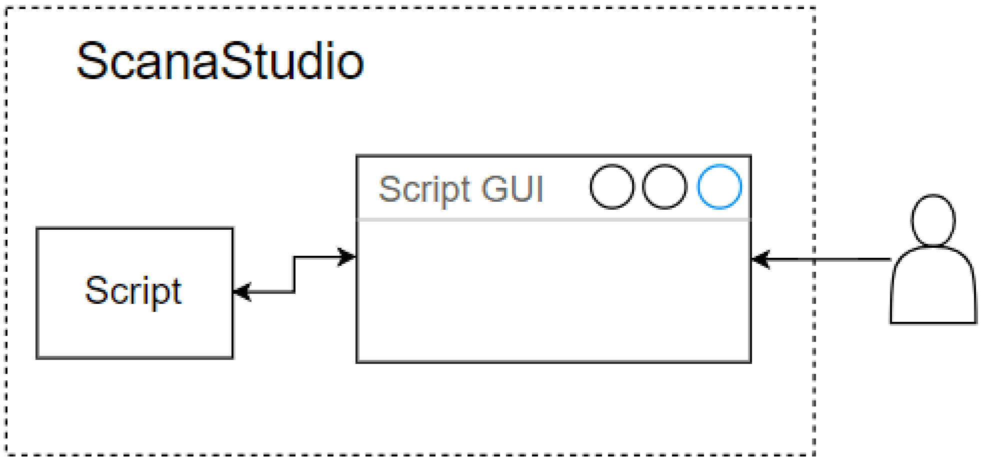


Figure 5: Script/User interaction via GUI

Currently, ScanaStudio uses (and needs) a GUI for decoder scripts and trigger sequence builder scripts. Users of those two kinds of scripts may very well have absolutely no experience in programming and/or JavaScript, so the GUI is here to allow those users to configure different parameters that will affect the operation of the script.

Here are the main groups of GUI related functions:

- Functions to construct (draw) the GUI
- Functions to evaluate (and validate) the choices set by the user in the GUI (optional)
- A function to retrieve the choices set by the user in the GUI

Note on writing convention: Through all this document, a function's parameter is considered to be numerical values unless it's written in "quotation_marks", which means the function expects a text string for this parameter.

GUI entry-point functions

As described in previous chapters, ScanaStudio will search for special "entry-point" functions in your script to perform certain actions. There are several entry-point functions used to construct a GUI, depending on what it will be used for:

```
function on_draw_gui_decoder()
{
  // function called when ScanaStudio needs to display the protocol
      decoder GUI
}

function on_draw_gui_trigger()
{
  // Function called when ScanaStudio needs to display the trigger
      sequence generator GUI
}

function on_draw_gui_signal_builder()
{
  // Function called when ScanaStudio needs to display the signal
      builder GUI
}
```

Any script that builds a GUI must be in one of these functions (or in a function that is only called from one of these functions).

Optionally, the script may implement other entry level functions used to evaluate the choices set by the user in the GUI, and prevent him from going further if some choices are incoherent or unlogic. The evaluation function can even display a message to the user explaining what choice is incoherent.

The GUI evaluation entry point functions (which will be discussed in detail later in this chapter) are:

- `on_eval_gui_decoder()`
- `on_eval_gui_trigger()`
- `on_eval_gui_signal_builder()`

Those functions are optional: If one is omitted, the corresponding GUI will always be considered to be valid, which may be the case for some scripts.

GUI construction

Below is the exhaustive list of functions exposed by the `ScanaStudio` object that can be used to build a GUI. Please note that GUI is drawn in the same order as the functions used to append different elements to it.

ScanaStudio.gui_add_ch_selector("id","caption","new_channel_name")

Description: this function appends a channel selector to the GUI, that is, a combo box with a list of channels. The exact number of channels in that combo box will depend on the logic analyzer device being used. This is probably the most used GUI elements.

Parameters

- "id" : a unique text string used to identify this GUI item (not displayed to the user)
- "caption" : a caption to describe (in a few words) what the channel will be used for. (e.g. You could use "Clock" to let the user select the channel to be used a clock input for some protocol.)
- "new_channel_name": the default (new) channel name. A decoder can prompt the user to rename the labels of logic analyzer channels to something more meaningful than the standard "Channel n" label.

Context : Global

ScanaStudio.gui_add_text_input("id","caption","default_value")

Description: This function appends a text input field to the GUI.

Parameters:

- "id" : a unique text string used to identify this GUI item (not displayed to the user)
- "caption" : a caption to describe this GUI item.
- "default_value" the value to be filled in the text box by default.

Context : Global

ScanaStudio.gui_add_baud_selector("id","caption",default_value)

Description: This function appends a BAUD rate selector element. The baud rate selector is an enhanced text box; the user can manually type a BAUD rate (like 115200), but he can also click on the "Auto detect" button as show in the image below.

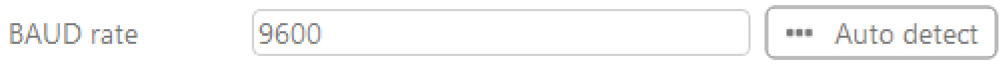


Figure 6: BAUD Rate selector example

Clicking the Auto detect button shows a dialog like in the image below:

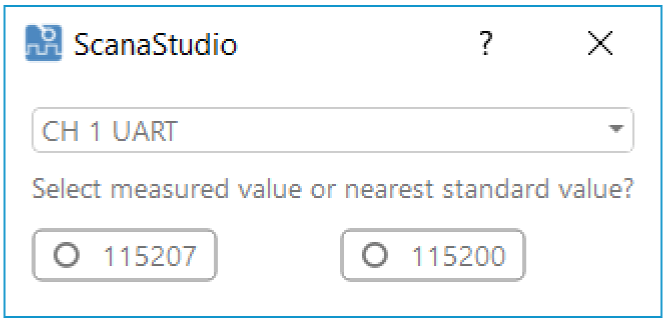


Figure 7: BAUD rate auto detect dialog

For each channel in the drop down list, ScanaStudio will offer two options to the user: the exact baud rate measured (guessed) from the signals on that channel, and the nearest standard baud rate (picked from a list of well known BAUD rates like 115200 for serial UART, or 1000000 for CAN bus). Clicking on one of those two baud rate would autofill the BAUD rate text box in the initial GUI item.

Parameters:

- “id” : A unique text string used to identify this GUI item (not displayed to the user)
- “caption” : A caption to describe this GUI item.

Context : Global

ScanaStudio.gui_add_check_box(“id”,“caption”,is_checked)

Description: This function appends a checkbox to the GUI.

Parameters:

- “id” : A unique text string used to identify this GUI item (not displayed to the user)
- “caption” : A caption to describe this GUI item.
- is_checked : A boolean indicating if the checkbox is checked by default. You can either use the keyword `true`/`false` or an integer `1`/`0`.

Context : Global

ScanaStudio.gui_add_combo_box(“id”,“caption”)

Description: This function appends a combo box to the GUI. The content of the combo box is added using the function `ScanaStudio.gui_add_item_to_combo_box`. Here is an example code that appends a combo box to the GUI, then inserts 3 elements to that combo box:

```
ScanaStudio.gui_add_combo_box("my_combo","Select an option");
  ScanaStudio.gui_add_item_to_combo_box("Option 1");
  ScanaStudio.gui_add_item_to_combo_box("Option 2");
  ScanaStudio.gui_add_item_to_combo_box("Option 3");
```

Parameters:

- "id" : A unique text string used to identify this GUI item (not displayed to the user)
- "caption" : A caption to describe this GUI item.

Context : Global

ScanaStudio.gui_add_item_to_combo_box("item_txt",is_selected)

Description: This function appends a new line (new item) to the combo box that was last appended to the GUI.

Parameters:

- "item_txt" : The text to display for that item
- is_selected : a boolean value that defines weather an item is selected by default. If this parameter is ignored, the value `false` will be used by default. If no any element as the parameter `is_selected` set to `true`, then the very first element will be selected by default.

Context : Global

ScanaStudio.gui_add_engineering_form_input_box("id", "caption", min_value, max_value, default_value, "unit")

Description: This function adds an input box specifically made for numbers that need to be entered in engineering format (e.g.: 2 ms or 25 MHz). By engineering format, we mean a number that is composed of:

- a value,
- a prefix
- and a unit.

For example, the following line of code:

```
ScanaStudio.gui_add_engineering_form_input_box("rate","Bit rate",100,1
    e6,125e3,"Bit/s");
```

will create this GUI item:

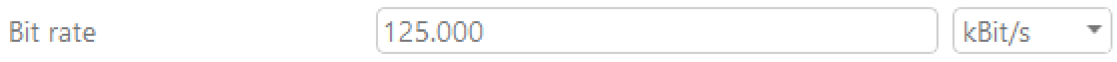


Figure 8: example bit rate selector using engineering form input box

Parameters:

- "id": A unique text string used to identify this GUI item (not displayed to the user).
- "caption": A caption to describe this GUI item.
- min_value: along with max_value, this parameter lets ScanaStudio calculate the most suitable prefixes for the unit (the prefixes are "n" for nano, "u" for micro, "m" for milli, "k" for kilo, etc.).
- default_value: the default value to be displayed in the input box when the GUI item is created
- "unit": a text, usually composed of one or a few characters. The prefix and the unit will be combined together on the GUI item, to form engineering values like "KHz" or "mS".

Context: Global

ScanaStudio.gui_add_info_label("text")

Description: This function appends an information label. This GUI element is not used to get input from the user, but rather to give him some instructions. There is no "id" parameter for this GUI element, for the simple reason that it does not need to be?? addressed later, neither can this GUI element hold an information to be retrieved later.

Parameters:

- "text": This parameter hold the text to be displayed.

Context : Global

ScanaStudio.gui_add_separator("title")

Description: This function simply adds a separator (horizontal ruler) between two sections in the GUI. This function has no "id" parameter, and cannot be addressed later by the script.

Parameters:

- "title": This optional parameter hold the title to be displayed on the ruler.

Context : Global

File system GUI elements

Some scripts can access files either for reading or writing. For example, an I2S protocol decoder (used to transmit audio signals) can output decoded data to a *.wav file. Similarly, a protocol decoder script may refer to some local configuration file - for example - to adjust various parameters that would be complex and tedious to set in GUI. Please note that for security reasons, ScanaStudio scripts cannot have access to the actual file path set by the user in the GUI, nor can it arbitrarily read or write files on the user's computer. Only the files selected by the user via the provided GUI items are accessible to the script (but without exposing the file path to the script).

ScanaStudio.gui_add_file_save("id","caption","extension")

Description: This function adds a GUI item that prompts the user to select a file path for writing. If the file already exists, ScanaStudio will ask the user to confirm if the file can be overwritten.

Parameters:

- "id": Unique ID of this GUI item. This ID is used by the script to select the file to be opened. As stated before, the script will never has access to the file's path; file system operation are only carried via this ID, which effectively limits the scope of the files read/write operation to the file(s) specified by the user of the script in the GUI.
- "caption": Text that will appear next to the save file item. This is a good place to describe what this file will be used for.
- "extension": A text string that defines the extension to be used by ScanaStudio's file dialog to filter the files. use "." or leave empty to show all files. use "*.csv" to show only CSV files, for example.

Context: Global

ScanaStudio.gui_add_file_load("id","caption","extension")

Description: This function is similar to the `ScanaStudio.gui_add_file_save` with the exception that a "load file" dialog will be used. That also means that this GUI item *cannot* be used to create a new file, it can only be used to select an existing file.

Hidden GUI elements

Hidden GUI elements can be treated like any other elements, with the difference that it wont be visible to the user. They exist for one particular purpose: sub-decoders.

First, let's explain what are **sub-decoders**: A sub-decoder is a decoder that is called by another "high level" decoder. By "high level", we mean a decoder that is accessed by the user, for which a GUI is displayed. A typical application of sub-decoders, is to build a temperature sensor protocol decoder that is based on a low level I2C decoder. Being able to call a sub-decoder from the high level decoder prevents the high level script from implementing the whole I2C decoding layer. Any decoder can become a sub-decoder provided that it's called by another decoder.

Since the sub-decoder has no way of displaying a GUI, it's the high level decoders' job to expose hidden GUI items with the exact same IDs expected by the sub-decoder.

ScanaStudio.gui_add_hidden_field("id","value")

Description: This function appends a new hidden GUI field.

Parameters:

- "id" : A text string used to identify this GUI item (not displayed to the user). This ID needs to be exactly the same as the ID of a GUI element of a sub-decoder.
- "value" : A text or numerical value attached to that hidden GUI item. **Context** : Global

Examples

```
//create a numeric item with the value = 1.
//Note that there are no "quotation marks" around the value parameter
ScanaStudio.gui_add_hidden_field("id_num",1);
//Create a text value item
ScanaStudio.gui_add_hidden_field("id_txt","text value");
//Create a boolean value item
ScanaStudio.gui_add_hidden_field("id_bool","true");
```

ScanaStudio.gui_set_hidden_field("id","value")

Description: This function changes the value of a hidden field. This can be useful when a decoder needs to change the behavior of a sub-decoder.

Parameters:

- "id" : ID of the hidden field to be changed
- "value" : New value to be set for that hidden field. It can be a number or a text string

Context : Global

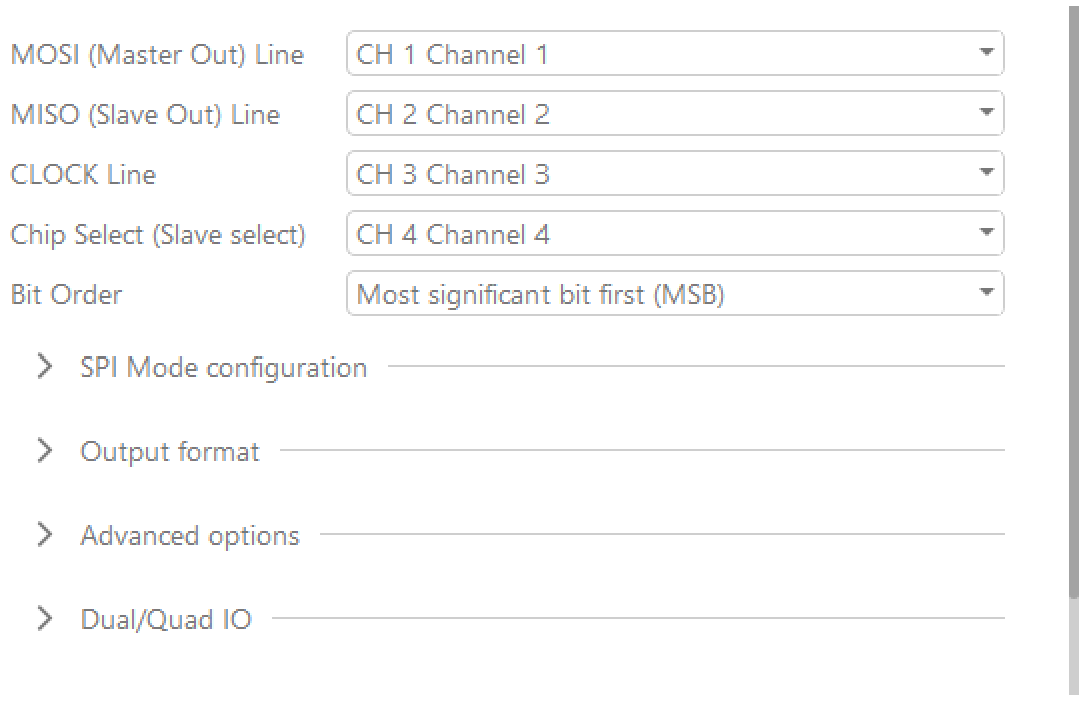
As with the `gui_add_hidden_field` function, you may set a numeric value by ignoring the "quotation mark" around the `value` parameter, e.g.:

```
ScanaStudio.gui_set_hidden_field("id_numeric",5);
```

Grouping GUI elements in tabs

It is possible to group GUI elements in separate accordion tabs. Each tab has a different title (defined by the "caption" parameter). Accordion tabs are great when it's needed to toggle between hiding and showing a large amount of content.

An example for the usage of tabs is the SPI protocol script, which uses tabs to categorize advanced configuration options, and prevent cluttering the user interface. Only the most important and most used parameters are visible right from the start.



MOSI (Master Out) Line	CH 1 Channel 1
MISO (Slave Out) Line	CH 2 Channel 2
CLOCK Line	CH 3 Channel 3
Chip Select (Slave select)	CH 4 Channel 4
Bit Order	Most significant bit first (MSB)

Figure 9: SPI protocol GUI configuration

Putting GUI elements inside accordion tabs is done using the functions presented below.

ScanaStudio.gui_add_new_tab("caption",is_expanded)

Description: This function creates a new accordion tab element. All GUI elements appended after a call to `ScanaStudio.gui_add_new_tab` is added to that tab.

Parameters:

- "caption" : A title to describe this tab.
- is_expanded : A boolean value that defines weather a tab is expanded by default.

Context : Global

ScanaStudio.gui_end_tab()

Description: This function ends the tab that was previously started with the function `gui_add_new_tab(...)`. Any GUI element added after a call to `gui_end_tab()` will not be grouped under a tab that can be expanded or minimized.

Context: Global

Examples:

```
ScanaStudio.gui_add_new_tab("Test tab 1",true);
  ScanaStudio.gui_add_check_box("c01","Option 1",false);
  ScanaStudio.gui_add_check_box("c02","Option 2",false);
ScanaStudio.gui_end_tab();

ScanaStudio.gui_add_new_tab("Test tab 2",false);
  ScanaStudio.gui_add_check_box("c11","Option 1",false);
  ScanaStudio.gui_add_check_box("c12","Option 2",false);
ScanaStudio.gui_end_tab();

ScanaStudio.gui_add_new_tab("Test tab 3",false);
  ScanaStudio.gui_add_check_box("c21","Option 1",false);
  ScanaStudio.gui_add_check_box("c22","Option 2",false);
ScanaStudio.gui_end_tab();
```

The above example would produce the GUI interface shown in the image below:

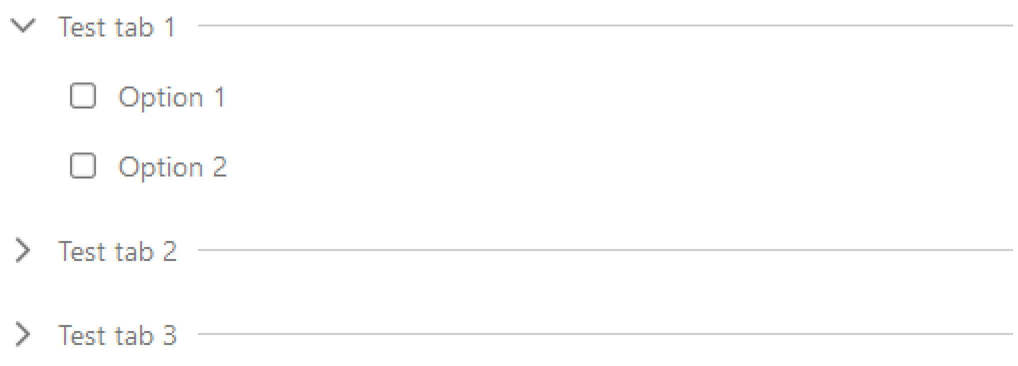


Figure 10: GUI tabs example

Selectable containers

Selectable containers are special GUI constructs that have three roles: * Grouping GUI elements in containers * Ensuring only one container is visible and selected at a given moment. * Knowing which container is chosen by the user

In other words, selectable containers are used when only one among several GUI designs alternatives *should* be used, depending user choices. Each one of those alternatives can have a totally different GUI, thus, this function groups the GUI elements of each alternative in a separate exclusive container.

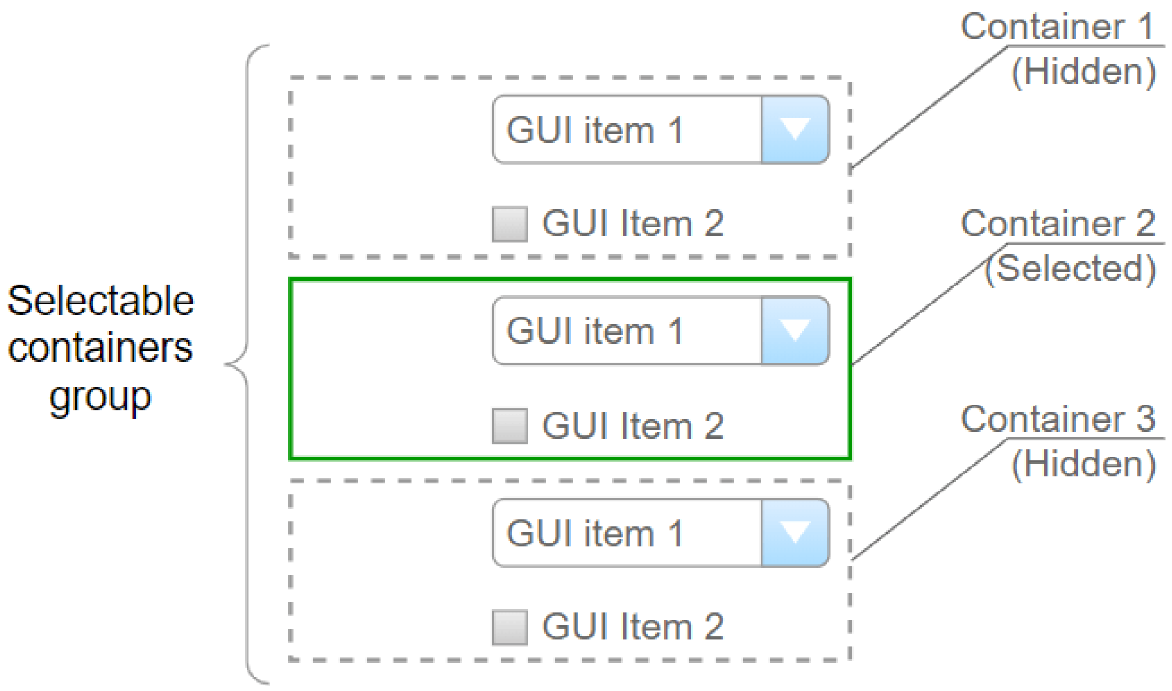
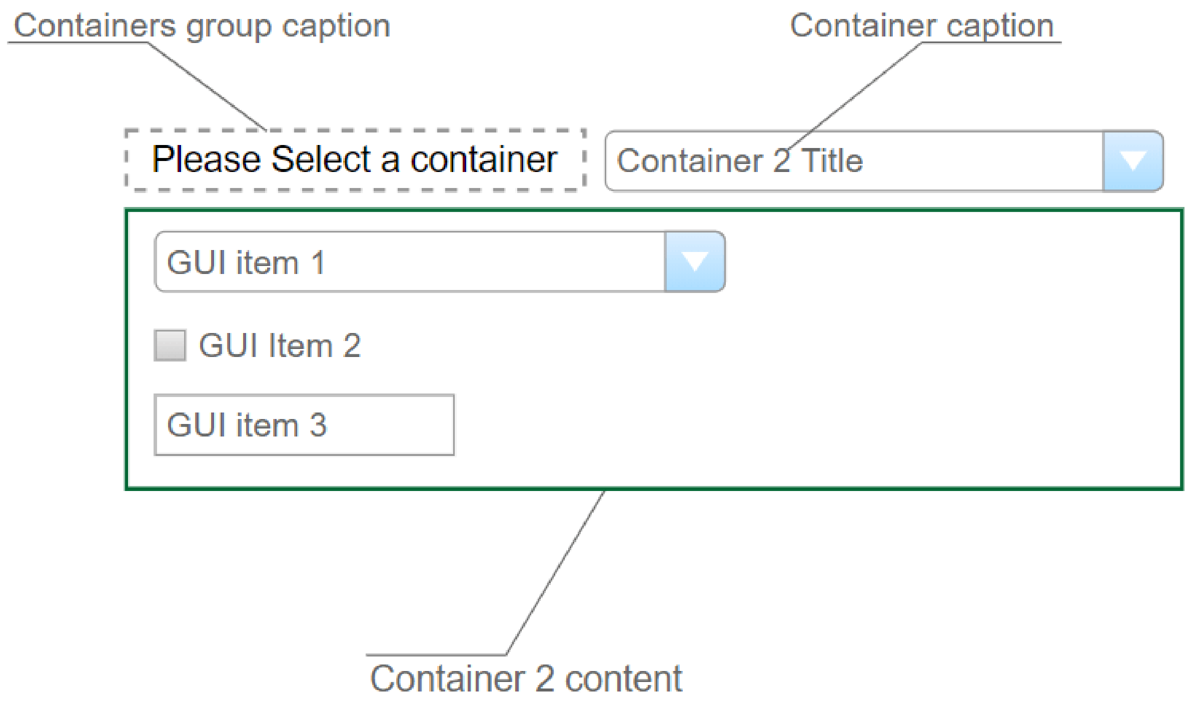


Figure 11: Selectable containers group

The script can later retrieve the state of each container (whether it's selected or not), and decide what GUI elements to consider from which container.

Only one container in a containers group - the one that is selected - is displayed to the end user. The image below shows the vocabulary used to describe Selectable containers' GUI.



Putting GUI elements inside containers, and putting containers inside a containers group is done using the functions presented below.

ScanaStudio.gui_add_new_selectable_containers_group("id","caption")

Description: This function creates a new selectable containers group. A call to this function *must* be followed by one or more calls to `ScanaStudio.gui_add_new_container`. Other GUI elements (like combo boxes, text input or check boxes) cannot be added "inside" the selectable_containers_group; a container must be created first, then GUI items can be added to that container.

Parameters:

- "caption": Title of the containers group
- "id": A unique text string used to identify this GUI item (not displayed to the user). This unique ID can later be used to retrieve the index of the selected container in a group (See the function `gui_get_value()` for more information on that matter).

Context: Global

ScanaStudio.gui_end_selectable_containers_group()

Description: This function ends the containers group that was previously started with the function `gui_add_new_selectable_containers_group(...)`.

Context: Global

ScanaStudio.gui_add_new_container("caption",is_selected)

Description: This function creates a new container. A container can only be created between `gui_add_new_selectable_containers_group` and `gui_end_selectable_containers_group` calls.

Parameters:

- "caption" : A title to describe this container.
- id_selected : A boolean value that defines weather a container is selected by default. Only one container in a containers group should have the property `is_selected` set to `true`.

Context: Global

ScanaStudio.gui_end_container()

Description: This function end the container that was created by `gui_add_new_container`.

Context: Global

GUI evaluation and validation

GUI evaluation is a process by which a GUI is tested against any incoherencies. By incoherencies, we mean choices made by the user that would yield to unexpected or wrong results. For example, choosing the same channel twice as the clock and the data for a protocol decoder is incoherent.

The GUI evaluation functions prevents the user to go any further until incoherencies are fixed, as it is described in the diagram below:

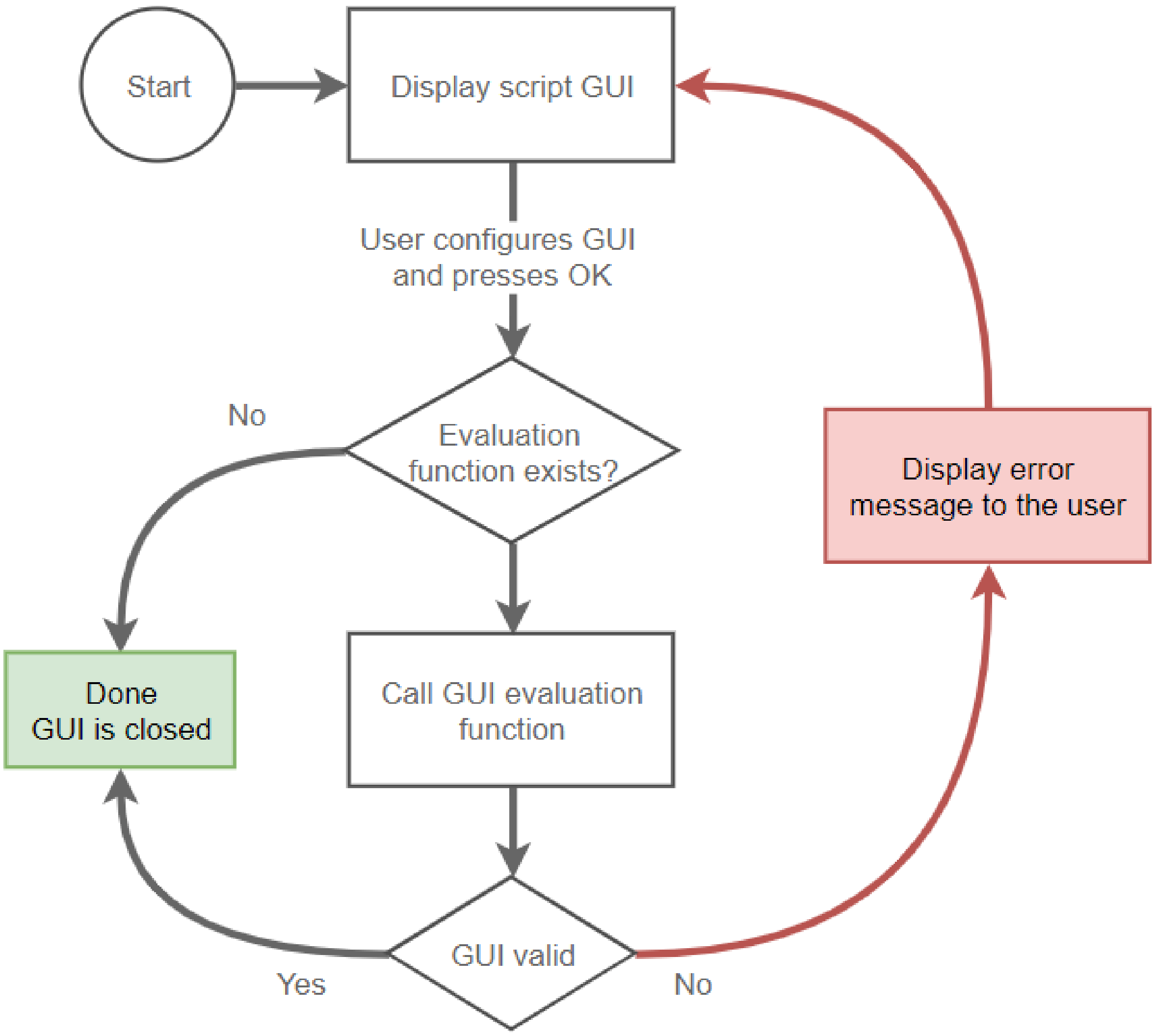

Start
Display script GUI
User configures GUI and presses OK
Evaluation function exists?
No
Done GUI is closed
Call GUI evaluation function
GUI valid
Yes
No
Display error message to the user

Figure 12: GUI Evaluation process

All GUI interfaces can be evaluated using one of the entry-function below:

- `on_eval_gui_decoder()`
- `on_eval_gui_trigger()`
- `on_eval_gui_signal_builder()`

Please note that this functions is optional: If omitted, the corresponding GUI will always be considered as valid.

All these functions behave in the same way: When called by ScanaStudio, they have to evaluate the GUI and return an empty string ("") in case of a valid GUI, or a text string describing the problem.

Below is an example of a decoder GUI evaluation function:

```
function on_eval_gui_decoder()
{
  if (ScanaStudio.gui_get_value("ch_data") == ScanaStudio.gui_get_value
     ("ch_clock"))
  {
      return "Error: Data and clock can't share the same channel";
  }
  return ""; //All good.
}
```

GUI data retrieval

As discussed in this chapter, each GUI element has a unique ID. This ID is used to retrieve the values set by the user in the GUI (which is the whole purpose of the GUI in the first place).

A single function allows GUI data retrieval for any GUI element:

ScanaStudio.gui_get_value("id")

Description: This function returns the value of the GUI element "id". For a combo box or channel selector, this function will return the index of the element selected by the user (0 based).

For checkboxes, it will return `true` or `false` depending on the choice of the user.

For other input boxes (text or numbers), this function will return the characters as they were entered by the user.

For tab GUI element, it will return `true` or `false` depending on weather a tab is selected or not. Obviously, only one tab can be selected at a given time, hence, only one tab's ID will return the value `true`.

For selectable containers group, this function will return the index of the container selected by the user.

Parameters:

- "id": text string representing the GUI element.

Context : Global

Complete example

As an example the code below is functional (but useless) decoder script:

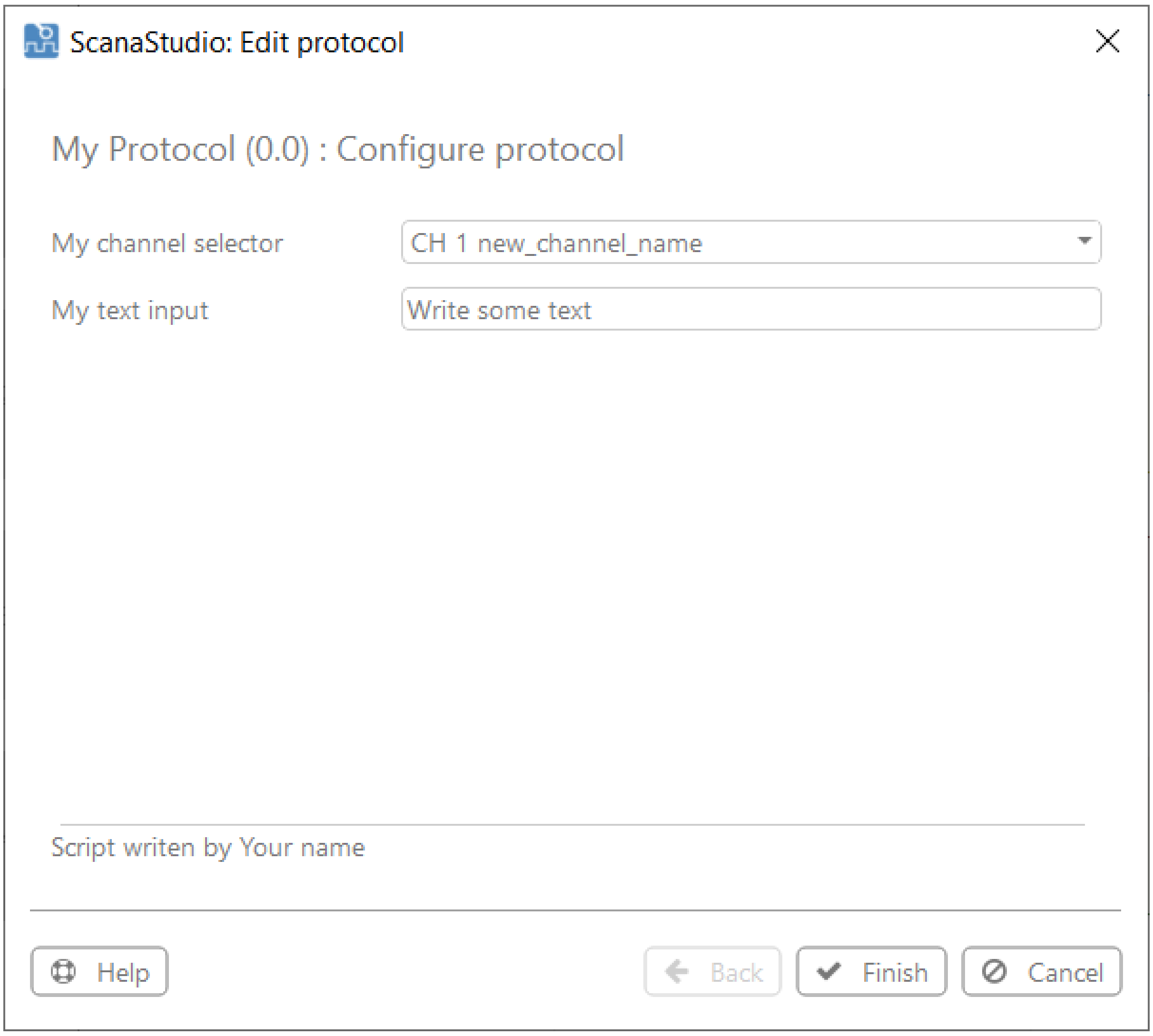
```
/* Protocol meta info:
<NAME> My Protocol </NAME>
<DESCRIPTION>
My protocol can decode pretty much any logic signal!
</DESCRIPTION>
<VERSION> 0.0 </VERSION>
<AUTHOR_NAME>  Your name </AUTHOR_NAME>
<AUTHOR_URL> your@email.or.website </AUTHOR_URL>
<HELP_URL> https://github.com/ikalogic/ScanaStudio-scripts-v3/wiki </
    HELP_URL>
<COPYRIGHT> Copyright your name </COPYRIGHT>
<LICENSE>  This code is distributed under the terms of
 the GNU General Public License GPLv3 </LICENSE>
<RELEASE_NOTES>
V0.0:  Initial release.
</RELEASE_NOTES>
*/

//Decoder GUI
function on_draw_gui_decoder()
{
  ScanaStudio.gui_add_ch_selector("ch","My channel selector","
      new_channel_name");
  ScanaStudio.gui_add_text_input("text","My text input","Write some
      text");
}

var ch,text;
function on_decode_signals(resume)
{
  ch = ScanaStudio.gui_get_value("ch");
  text = ScanaStudio.gui_get_value("text");

  ScanaStudio.console_info_msg("The selected channel is:" + (ch+1) +
  ", and the input text is: "+ text);
}
```

This example creates a simple GUI with two elements as shown in the image below:


ScanaStudio: Edit protocol

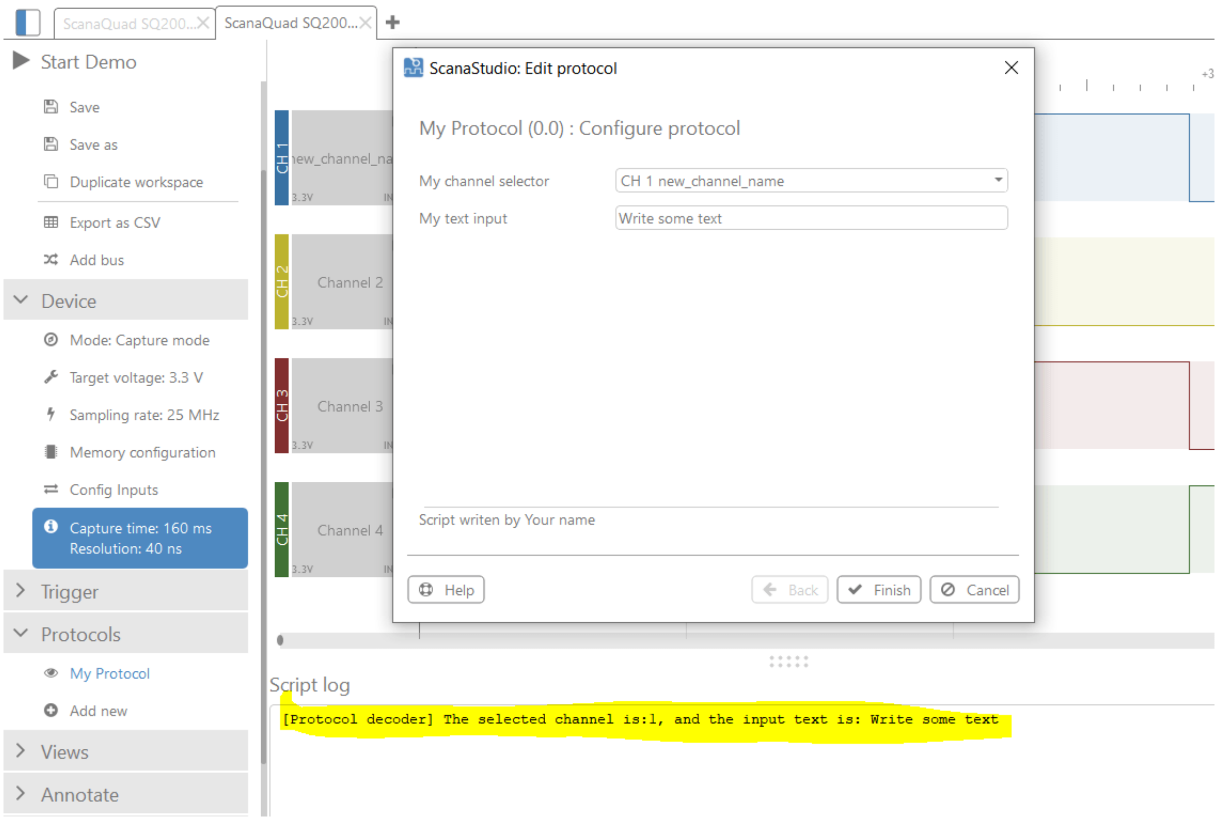
My Protocol (0.0) : Configure protocol

My channel selector	CH 1 new_channel_name
My text input	Write some text

Script writen by Your name

Figure 13: Example decoder GUI

In this examples, the GUI values are retrieved and shown in the console:



ScanaStudio: Edit protocol

My Protocol (0.0) : Configure protocol

My channel selector: CH 1 new_channel_name

My text input: Write some text

Script writen by Your name

Script log

```
[Protocol decoder] The selected channel is:1, and the input text is: Write some text
```

Figure 14: output of example decoder showing how to retrieve GUI values

For your information, the console can be displayed by going to the setting menu in ScanaStudio (top right icon) and tick the “Show log” or “Show console” depending on your version of the software.

Protocol decoding

This chapter discusses the process of decoding logic signals via a script. Some of them may refer to this kind of scripts as "protocol interpreter". Their objective is to extract and display meaningful data out of a sequence of brute logic signals (0's and 1's).

To implement a protocol decoder, you will need to perform the following tasks:

- Access the logic signals, which we will also refer to as "navigating thought the samples"
- Build decoder items, which is the final outcome of the protocol decoder: Human readable pieces of information explaining the underlying bits and bytes of the protocol being decoded. (There are other output forms that will be discussed in next chapters.)

Logic signals decoding entry-point function

Logic signals decoding is started by ScanaStudio using the entry-point function `on_decode_signals(resume)`. This function is called each time ScanaStudio needs to decode logic signals using that script. In some situations, this function may be called repeatedly as new chunks of logic signals come in (in case of a logic analyzer that supports live data stream). The function argument "resume" is **`true`** if it's a repeated function call for the same capture. At the very first call to that function, the `resume` parameter is always **`false`**. All global variables (that are declared outside `on_decode_signals()` function) retain their value between different calls with `resume == true`.

```
function on_decode_signals(resume)
{
  if (resume != true) //First call, initialize
  {
      //Initialize your script here
  }

  while (ScanaStudio.trs_is_not_last(pwm_ch))
  {
    // decoding goes here
  }
}
```

The script must be carefully written in a way that supports and makes use of the `resume` parameter. Practically, this means that each time the `on_decode_signals(true)` function is called, it needs to resume from where it left. The way this can be implemented is left to the programmer behind each script, but usually, implementing a state machine is a good start to ensure your `on_decode_signals` function scales up smoothly while more features are added to your script. A global value can hold the current state of the state machine, allowing the operation to be easily resumed from where it left.

The script must also implement the `on_draw_gui_decoder()` entry point function. This function shall display the GUI that will be used to configure the decoder. The choices made by the user of the script in that GUI can be retrieved via the `gui_get_value` function as described in the GUI chapter.

More about logic signals in ScanaStudio

In order to "navigate" through millions (and sometimes billions) of samples in an efficient way, it's important to understand how ScanaStudio stores samples, and how you're meant to browse those samples. Optimizing the speed at which your script accesses samples and transitions is paramount and will naturally affect the speed at which your script can decode the signals.

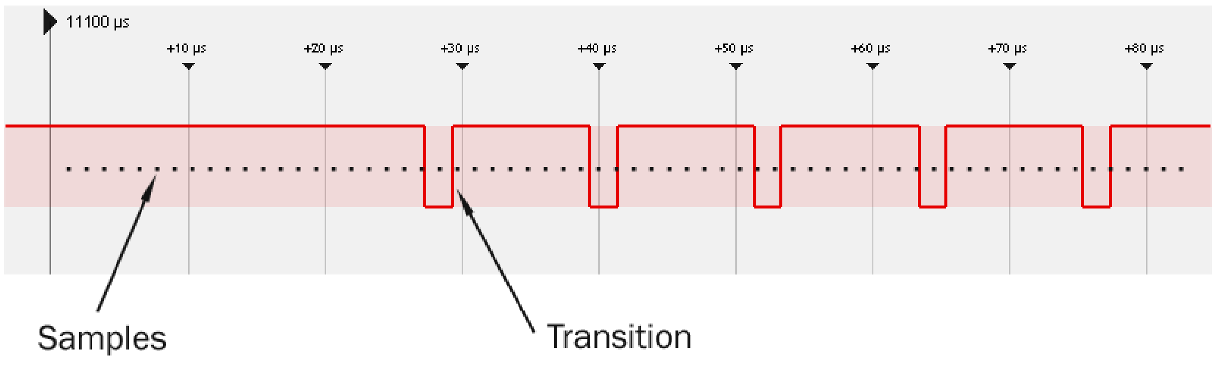


Figure 15: logic samples and transitions

ScanaStudio does not store each and every sample captured. Only transitions are stored using this format:

- Transition polarity (also referred to as "transition value")
- Sample index associated with that transition

Knowing the sampling rate and the sample index, you can calculate the exact position in time of each transition.

This leads us to the `trs_t` object type (trs is the short form for "transition"), which is used all along ScanaStudio decoder scripts. The object `trs_t` is constructed as shown below:

```
//trs_t constructor
function trs_t(sample_index, value) {
  this.sample_index = sample_index;
  this.value = value;
}
```

If you're not very familiar with JavaScript, this is simply an object constructor. It means that a function returning an object of type `trs_t` has two properties: `sample_index` and `value`.

Side note: You may also be wondering why ScanaStudio provides a sample index instead of the time (expressed in seconds) for a specific transition. The answer is about precision and efficiency. When converting a sample index (an integer) to a time, we may lose precision depending on the numbers involved. Also, if ScanaStudio had to convert each and every transition from sample index to time, it would be processing intensive for no particular reason. Working with sample indexes is not more complicated than working directly with time, as you will see in this document.

Now that you know about the most basic building bloc - the "transition" - we can move on to the concept of iterators. ScanaStudio uses iterators to navigate through transitions in a long sequence of logic signals. Each channel has a dedicated iterator used to browse through the logic transitions.

This iterator is very efficient if you request the very first transition, the next transition or the previous one. On the other hand, it's slower if you request - for example - the transition number 10 000, or the transition just after the sample numbered 50 000.

What this means is, for a protocol decoder to perform decoding tasks as quickly as possible, it must navigate through the samples, by looking at transitions, one after the other, in a unique sequential order.

Not observing this simple rule will lead to poor performance, that is, decoders scripts that are very slow to execute.

Samples, time, and sampling rate

It's important to clear up any doubts about samples, time and sampling rate. All three parameters are tightly related.

Samples (and samples indexes) is the only way for a script and ScanaStudio, to agree on a particular instant in a capture. Depending on what protocol you are decoding, time may not be of any importance. That is usually the case for protocols like I2C and SPI which are fully synchronous to a clock signal. For other protocols like serial UART or CAN, time plays an essential role.

So how do you convert a sample index to a time (expressed in seconds)? For that, you need to retrieve the sampling rate which was used to capture the samples. The sampling rate may change from one capture to another but is constant for a given capture. (For the sake of simplicity, we are ignoring the case of state mode operation supported by some late logic analyzer devices, where the sampling rate have no any meaning and may change from one sample to another.)

The sampling rate can be retrieved using this function:

ScanaStudio.get_capture_sample_rate()

Description: This function simply returns the sampling rate for the last capture. **Note**: If the user changed the sampling rate in the device configuration tab in ScanaStudio, this won't change the value of the sampling rate, until a new capture is initiated. In other words, the sampling rate returned by this function is the one that was used to capture the samples displayed on the screen.

Context: Protocol decoder

So for instance, for a given sample index and sampling rate, the time is given by the following equation:

`time = (1 / sampling_rate)* sample_index`

Which could be simply rewritten as:

`time = sample_index/sampling_rate`

Browsing through logic signals

Below is the full list of functions available for the script to browse through the logic signals of a capture.

ScanaStudio.trs_reset(channel_index)

Description: This function set the position of the iterator at the very first transition of the channel `channel_index`

Parameters:

- channel_index : index of the channel (0 Based, that is, the first channel's index in 0).

Context : Protocol decoder

ScanaStudio.trs_get_before(channel_index,target_sample)

Description: This function sets the position of the iterator for channel `channel_index` at the first transition that follows the sample `target_sample`.

Parameters:

- channel_index: index of the channel (0 Based, that is the first channel's index in 0).

- target_sample: index of the sample (0 based).

Return value: Returns a `trs_t` object.

Context: Protocol decoder

ScanaStudio.trs_get_next(channel_index)

ScanaStudio.trs_get_previous(channel_index)

Description: Those two functions advance the iterator to the next/previous transition.

IMPORTANT NOTE: Before those functions can be used for a specific channel, the function `ScanaStudio.trs_reset` or `ScanaStudio.trs_get_before` need to be called first to initialize the iterator.

Parameters:

- channel_index: index of the channel (0 Based, that is the first channel's index in 0).

Return value: Returns a `trs_t` object.

Context: Protocol decoder

ScanaStudio.trs_is_not_last(channel_index)

Description: This function is used to check if the iterator for the channel `channel_index` has reached the last transition

Parameters:

- channel_index: index of the channel (0 Based, that is the first channel's index in 0).

Return value: Returns true if the iterator has still not reached the last transition

Context: Protocol decoder

get_available_samples(channel_index)

Description: This function returns the total number of available samples for a channel. This function is particularly useful when working with asynchronous protocols (like UART) and when decoding is performed live - while samples are being captured. It allows the script to wait until a minimum number of samples is available before attempting to decode a whole word or a whole packet.

Parameters:

- channel_index: index of the channel (0 based).

Return value: Returns the number of samples

Context: Protocol decoder

Using the bit sampler feature

In some situations, navigating using just transitions can be complicated or limiting. For example, if we're decoding serial UART or CAN bus, the position of the bits won't fall on exact transition positions. On the contrary, 0 and 1 bits in a serial data stream are sampled at some point in time between two transitions.

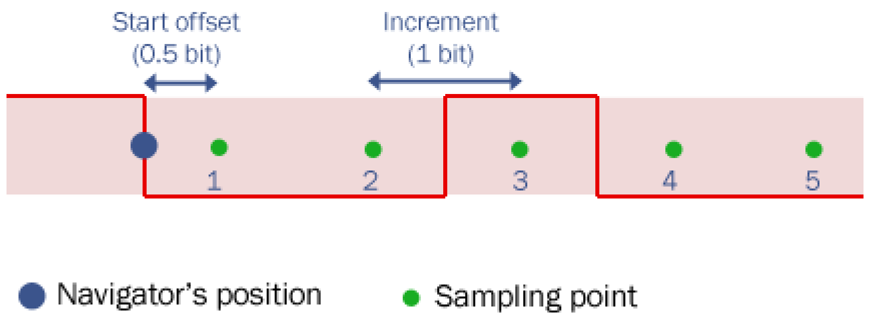


Figure 16: bit sampler

The bit sampler is a helper module in ScanaStudio script that is designed to help you to easily extract the bit values (0's or 1's) at certain sample indexes, without having to worry about the actual transitions and the underlying iterator's position.

The bit sampler is used via those two functions:

- `ScanaStudio.bit_sampler_init`
- `ScanaStudio.bit_sampler_next`

ScanaStudio.bit_sampler_init(channel_index,start_sample_index,samples_increment)

Description: This function initializes the bit sampler for the channel `channel_index`. There is only one bit sampler per channel, so each time a bit sampler is initialized for a channel, the previous bit sampling operation on that same channel will be aborted.

Parameters:

- channel_index: Index of the channel (0 Based, that is the first channel's index in 0).
- start_sample_index: Sample position of the very first bit
- samples_increment: The number of samples increment between two bits. For a known BAUD rate, the `samples_increment` parameter is usually calculated as:

$$samples\_increment = \frac{sampling\_rate}{BAUD\_rate} \quad (1)$$

Context: Protocol decoder

ScanaStudio.bit_sampler_next(channel_index)

Description: This function returns the binary value (0 or 1) of the next bit in a sequence of bits defined by `ScanaStudio.bit_sampler_init`.

Parameters:

- channel_index: Index of the channel (0 Based, that is the first channel's index in 0).

Example:

Let's consider this example logic signal, where the sample counter is displayed to illustrate the example (starting arbitrarily from the sample number 15). The first sample that represents the first bit in a serial data word is sample 21. Then, we need to increment 3 samples to jump at the next bit in that serial word.

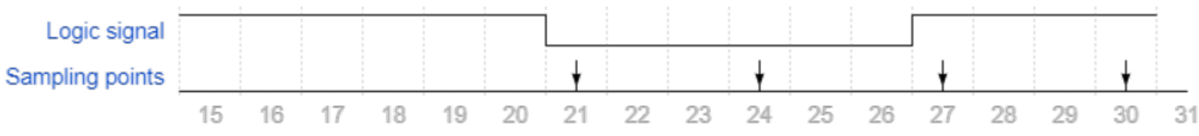


Figure 17: Bit sampler example

The bit sampler initialization and usage for that logic signal would be:

```
ScanaStudio.bit_sampler_init(channel,21,3);
ScanaStudio.bit_sampler_next(channel); //returns 0
ScanaStudio.bit_sampler_next(channel); //returns 0
ScanaStudio.bit_sampler_next(channel); //returns 1
ScanaStudio.bit_sampler_next(channel); //returns 1
```

Context: Protocol decoder

Adding decoder items

Decoder items is the most standard way of displaying decoded information (interpreted bits and bytes of a specific protocol) on the waveform. Historically, in the very earlier versions of ScanaStudio, this was the only way to display the result of a decoder. Later on, other solutions were introduced like the “Hex View” or the “Packet View”.

Back to the decoder items. The image below shows exactly how decoder items are supposed to look like, and provides some essential vocabulary.

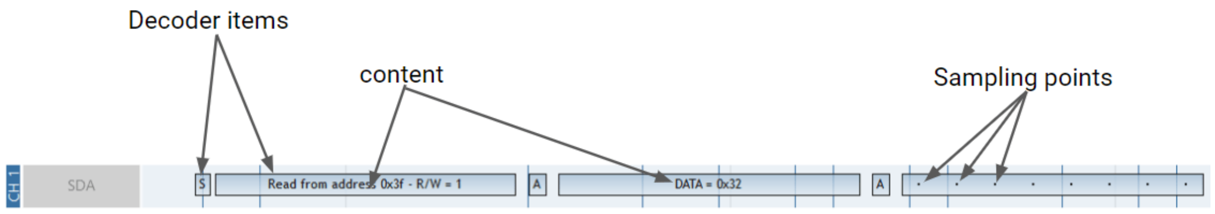


Figure 18: decoder items

Before getting into the details of the different functions used to construct decoder items, it’s important to have a global overview. A decoder item can be seen as a container. This container is materialized on the screen as a box, which is attached to a specific channel and is drawn on a semi-transparent layer on top of the waveforms. It is delimited by a “start_sample_index” and an “end_sample_index”. Those 2 parameters are mandatory for any decoder item. The content of the decoder item, however, is totally optional (you may even draw an empty decoder item, if that makes any sense in your particular protocol). The content is composed of plain text.

Additionally, one may add sample points to a decoder item. Those visual markers are only here to show when the data was sampled according to the specific protocol being decoded. For example, in a CAN bus protocol, the sample points may be used to show the points in time where a valid bit is sampled, and where a stuffed bit is discarded.

Creating decoder items is done in the following steps:

1. Create (open) a new decoder item
2. Add content to last created decoder item
3. Add sampling points (Optional).
4. End (close) the decoder item

In other words, all the function calls that add content to a decoder item need to be encapsulated between `dec_item_new` and `dec_item_end` functions.

Information: Decoder items must be created in a chronological order, that is, the `start_sample` of a decoder item must be bigger that the `end_sample` of the previous sample.

ScanaStudio.dec_item_new(channel_index,start_sample,end_sample);

Description: This function creates a new decoder item. In some situations, the decoder item creation may fail. To verify that a decoder item was added or to know the reason what it wasn't created, you should refer to the returned value as described below.

Parameters:

- channel_index: The index of the channels to which this decoder item is to be attached.
- start_sample: The sample index representing the left boundary of the decoder item box.
- end_sample: The sample index representing the right boundary of the decoder item box.

Return value: Returns a success or error code:

Returned value	Meaning
1	Success
0	Ignored because decoder have been aborted by the user
-1	Ignored because it does follow a chronological order (`start_sample` smaller than previous item's `end_sample`)
-2	Ignored because of incoherent parameters (`start_sample` is bigger than `end_sample`)

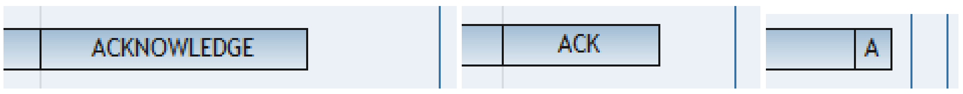
Context: Protocol decoder

ScanaStudio.dec_item_add_content("content");

Description: This function adds (text) content to the last created decoder item. This text content can be anything like HEX data bytes, ASCII characters, plain text, or any association of these. It is possible to add more than one version of the content, to allow ScanaStudio to display the most appropriate version depending on the zoom level. For instance, if the text to be displayed is "Acknowledge", one may add different texts as explained in the example below:

```
ScanaStudio.dec_item_new(0,1000,10000);
ScanaStudio.dec_item_add_content("ACKNOWLEDGE");
ScanaStudio.dec_item_add_content("ACK");
ScanaStudio.dec_item_add_content("A");
```

This will give the following results (screen shots taken at different zoom levels)



Another example of some decoder item content having a mix of text and data in hex format is presented below:

```
ScanaStudio.dec_item_new(0,1000,10000);
ScanaStudio.dec_item_add_content("Data = 0x" + data_value.toString(16))
    ;
ScanaStudio.dec_item_add_content("D = 0x" + data_value.toString(16));
ScanaStudio.dec_item_add_content("0x" + data_value.toString(16));
```

Parameters:

- "content": text content to be appended

Context: Protocol decoder

ScanaStudio.dec_item_add_sample_point(sample_index,"drawing");

Description: This function adds a sample point to the last created decoder item

Parameters:

- sample_index: the index of the sample at which the sampling point should be added
- "drawing": A character to specify what drawing to be used for that sample point (all options are listed in following table).

"drawing" character	Drawing description
"0"	A 0 character
"1"	A 1 character
"P"	A point
"X"	A cross (usually used for "don't care" or stuffed bits)
"U"	An arrow pointing up
"D"	An arrow pointing down
"R"	An arrow pointing right
"L"	An arrow pointing left

Context: Protocol decoder

ScanaStudio.dec_item_emphasize_error()

Description: This function adds emphasis for the last created decoder item, showing this item as an error (by displaying a bold red border around it).

Context: Protocol decoder

ScanaStudio.dec_item_emphasize_warning()

Description: This function adds emphasis for the last created decoder item, showing this item as a warning (by displaying a bold yellow border around it).

Context: Protocol decoder

ScanaStudio.dec_item_emphasize_success()

Description: This function adds emphasis for the last created decoder item, showing this item as an success (by displaying a bold green border around it).

Context: Protocol decoder

ScanaStudio.dec_item_end()

Description: This function, along with `dec_item_new()` encapsulates a decoder item. This function **must** be called after all content have been added, and after any manipulation have been made to the decoder item. If this function is not called, the newly created decoder item will not be displayed, and will be discarded.

Context: Protocol decoder

Packet view

If you haven't already used the Packet View feature of ScanaStudio, it's a good idea to use it (by generating a demo workspace, adding an I2C protocol and generating some demo signals). Packet view has the advantage of totally abstracting the electrical signals from the meaningful data packets. However, as you will notice, packet view still allows a user to jump to a very specific instant in the logic signals that is related to a particular packet.

A packet is composed of two parts: title and content. Also, a packet may contains sub-packets (each sub packet has its own title and content).

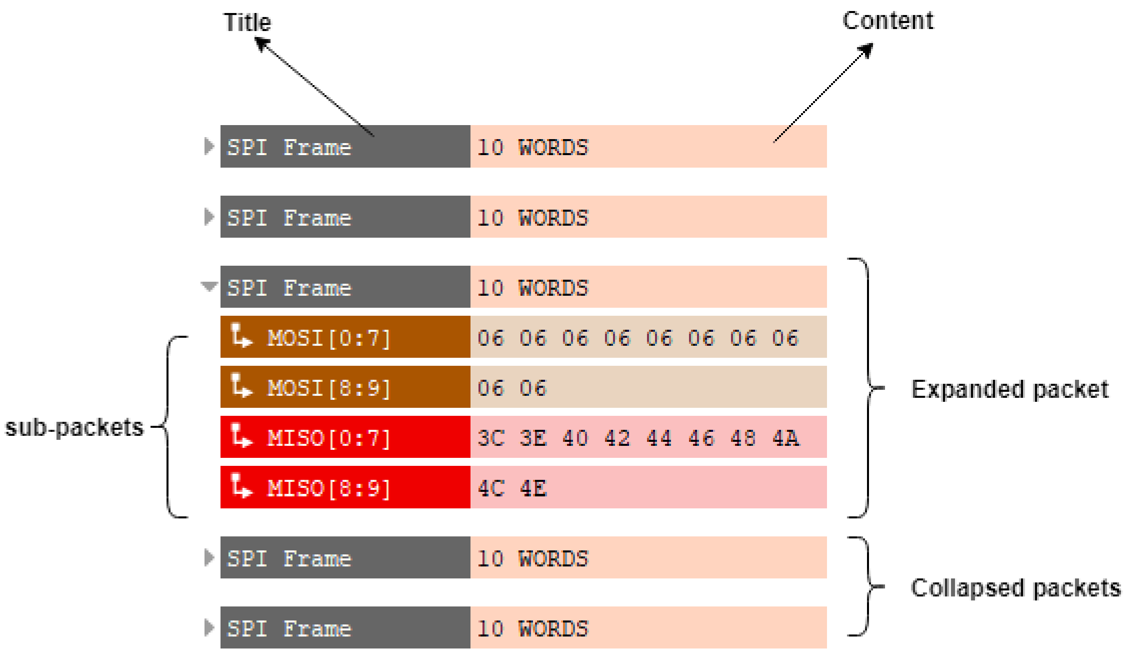


Figure 19: Packets structure

The only difference between a root packets and a sub-packets is that sub packets are contained inside a root packet that can be either collapsed (by default) or expanded. Expanding a root packet reveals the sub packets contained in it.

ScanaStudio.packet_view_add_packet(root, ch, start_sample, end_sample, "title", "content", "title_bg_html_color", "content_bg_html_color")

Description: This function creates a new root packet or sub-packet. A sub packet can only be created (and added to a parent root packet) if a root packet was previously created.

Parameters:

- root: A booloan value. If **`true`**, this packet is created as a root packet. If **`false`**, this packet is created as a sub packet as is appended to the last created root packet.
- ch: 0-based index of the channel related to this packet (if relevant). Set to -1 if not used. If `ch`, `start_sample` and `end_sample` are set to valid values, ScanaStudio will be able to highlight the portion of the signals related to a specific packet.
- start_sample: The sample index pointing at the start of the signals related to that packet. It can be set to -1 if not used.
- end_sample: The sample index pointing at the end of the signals related to that packet. It can be set to -1 if not used.

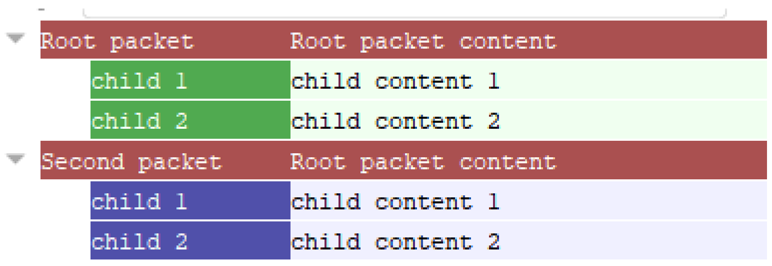
- "title": The title of the packet
- "content": The content of the packet
- "title_bg_html_color": Background color of the title. This is an HTML color encoded as a string. For example, white background is "#FFFFFF"
- "content_bg_html_color": Background color of the content. This is an HTML color encoded as a string.

Context: Protocol decoder.

Example: The following code creates two packets, with 2 child elements each:

```
function on_decode_signals(resume)
{
  ScanaStudio.packet_view_add_packet(true,0,1000,2000,"Root packet","
      Root packet content","#AA5050","#AA5050");
  ScanaStudio.packet_view_add_packet(false,0,1000,2000,"child 1","child
       content 1","#50AA50","#F0FFF0");
  ScanaStudio.packet_view_add_packet(false,0,1000,2000,"child 2","child
       content 2","#50AA50","#F0FFF0");
  ScanaStudio.packet_view_add_packet(true,0,1000,2000,"Second packet","
      Root packet content","#AA5050","#AA5050");
  ScanaStudio.packet_view_add_packet(false,0,1000,2000,"child 1","child
       content 1","#5050AA","#F0F0FF");
  ScanaStudio.packet_view_add_packet(false,0,1000,2000,"child 2","child
       content 2","#5050AA","#F0F0FF");
}
```

When the decode function is called, the PacketView should show the following packets:



```
Root packet     Root packet content
  child 1       child content 1
  child 2       child content 2
Second packet   Root packet content
  child 1       child content 1
  child 2       child content 2
```

Figure 20: example packets with child elements

Please note that the packets may be collapsed by default and may need to be expanded.

Hex View

The HEX view in ScanaStudio is a way to present data bytes in similar presentation as in HEX memory dump. ScanaStudio also allows any byte in the HEX View to be traced back to the logic (electric) signals related to it.

ScanaStudio.hex_view_add_byte(channel_index,start_sample,end_sample,data_byte)

Description: This function appends a new byte to the HEX View.

Parameters:

- channel_index: 0-based index of the channel related to this byte. set to -1 if not used or if irrelevant.
- start_sample: The sample index pointing at the start of the signals related to that byte. Can be set to -1 if not used.
- end_sample: The sample index pointing at the end of the signals related to that byte. Can be set to -1 if not used.
- data_byte: A value that fits in 8 bits (1 byte).

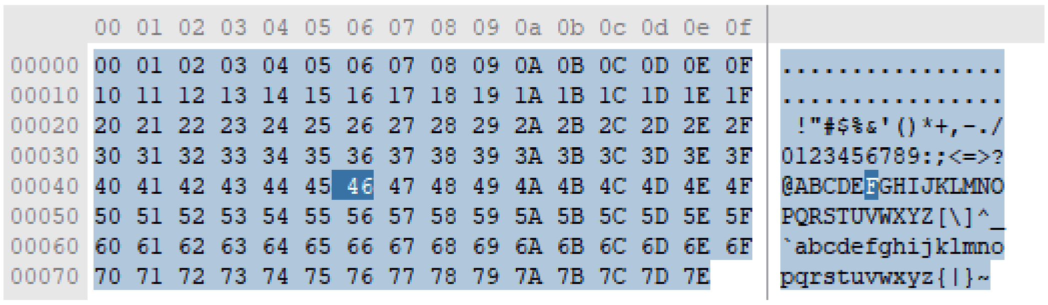
Context: Protocol decoder

Example: The following example fills the hex view with bytes ranging from 0 to 127.

```
function on_decode_signals(resume)
{

  for (var i = 0; i < 127; i++)
  {
    ScanaStudio.hex_view_add_byte(0,100,200,i);
  }
}
```

When the code above is executed, the hex view should show the following HEX dump:



```
      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
00000 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ................
00010 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  ................
00020 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
00030 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
00040 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
00050 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
00060 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
00070 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E     pqrstuvwxyz{|}~
```

Figure 21: HEX View example

Colors

At some point, it may be useful to retrieve the colors of the channels. Typically, this can be used to ensure packets are using the same color as a particular channel. For this purpose, the `get_channel_color()` function is available.

ScanaStudio.get_channel_color(channel_index)

Description: This function return the HTML color (e.g. "#FFFFFF") of a channel

Parameters:

- channel_index: 0-based index of the channel

Return value: Returns the color code in HTML format (e.g. "#FFFFFF" for white).

Context: Protocol decoder

Default packet colors A set of global colors are defined by ScanaStudio. Those standard colors are recommended if you wish to build packets (in the PacketView) that offer the same "look-and-feel" of all other protocols.

The default packet colors are simply global variables defined under `ScanaStudio.PacketColors`

We have defined the following packets element types (or categories if you prefer):

- Wrap: All elements that wrap a packet, like Start, Stop, SOF, EOF, etc.
- Head: Header of a packet
- Preamble
- Data
- Check : Like checksum, CRC or any integrity checking related fields
- Error

- Misc : Anything that does not fall in the categories above.

And for each category, there is color for the Title and the Content of the packet.

To sum up, here are a few example of valid, globally defined, colors: `ScanaStudio.PacketColors.Head.Title` or `ScanaStudio.PacketColors.Data.Content`.

One big advantage of using those pre-defined colors, is that your script will automatically adopt any new color style that was updated in ScanaStudio, and will always be in full harmony with other protocol decoder scripts.

So, for instance, instead of adding a PacketView item this way:

`ScanaStudio.packet_view_add_packet(true,0,1000,2000,"Root packet","Root packet content","#AA5050","#AA5050");`

You could simple write it this way:

```
ScanaStudio.packet_view_add_packet(true,0,1000,2000,"Root packet",
"Root packet content",
ScanaStudio.PacketColors.Head.Title,
ScanaStudio.PacketColors.Head.Content);
```

Sub-decoder scripts

A sub-decoder is a decoder that is called by another "high level" decoder (also known as a nested decoder). By "high level", we mean a decoder that is accessed by the user, for which a GUI is displayed. A typical application of sub-decoders, is to build a temperature sensor protocol decoder that is based on a low level I2C decoder. Being able to call a sub-decoder from the high level decoder prevent the high level script from (re)implementing the whole I2C decoding layer. In that specific scenario, the low level sub-decoder would interpret the logic signals and output I2C packets. The high level temperature sensor decoder would interpret I2C packets provided by the sub-decoder and provide meaningful temperature information. Below is an example architecture of decoder/sub-decoder system.

Note: Along this documents, the terms "sub-decoder" and "low level decoders" are both used and have the same meaning.

The process of having a sub-decoder interpret the logic signals and provide decoded packets for another high level decoder is referred to as "**pre-decoding**".

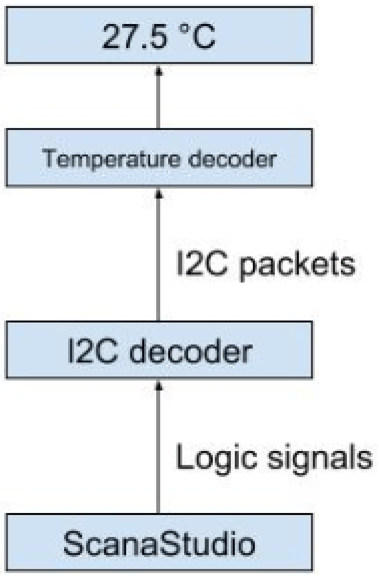


Figure 22: Example achitecture of temperature sensor decoder

Any decoder can become a sub-decoder provided that it's called by another decoder.

A decoder used as a sub-decoder is not "aware" of that situation, thus, it's the high level decoder (the one calling the sub-decoder) to provide all the information needed by the sub-decoder to operate properly. This information is nothing more than GUI values that the sub-decoder would normally rely on to operate. As described in the GUI functions chapter, a sub-decoder cannot display it's GUI to the user, but special GUI elements called “hidden elements” can be used to trick the sub-decoder into behaving as if its GUI was displayed and configured by the user.

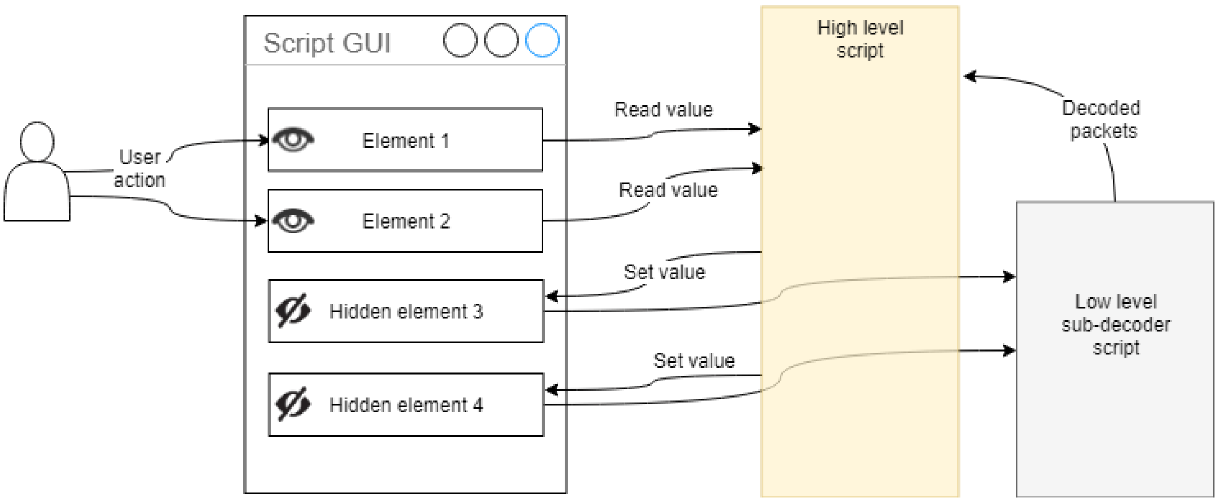


Figure 23: Sub-deocder interactions

Let’s look at things from a programming perspective: a high-level script must ensure that when a low level scripts calls the function `gui_get_value` for a specific GUI ID, it gets a correct value. There are two ways to do that:

1. Create hidden elements with the exact same ID as the one used in the sub-decoder’s GUI (the method described above).
2. Use the same GUI IDs in the high level decoder and sub-decoder, when this is possible and makes sense.

Of course, it’s also possible to mix those two solutions. For example, the SDA and SCL channels for an I2C decoder is a shared information between the high-level decoder and sub-decoder, so it would be wise to give the channel selector GUI elements the same ID chosen by the sub-decoder; this way they can share the same GUI element.

Next comes the most important part of the process, the **pre-decoding**: calling the sub-decoder and making use of the decoded data. This is simply done by calling the `pre-decode()` function as described below. This function will usually take some time to execute (depending on the quantity of logic signals to be analyzed). Then, it will return the newly decoded data packets. Those packets are called “decoder items” and are simply objects of the type `dec_item_t` as defined by this JavaScript constructor:

```
//dev_item_t constructor
```

```
function dec_item_t(channel_index, start_sample_index, end_sample_index
    , content)
{
  this.channel_index = channel_index;
  this.start_sample_index = start_sample_index;
  this.end_sample_index = end_sample_index;
  this.content = content;
}
```

As you may have recognized, this data type simply encapsulates all the information that the script provides for ScanaStudio to draw the decoder items on the waveforms. Once you get hold of this array of `dec_item_t`, you can very quickly work on your high level decoding without worrying about the details of low level decoding. Another big advantage of this technique, is that your high-level decoder benefits from all the evolutions and bug corrections that are made over the time to the low level decoder. This has one downside though: you may need - from time to time - to adjust your high level decoder script if some modifications are made to the low level script that are not backward compatible with older versions.

Please note that a sub-decoder may only create decoder items. Any attempts from a sub decoder to add items to packet view or hex view will be silently ignored without generating errors or warnings.

Also when a sub-decoder adds content to a decoder item, only the first version of that content is taken into consideration, e.g.:

```
ScanaStudio.dec_item_add_content("ACKNOWLEDGE"); //Only this line will
    be considered when pre-decoding
ScanaStudio.dec_item_add_content("ACK"); //Will be ignored when pre-
    decoding
ScanaStudio.dec_item_add_content("A"); //Will be ignored when pre-
    decoding
```

A decoder script can detect if it's called directly from ScanaStudio, or if it's called by another script using the function `is_pre_decoding()`. This can be used to adapt the output of a decoder to be easily used by another script.

ScanaStudio.pre_decode("dec_name",resume)

Description: This function calls the `on_decode_signals(resume)` function in the decoder `dec_name`

Parameters:

- "dec_name" : The name of the script where the `on_decode_signals` function should be called. This name must include the extension *.js, for example: "i2c.js".
- resume : The resume parameter as described in the `on_decode_signals` documentation.

Return value: Returns an array of `dec_item_t` with the newly decoded packets.

Context: Protocol decoder.

Example

Here is an example of implementation of the `pre_decode` function that shows how simple a script can get, provided that you use a sub-decoder to do all the complicated low level processing:

```
function on_decode_signals(resume)
{
  var decoder_items = ScanaStudio.pre_decode("my_low_level_decoder.js",
      resume);
  for (i=0; i < decoder_items.length; i++)
  {
      //Interpret decoder_items[i] and create new decoder items with
          your interpreted data
      ScanaStudio.dec_item_new(decoder_items[i].channel_index,
          decoder_items[i].start_sample_index,decoder_items[i].
          end_sample_index);
      var my_new_data = decoder_items[i].data + 1; //create new data
          based on low level data
      ScanaStudio.dec_item_set_data(my_new_data);
  }
}
```

ScanaStudio.is_pre_decoding()

Description: This function returns true if the script's `on_decode_signals` function is being called by another script, i.e. using the `pre_decode` function.

Context: Protocol decoder

Implementing a sub-decoder in your decoder

In this section, we're going to present a standard methodology that we recommend when one needs to build a decoder based on another (sub) decoder. The process we recommend can be split in several steps:

1. Start by copying the content of the `on_draw_gui_decoder()` function into your own script.
2. Create a `on_decode_signals()` function that simply displays the sub-decoder items without any processing, e.g.:

```
function on_decode_signals(resume)
{

  items = ScanaStudio.pre_decode("sm-bus.js",resume);
  var i;
  for (i = 0; i < items.length; i++)
  {
    ScanaStudio.dec_item_new(items[i].channel_index,items[i].
        start_sample_index,items[i].end_sample_index);
    ScanaStudio.dec_item_add_content(items[i].content);
  }
}
```

3. Test the decoder with some arbitrary signals, and ensure signals are displayed correctly.
4. Start modifying the `on_draw_gui_decoder()` and `on_decode_signals()` functions to implement your high-level decoder that processes the low-level decoder items. When modifying `on_draw_gui_decoder()`, ensure that each GUI item you remove is replaced by a “hidden” GUI value (to ensure the low level decoder has access to all the information needed to operate correctly).

Trigger sequences (FlexiTrig)

All Ikalogic logic analyzers manufactured after 2012 come with an IP (intellectual property) bloc called "FlexiTrig". This is a trigger engine that is unique of its kind, because it allows almost any logic sequence to be described in order to generate a trigger condition when such a sequence is detected. For example, a non acknowledged I2C address call can be used to trigger a capture.

Although recent devices like SP209 series logic analyzers have very deep sampling depths, making this feature less vital, it still can be very useful to generate such precise trigger to synchronize other lab instruments. Indeed, SP209 series logic analyzers have TRIG-OUT SMA connectors that will generate a trigger pulse when a trigger condition is detected by the FlexiTrig engine.

Under the hood, FlexiTrig's trigger sequence works with what we call "trigger steps". Each time the logic signals match with a trigger step, the FlexiTrig engine advances to the next step. This operation continues until the very last step is reached, which generates the actual trigger pulse.

Each trigger step is described with a text string, where each character represents a logic state according to the following table:

Character	Logic state or transition
0	Logic low level
1	Logic high level
R	Rising edge
F	Falling edge
X	Don't care (ignore channel)

For example, a rising edge transition on the second channel of a 4 channels logic analyzer, can be described with this text string: `"XXRX"`. Note that the rightmost character represents the first channel.

Important note: A step description may never contain more than one transition (like a Falling edge or Rising edge). Adding a step with 2 or more edges will generates errors, since it's physically impossible to make 2 edges coincide at the same instant in time.

Here are some valid trigger step examples:

Trigger step text string	Description in plain words
`"001X"`	Low level on channels 3 and 4, high level on channel 2, and don't care on channel 1.

Trigger step text string	Description in plain words
`"1XXXXFXX"`	Falling edge on channel 3, while channel 9 is logic high. Other channels are ignored.

The number of characters in the text string must be equal to the number of channels of a logic analyzer device. Thus, it's the script's responsibility to retrieve the number of channels, and build a correct trigger step string accordingly. The number of channels can be retrieved using the function `ScanaStudio.get_device_channels_count()` which is described in the General functions chapter.

The time between two steps can be constrained using:

- A minimum time
- A maximum time
- A minimum and maximum time

This effectively tells the logic analyzer device to only accept a trigger step match if it happens within defined time limits. (This is essential for triggering on asynchronous protocol words like UART, CAN , 1-Wire or LIN).

Consecutive edges of the same polarity

It is possible to build a trigger sequence with two consecutive edges of the same polarity on the same channel.

One example application for this, is to trigger when a signal reaches a certain frequency:

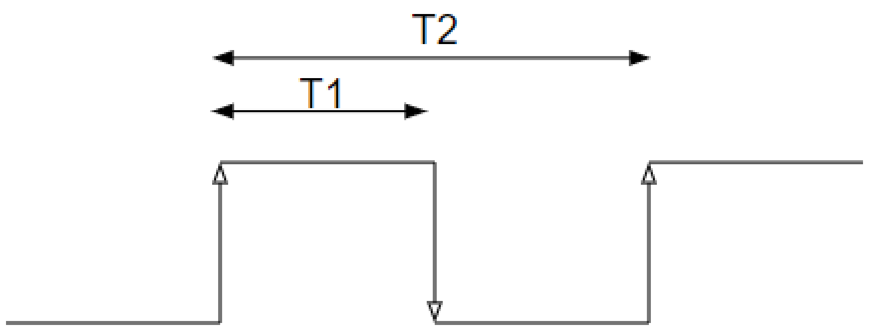


Figure 24: test

In that particular case, we're interested in measuring a signal period (T2), that is, the time between two consecutive rising or falling edges, but the relation between a rising edge and the next falling edge (T1) is not useful.

It's important to understand that FlexiTrig IP translates 2 same polarity consecutive edges into 3 steps:

1. First edge
2. Implicitly added edge of opposite polarity
3. Second edge

If the logic level on the channels is different between the first and the last edge, the (Implicitly) added edge will have `DONT CARE` values. If the logic value is stable, it will be maintained in the added edge.

The example below describing how the Implicitly added edge deals with the other channels, the ones without edges (the 3 MSB channels in that example).

For instance, if the script is creating the trigger sequence below:

```
01XR
11XR
```

It would be translated - inside the FlexiTrig system - into those 3 steps

```
01XR
X1XF <----- Implicitly added falling edge step
11XR
```

In practical terms, the following example simply means that the trigger should happen if there are two consecutive rising edges on channel 0. The logic level on channel 2 must be stable at logic level 1, and the logic level on channel 3 must have the values 0 and 1 at the instant of the first and second rising edge respectively. Channel 3's value may be unstable between the two edges without any consequences on the trigger sequence. Finally, channel 1 is `DONT CARE` (X) during the whole sequence, which means that it's value is ignored.

```
01XR
11XR
```

While this may work in many situations, sometimes, this may not be the required behavior. In this case, it's recommended to fully specify all the steps of a trigger sequence, without skipping any edge.

Entry point function

Like any other feature of ScanaStudio scripts, the trigger script work with entry point functions that are described below:

- `on_draw_gui_trigger()` is called when ScanaStudio need to draw the trigger configuration GUI
- `on_eval_gui_trigger()` is called when ScanaStudio need to evaluate the trigger configuration GUI
- `on_build_trigger()` is called when ScanaStudio need to build or rebuild the trigger sequence using that specific script.

ScanaStudio.flexitrig_append("step_description",t_min,t_max)

Description: This function appends a new trigger step described by `"step_description"` and constrained with the previous step by t_min and t_max.

Parameters:

- "step_description": Text string describing the trigger step
- t_min : Minimum time between this step and the previous step expressed in seconds. For example, if this parameter is set to 0.1, it mean 100ms. This parameter is ignored for the very first appended step. set to -1 to ignore this constrain
- t_max : Maximum time between this step and the previous step expressed in seconds. This parameter is ignored for the very first appended step. set to -1 to ignore this constrain

Context: Trigger

ScanaStudio.flexitrig_print_steps()

Description: This function is simply used for diagnosting and debugging of your trigger steps builder script. Calling this function will simply display a list describing all the trigger steps that have been created by your script, and as they were understood by ScanaStudio.

Steps will be displayed in ScanaStudio's console.

Context: Trigger

Example trigger sequence generator script

The script below shows how to build a trigger sequence in a way that the logic analyzer triggers when a threshold frequency is reached.

Note: A trigger builder script need also to implement protocol decoding features to be used by ScanaStudio. Those functions are not shown in this example for the sake of clarity, but let's assume it's part of a more complete script called "My protocol". Also, prior to using a script based trigger, the corresponding protocol decoder must be added to the workspace. For example, to use the UART trigger, one must first add (and configure) UART trigger in the ScanaStudio workspace.

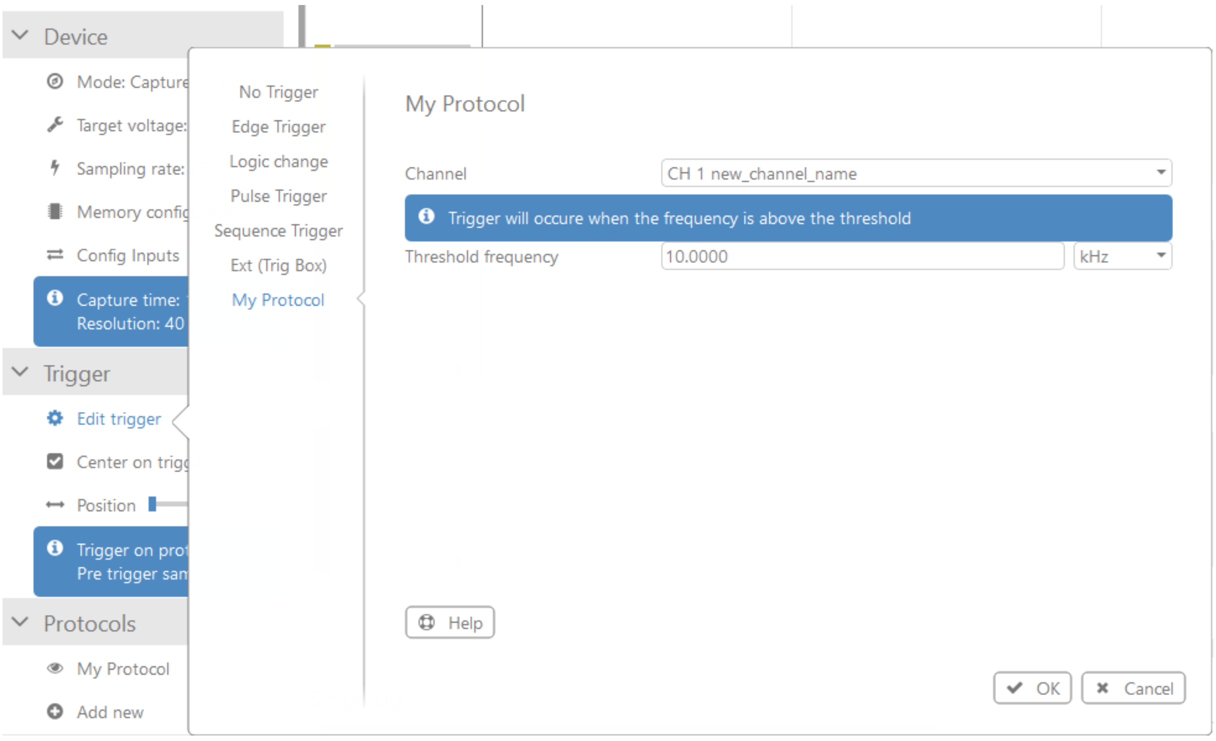
```
//Trigger sequence GUI
function on_draw_gui_trigger()
{
  ScanaStudio.gui_add_ch_selector("f_channel","Channel","");
  ScanaStudio.gui_add_info_label("Trigger will occure when the
      frequency is above the threshold");
  ScanaStudio.gui_add_engineering_form_input_box("f_th","Threshold
      frequency",1,20e6,10e3,"Hz");
}

function on_build_trigger()
{
  var f_channel = ScanaStudio.gui_get_value("f_channel");
  var f_th = ScanaStudio.gui_get_value("f_th");

  ScanaStudio.flexitrig_append(build_trigger_step_string(f_channel)
      ,-1,-1);
  ScanaStudio.flexitrig_append(build_trigger_step_string(f_channel)
      ,-1,1/f_th);
  ScanaStudio.flexitrig_print_steps();
}

function build_trigger_step_string(channel)
{
  var i;
  var ret = "";
  for (i = 0; i < ScanaStudio.get_device_channels_count(); i++)
  {
    if (i == channel)
    {
      ret = "R" + ret;
    }
    else {
      ret = "X" + ret;
    }
  }
  return ret;
}
```

Now, we can use this script (called “My protocol” in this example) in ScanaStudio and add this trigger sequence to a workspace as in the image below:



My Protocol

Channel: CH 1 new_channel_name

Trigger will occure when the frequency is above the threshold

Threshold frequency: 10.0000 kHz

Figure 25: adding trigger to workspace

When you click on **OK** button, the function `on_build_trigger` is called and the trigger sequence is effectively built. Since the script implements a call to `ScanaStudio.flexitrig_print_steps()`, the following printout should be visible in ScanaStudio log (console):

```
[My Protocol Trigger] ** Trigger steps printout, total 2 steps
[My Protocol Trigger] Step index, Description, Tmin, Tmax
[My Protocol Trigger] Step (1): XXXR            N/A      N/A
[My Protocol Trigger] Step (2): XXXR            1.00e-04         N/A
```

This simply shows the two trigger steps created by the script. The two steps define a **R**ising edge on the first channel (channel 0). The first step has no any time constraints on Tmin or Tmax (which is normal, since it’s the first step), the second trigger step though, defines a 0.1 ms maximum time constraint between this step and the previous step. A maximum time of 0.1 ms means a minimum frequency of 10 KHz.

Signal builder

This chapter focuses on the script functions related to building logic signals. Building signals is used for two different purposes:

1. Building demonstration signals (when no device is connected). Demonstration signals are very useful when working on a protocol decoder, and they allow specific and repeatable signals to be generated and decoded without having to rely on specific, hard to find or costly hardware.
2. Building logic patterns to be generated via a logic analyzer devices that support logic pattern generator feature (like the SQ series logic analyzers). This can be useful if one needs to generate special tests patters and study how a specific device reacts to those patterns. It can also be useful to generate serial communication patterns, like a group of UART words or an I2C packet.

Important note: Since signals are built, and then generated at a later time, it is currently impossible the dynamically change the generated patterns on the fly. To achieve such results, one would need to use another class of devices, that can run a firmware, like the Arduino platform.

There are two main approaches when it comes to logic signals builder scripts. The first is to build the signals from scratch, specifying the high and low levels and the number of sample. This is the simplest approach, but it can become tedious if you wish to generate more sophisticated logic patters. The second approach is to use “builder objects” that are exposed by some scripts. For example, the CAN bus protocol script exposes a builder object that encapsulate all the complexities of CAN bus signal building, like CRC calculations and bit stuffing.

Finally, please note that the signal builder script only focuses on building the signals. Device configuration (like which channel is set as input or output), Logic level (3.3V or 5V), and drive type (push-pull or open drain) can only be set by the user in ScanaStudio internal.

Entry point functions

As described above, two possible use cases exist when it comes to signal builder scripts. The first is when a user adds a protocol script to a demo workspace (no actual device connected) and hits the “run” button. At this moment, ScanaStudio will generate arbitrary signals, and then, will check if the protocol script attached to this workspace has a `on_build_demo_signals()` functions and call it.

The second use case, is when a user want to explicitly run the signal builder, which is only possible for devices that support logic pattern generation.

```
function on_build_signals()
{
  // function called when ScanaStudio needs build signals using this
     script
}

function on_build_demo_signals()
{
  // Function called when ScanaStudio needs to build demo signals
}
```

Sampling rate and device memory

When building signals, at the most basic level, we're appending samples, one after the other, in a long memory buffer. Those samples are later played back (generated) to output electrical signals. The sampling frequency (the speed at which those samples are played) will define the time of each sample. The total number of samples, will define the length of the pattern. If the user decides to loop this pattern, then the total number of samples will determine the periodicity at which the pattern repeats itself. Finally, one has also to take into consideration that the device that will generate the signals may have limited embedded memory.

All those parameters are very important and need to be taken into consideration. For that reasons, the following functions were created.

ScanaStudio.builder_get_sample_rate()

Description: This function returns the sampling rate used to playback (generate) the samples.

Context: Signal builder

ScanaStudio.builder_get_maximum_samples_count()

Description: This function returns the maximum number of samples that can be stored in a device's internal memory. If the device has no memory limitation (i.e. streaming the signals directly from the computer), this function will return -1 (at the time this document is written, there are no devices with such unlimited memory).

Context: Signal builder

ScanaStudio.builder_get_samples_acc(channel_index)

Description: This function simply return the total number of samples appended for a specific channel. This function is usually used to stop the script after a channel's memory become full.

Parameters:

- channel_index: the 0-based index of the concerned channel.

Context: Signal builder

Basic signal builder functions

ScanaStudio.builder_add_samples(channel_index,logic_level,samples_count)

Description: This function appends samples to a channel

Parameters:

- channel_index: 0-based index of the channel to which samples should be appended.
- logic_level: set to 0 to append low level samples, 1 for high level samples.
- samples_count: the total number of samples to append (should be greater than 0)

Return value: Returns

Context: Signal builder

ScanaStudio.builder_add_cycles(channel_index,duty_cycle,samples_per_cycle,cycles_count)

Description: This function appends a cycle (one period of a signal) to a channel

Parameters:

- channel_index: 0-based index of the channel to which cycles should be appended.
- duty_cycle: As the name implies, this is a fraction that represents the ratio between the high and low level samples in that cycle.
- samples_per_cycle: The total number of samples in that cycle
- cycles_count: The total number of cycles

Example

The code below will create 10 cycles, with a 50% duty cycle on channel 4 (remember, channel index is 0 based). Each cycle will have 50 samples, 25 of them with a high level, and 25 with a low level.

```
ScanaStudio.builder_add_cycles(3,0.5,50,10);
```

Context: Signal builder

Implementing a builder object

It’s often a good idea to group all the “helper” functions used to build signals for a specific protocol or application in a JavaScript object called `ScanaStudio.BuilderObject`. This has two main advantages:

1. Your script will be neatly organized: all signal building functions will be grouped in one place
2. Your script will be useable by other scripts since it exposes such a “signal builder” object.

This builder object has to have his exact name (`ScanaStudio.BuilderObject`), and thus, there may only be one builder object in a given script. Here is an example implementation of such a builder object:

```

function on_build_signals()
{
  // function called when ScanaStudio needs build signals using this
      script
  var my_builder = ScanaStudio.BuilderObject;

  mu_builder.configure(1,115200);
  my_builder.generate_one_byte(10);
  my_builder.generate_one_byte(20);
  my_builder.generate_one_byte(30);
}

ScanaStudio.BuilderObject = {
  my_ch: 0,
  my_baud : 9600,
    generate_one_byte : function(data_byte)
  {
      // Build the data byte accodring to your protocol
      // ex: ScanaStudio.builder_add_samples(my_ch,1,20)
  },
  configure: function(channel,baud)
  {
    this.my_ch = channel;
    this.my_baud = baud;
  }
};
```

Although this script does not generate any meaningful data, its simplistic approach will help you to understand the usage of the builder object. As you can see, all variables and functions related to signal building are grouped in one object. The ability to instantiate several objects (like the object `my_builder`) gives a lot of flexibility and allows easy code recycling. For instance, another builder object may be instantiated in the `on_build_demo_signals()` entry point function.

Using a builder object from another script

Making use of an existing builder object from another script implies calling the function `ScanaStudio.load_builder_object(script)` to load the builder object. Then, this object can be used as any other JavaScript object.

Let's assume we want to use the builder from the previous example in another script. Let's assume the previous example belongs to a script called "my_script.js":

```
function on_build_signals()
{
  // function called when ScanaStudio needs build signals using this
      script
  var my_builder = ScanaStudio.load_builder_object("my_script.js");
  my_builder.my_ch = 1;
  my_builder.my_baud = 115200;
  my_builder.generate_one_byte(10);
  my_builder.generate_one_byte(20);
  my_builder.generate_one_byte(30);
}
```

As you can see, we only had to change a single line (compared to the previous example), which is the line that loads the builder object. Past this point, the my_builder object behaves exactly as if it was included in your script. As a matter of fact, behind the hood, ScanaStudio will actually appends the builder object to the end of your script before it's executed.

One big advantage of this method is that you may load different builder objects from different files, hence building very sophisticated signals using few, simple, easy to read lines of code.

Putting it all together

Example 1

Suppose we want to write a script that builds a 25% duty cycle square wave, on channel 2 (channel index 1), with a frequency of 100KHz. The example code below shows how this could be achieved.

```
function on_build_signals()
{
  var samples_per_cycle = ScanaStudio.builder_get_sample_rate() / 100e3
      ;
  var maximum_cycles_count = Math.floor(ScanaStudio.
      builder_get_maximum_samples_count() / samples_per_cycle);
  ScanaStudio.builder_add_cycles(1,0.25,samples_per_cycle,
      maximum_cycles_count);
}
```

Example 2

This overly simplified example shows how `BuilderObject` can be used to encapsulate the functions related to servo motor signal generation. The example below may need more development to do a good job controlling a servo motor, but we kept it simple for the purpose of making it easier to follow.

Please note that the `ScanaStudio.BuilderObject` could be replaced by any other object name, it would perfectly work for this example, but it would not allow it to be used by other scripts via the `load_builder_object(script.js)` function. For this reason, it's recommend to use this exact naming convention for the `BuilderObject`.

```
function on_build_signals()
{
  var servo_signal_builder = ScanaStudio.BuilderObject;
  servo_signal_builder.set_channel(2);
  servo_signal_builder.generate_one_cycle(-90);
  servo_signal_builder.generate_one_cycle(0);
  servo_signal_builder.generate_one_cycle(90);
}

ScanaStudio.BuilderObject = {
  servo_ch: 0,
    min_pulse_ms: 1,
    max_pulse_ms: 2,
    min_angle: -90,
    max_angle: +90,

  set_channel : function (ch)
  {
    this.servo_ch = ch;
  },
    generate_one_cycle : function(angle)
  {
    var samples_per_20ms = (20e-3) * ScanaStudio.
        builder_get_sample_rate();
        var angle_ratio  = ((angle-this.min_angle)/(this.max_angle -
            this.min_angle))
        var pulse_width_ms = this.min_pulse_ms + (angle_ratio * (this.
            max_pulse_ms - this.min_pulse_ms));
        var duty_cycle = pulse_width_ms / 20;
        ScanaStudio.builder_add_cycle(this.servo_ch,duty_cycle,
            samples_per_20ms);
  }
};
```

File system functions

This chapter is dedicated to all file system related features, that is, functions that allows a script to access files on the host computer either for writing or reading of data. Please note for security reason, a script cannot read or write to/from *any* file: Only a file path that have been specified by the user in the GUI using file system GUI items can be accessed. For instance, to be able to access a file from a script, there must be one of those two GUI elements:

```
ScanaStudio.gui_add_file_save(id,...)
ScanaStudio.gui_add_file_load(id,...)
```

Please refer to the GUI functions chapter for more details about the usage of those two functions. It's worth reminding though that the `id` parameter of this function is what is needed to access the file set by the user, as it will be explained in details in this chapter. (This confirms that at no time, the scripts gets to know anything about the host's file system or the actual path to the file).

To access a file for reading, first, you need to add a GUI elements in a relevant GUI (like the signal builder GUI). Below is an example:

```
ScanaStudio.gui_add_file_load("my_file_id", "Select CSV file", "*.csv")
    ;
```

which should create a GUI element that looks like this:

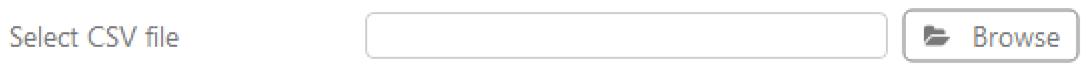


Figure 26: script-import-csv-file

Please note that the text string `"my_file_id"` must be a unique ID in your script's GUI, and is used to open a file.

ScanaStudio.file_system_open("file_id","mode");

Description: This function open a file before being able to read or write to it.

Parameters:

- "file_id": the unique text string ID of the GUI elements used to define the file path.

- “mode”: a single character text string that defines the access mode, according the the table below

Mode	Description
“r”	Read only
“w”	Write only (Deletes existing file content before writing)
“a”	Append (Preserves existing file content)

Return value: In case of error (unable to open the file) this function returns -1. otherwise, this function returns the **file handle** that can later be used for writing and reading.

Context: Global context

ScanaStudio.file_system_close(file_handle)

Description: This function is used to close a file.

Parameters:

- file_handle: The handle that was returned by the function `ScanaStudio.file_system_open()`.

Return value: None.

Context: Global context

ScanaStudio.file_system_read_binary(file_handle)

Description: This function read the content of a file and return the content in binary format

Parameters:

- file_handle: The handle that was returned by the function `ScanaStudio.file_system_open()`.

Return value: Returns an array of integers, each integer represent a byte the file.

Context: Global context

ScanaStudio.file_system_read_text(file_handle,“encoding”)

Description: This function is used to read the content of a file in text format.

Parameters:

- file_handle: The handle that was returned by the function `ScanaStudio.file_system_open()`.
- "encoding": A text string that defines the encoding used to read the file in text format, e.g. "UTF-16". The full list of supported formats is:
  - Big5
  - Big5-HKSCS
  - CP949
  - EUC-JP
  - EUC-KR
  - GB18030
  - HP-ROMAN8
  - IBM 850
  - IBM 866
  - IBM 874
  - ISO 2022-JP
  - ISO 8859-1 to 10
  - ISO 8859-13 to 16
  - Iscii-Bng, Dev, Gjr, Knd, Mlm, Ori, Pnj, Tlg, and Tml
  - KOI8-R
  - KOI8-U
  - Macintosh
  - Shift-JIS
  - TIS-620
  - TSCII
  - UTF-8
  - UTF-16
  - UTF-16BE
  - UTF-16LE
  - UTF-32
  - UTF-32BE
  - UTF-32LE
  - Windows-1250 to 1258

Return value: Returns a text string containing the content of the file

Context: Global context

ScanaStudio.file_system_write_binary(file_handle,data_array)

Description: This function writes binary data to a file (i.e. an array of bytes)

Parameters:

- file_handle: the handle that was returned by the function `ScanaStudio.file_system_open()`.
- data_array: An array containing the bytes that should be written to the file.

Return value: None.

Context: Global context

ScanaStudio.file_system_write_text(file_handle, "text", "encoding")

Description: This function writes text to a file.

Parameters:

- file_handle: the handle that was returned by the function `ScanaStudio.file_system_open()`.
- "text": the text to be written to the file
- "encoding": A text string that defines the encoding used to write to the file in text format, e.g. "UTF-16". The full list of supported formats is:
  - Big5
  - Big5-HKSCS
  - CP949
  - EUC-JP
  - EUC-KR
  - GB18030
  - HP-ROMAN8
  - IBM 850
  - IBM 866
  - IBM 874
  - ISO 2022-JP
  - ISO 8859-1 to 10
  - ISO 8859-13 to 16
  - Iscii-Bng, Dev, Gjr, Knd, Mlm, Ori, Pnj, Tlg, and Tml
  - KOI8-R
  - KOI8-U
  - Macintosh

- Shift-JIS
- TIS-620
- TSCII
- UTF-8
- UTF-16
- UTF-16BE
- UTF-16LE
- UTF-32
- UTF-32BE
- UTF-32LE
- Windows-1250 to 1258

Return value: Returns

Context: Global context

Full example

A good example that demonstrates the file system features is the CSV import script, which takes a CSV file containing some samples, and builds logic signals from those samples. Those samples can be used to generate signals using a compatible signal generator (like the SQ series logic analyzer devices). Below is simplified version of the script:

```

var ENCODING = "UTF-8";

//Signal builder GUI
function on_draw_gui_signal_builder()
{
  ScanaStudio.gui_add_file_load("csv_file","Select CSV file","*.csv");
  ScanaStudio.gui_add_text_input("sep","Column separator",";");
  ScanaStudio.gui_add_new_tab("CSV Mapping",false);
    var ch;
    for (ch = 0; ch < ScanaStudio.get_device_channels_count(); ch++)
    {
      ScanaStudio.gui_add_combo_box("col_ch"+ch,"CH " + (ch+1).toString
          ());
      ScanaStudio.gui_add_item_to_combo_box("Do not import",false);
      for (col = 0; col <= ScanaStudio.get_device_channels_count(); col
          ++)
      {
        ScanaStudio.gui_add_item_to_combo_box("Column "+(col).toString(), ((col == (ch+1))?true:false));
      }
    }
  ScanaStudio.gui_end_tab();
}


//Function called to build siganls (to be generate by capable device)
function on_build_signals()
{
  //Use the function below to get the number of samples to be built
  var samples_to_build = ScanaStudio.builder_get_maximum_samples_count();
  var sample_rate = ScanaStudio.builder_get_sample_rate();
  var max_time = samples_to_build / sample_rate;
  var file = ScanaStudio.file_system_open("csv_file","r");
  if (file < 0) //Is the file successfully opened?
  {
    return;
  }
  var separator = ScanaStudio.gui_get_value("sep");
  var data = ScanaStudio.file_system_read_text(file,ENCODING);;
  var lines = data.match(/[^\r\n]+/g);
  ScanaStudio.file_system_close(file);
  var ch_map = [];
  var ch;
  for (ch = 0; ch < ScanaStudio.get_device_channels_count(); ch++)
  {
    ch_map.push(ScanaStudio.gui_get_value("col_ch"+ch)-1);
  }
  var samples_acc = 0;
  var i = 0;
  for (i=0; i < lines.length; i++)
  {
    var cols = lines[i].split(separator);
    var new_line = "";
    //process one line:
    samples_acc += 1;
    if (samples_acc > samples_to_build)
    {
      break;
    }
    for (ch = 0; ch < ScanaStudio.get_device_channels_count(); ch++)
    {
      if (ch_map[ch] != -1)
      {
        sample_val = parseInt(cols[ch_map[ch]]);
        ScanaStudio.builder_add_samples(ch,sample_val,1);
      }
    }
  }
}
```

General functions

This chapter is dedicated to general purpose functions that are used for general tasks like:

- Script debugging
- Progress reporting (for slow script operations)

Script renaming

It may be of interest to rename an instance of a script to a more unique name. For example, a workspace may have several instances of the UART protocol decoder scripts, each one targeting a different channel. If they all keep their default name, that is: "UART", then it's impossible to know which decoder is for which channel. For that particular reason, a script renaming function is provided so that at any given moment, a script can rename itself to a more "unique" name.

If we keep the example of the multiple UART decoder scripts, one solution would be to call the script renaming function in the GUI evaluation function, which is the moment where we know exactly what channel is targeted by that script.

The different UART decoders could then be renamed:

- UART on CH1
- UART on CH2
- UART on CH3
- etc…

The script renaming function is presented below

ScanaStudio.set_script_instance_name("script_name")

Description: This function renames the instance of the script. This temporary name changing does not affect the hard coded name of the script which is defined via meta-information tags (see chapter 2).

Parameters:

- "script_name": The new script of the name. This name will totally replace the script

Context: Global

Progress reporting

ScanaStudio.report_progress(progress_percentage)

This function provides the user with a progress indication for slow operations (like decoding a very big amount of logic signals). This function should be called periodically (as often as deemed necessary).

Calling the `report_progress()` function multiple times with the same `progress_percentage` value will have no effect (only a `progress_percentage` value different that previous one will be considered). It is highly recommend to implement this function in your script, for each and every entry-point function.

Example:

```
function on_decode_signals(resume)
{
 for (int i = 0; i < total_samples_count; i++)
 {
   ScanaStudio.report_progress(i*100/total_samples_count);

   /*
   Your slow, time consuming decoding code goes here
   */
 }
}
```

Context : Global

Console messages

ScanaStudio.console_info_msg("msg",sample)

Description: This function prints the message `msg` in a console box in ScanaStudio. `sample` parameter is optional. If `sample` is defined, the console message will be displayed as hyperlink linked to the provided sample number. If the user clicks on that hyperlink, ScanaStudio's waveform will center on that specific sample. This function is intended for debugging phase of a script, that being said, it may be used to display messages to the script's end user if it does not fit elsewhere.

Example:

```
ScanaStudio.console_info_msg("A simple message");
ScanaStudio.console_info_msg("My variable =" + variable);
ScanaStudio.console_info_msg("A simple message linked to a sample",
    50000);
```

Context : Global

ScanaStudio.console_warning_msg("msg",sample)

Description: This function is identical in operation to `ScanaStudio.console_info_msg()`. The only difference is the way messages will be presented to the user as a "warning messages".

Context : Global

ScanaStudio.console_error_msg("msg",sample)

Description: This function is identical in operation to `ScanaStudio.console_info_msg()`. The only difference is the way messages will be presented to the user as an "error messages".

Context : Global

Formatting

ScanaStudio.engineering_notation(number,digits)

Description: This function formats a number in engineering notation. For example, `ScanaStudio.engineering_notation("1500",3)` would return "1.50 k"

Parameters:

- `number`: The number to be formatted
- `digits`: The number of digits (Please note that the minimum allowed number of digits is 3).

Return value: Returns text containing formatted number along with engineering prefix.

Context: Global

Protocol decoder scripting methodology

In this chapter, we'll present a methodology that we recommend when writing a script that decodes a specific protocol.

Please note that this is just a recommendation, and we believe this to be an effective way of quickly developing protocol decoder scripts. That being said, depending on the specific protocol you're attempting to write, or the tasks carried by the script, this may not be the best approach.

Obviously, it's mandatory to fully understand how the protocol works and obtain all necessary documentation that describes all timings, waveform, packetization, CRC calculations or whatever specificity for that protocol.

First, we start by including the mandatory entry point function `on_decode_signals` and `on_draw_gui_decoder` so that ScanaStudio can detect this script as a valid signal decoder script. It's okay if those functions are empty at that moment.

According to this methodology, we start by writing the decoder GUI function (`on_draw_gui_decoder`). This is usually a simple straight forward task. One can easily test how the GUI looks like by adding the script to a ScanaStudio workspace.

The next step is to write the signal builder object for that protocol (the `ScanaStudio.BuilderObject` object). The builder object should be a fully independent piece of code, so that it can be included in any other script.

Once the `ScanaStudio.BuilderObject` is written, we can start testing it by writing the demo signal generation function: `on_build_demo_signals`.

As soon as this function is implemented, we can add this script to a demo ScanaStudio workspace (no any device connected) and hit the start button: ScanaStudio will launch the `on_build_demo_signals` function and generate the demo signals.

At this point, the design loop described in the image below can start:

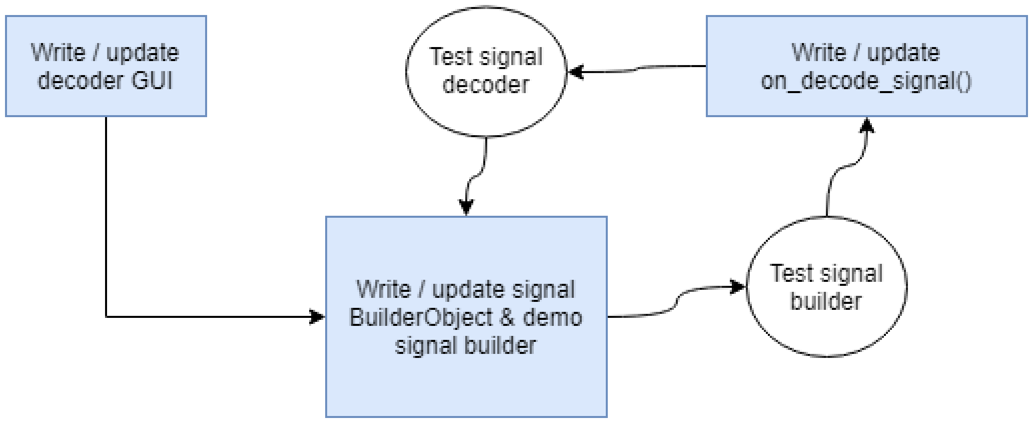

Write / update decoder GUI
Test signal decoder
Write / update on_decode_signal()
Write / update signal BuilderObject & demo signal builder
Test signal builder

Figure 27: script writing methodology

This design loop will help you to quickly converge your script into a fully functional and tested protocol decoder script. You start by writing/testing/debugging the BuilderObject for a particular configuration (that was set in the script GUI), then, the `on_decode_signals` can be written and tested. Sometimes, one may find bugs in the BuilderObject while testing the signal decoder and vice versa.

In other words, the BuilderObject helps you to debug the signal decoder, and the signal decoder helps you to debug the BuilderObject.