High level decoding with ScanaStudio

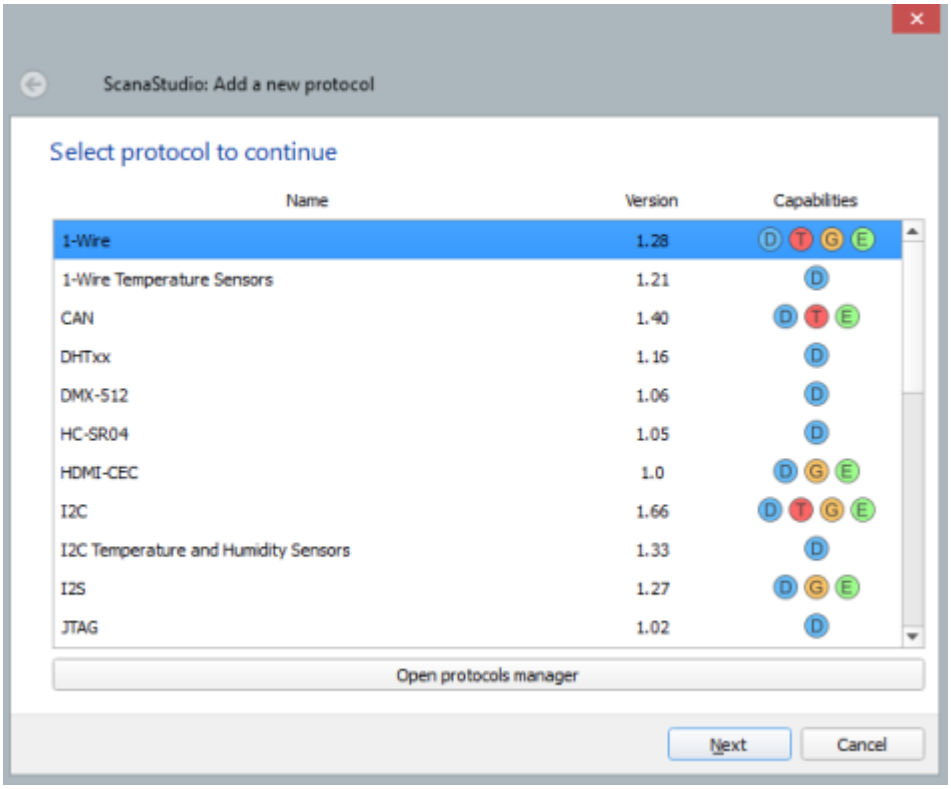
(applies to ScanaStudio V2.4 and above.)

Introduction

ScanaStudio allows users to write their own protocols analyzers. By “Protocol analyzer”, we mean a script that can handle one or more of the following actions:

- Decode a protocol (like I2C, UART or SPI) and display decoded data superposed on the wave forms (example)
- Display decoded data as packets (like WireShark does) or as a HEX dump (as in a regular HEX file viewer).
- Trigger on a specific data content (like a UART Data byte or I2C address)
- Build arbitrary signals that can be used to generate data latter.

The features above are called capabilities. Not all decoders have all those features. As of version 2.4 of ScanaStudio, you can see what capabilities are supported by each decoder, as shown in the image below:



Each capability is symbolized with a letter:

- D : Can decode signals
- T : Can generate trigger sequences
- G : Can generate arbitrary signals
- E : Can generate example signals (also called demo signals). Demo signals are generated when no real logic analyzer device is detected.

Pre-requisites

This tutorial assumes that you are familiar with the following subjects:

- An overview of how ScanaStudio decoding works, specially this chapter, this one and this one too.
- Javascript coding. A nice tutorial for that can be found here, but you may find much more free resources on the web.
- Serial protocols like UART, I2C or SPI.

Aim of this document

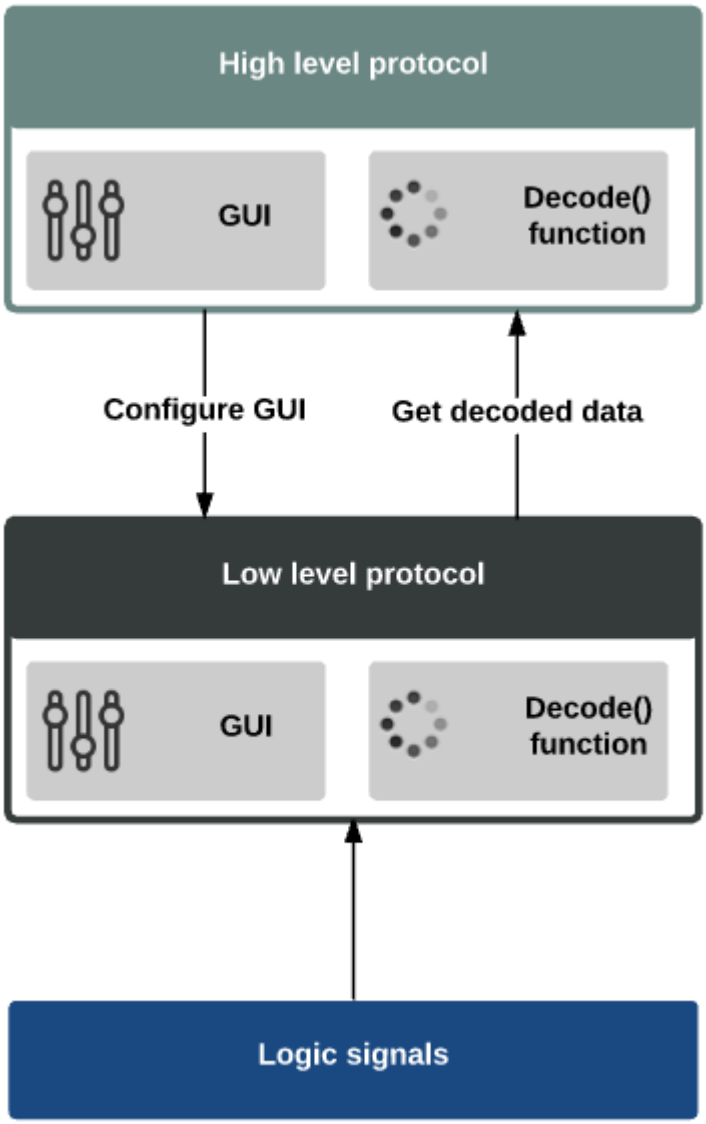
In this document, we're going to concentrate on the High Level decoding, that is, the ability of ScanaStudio to interpret signals that were already decoded, in order to provide more meaningful presentation. For example, Let's assume you have a GSM module controlled with AT commands via a UART interface, and you want ScanaStudio to decode the signals in a way that is meaningful for your application; not just display the HEX value of each byte. In that example, UART decoder is the low level decoder, while another "High Level" decoders, adds a new layer, that is specific to your application, like "GSM module ready" or "Send SMS command".

In this document, we're going to present a practical example of writing a high level decoder based on UART protocol.

How does it work

The idea with Low level and high level protocols, is that one does not have to re-write a low level protocol that already exists. The way this works is illustrated in the diagram below. The high level protocol (the one we're going to write) access an existing low level protocol to retrieve decoded data. This is done in 3 steps:

1. First, the high level protocol needs to inform the low level protocol about configuration options (like BAUD rate, channels to decode, bits per frame, etc..). This configuration is normally made via a graphical user interface (GUI), but since the protocol is in "low level" mode, the GUI is hidden.
2. Second, the low level protocol decodes the signals according to the configuration provided by high level protocol
3. Finally, high level protocol can retrieve decoded data as one big data array.



Practical example: UART

Let's assume you have developed your own communication protocol, that is based on UART. If you need to decode your proprietary protocol, you can write a high level decoder using the following steps:

First, you have to study the GUI function of the low level decoder, to get a hold of all the variables that need to be configured. In case of UART, the GUI fuction looks like that:

```
function gui()  //graphical user interface
{
        ui_clear();  // clean up the User interface before drawing a new one.

        ui_add_ch_selector( "ch", "Channel to decode", "UART" );
        ui_add_baud_selector( "baud", "BAUD rate", 9600 );
        ui_add_txt_combo( "nbits", "Bits per transfer" );
                ui_add_item_to_txt_combo( "5" );
                ui_add_item_to_txt_combo( "6" );
                ui_add_item_to_txt_combo( "7" );
                ui_add_item_to_txt_combo( "8", true );
                ui_add_item_to_txt_combo( "9" );
                ui_add_item_to_txt_combo( "10" );
                ui_add_item_to_txt_combo( "11" );
```

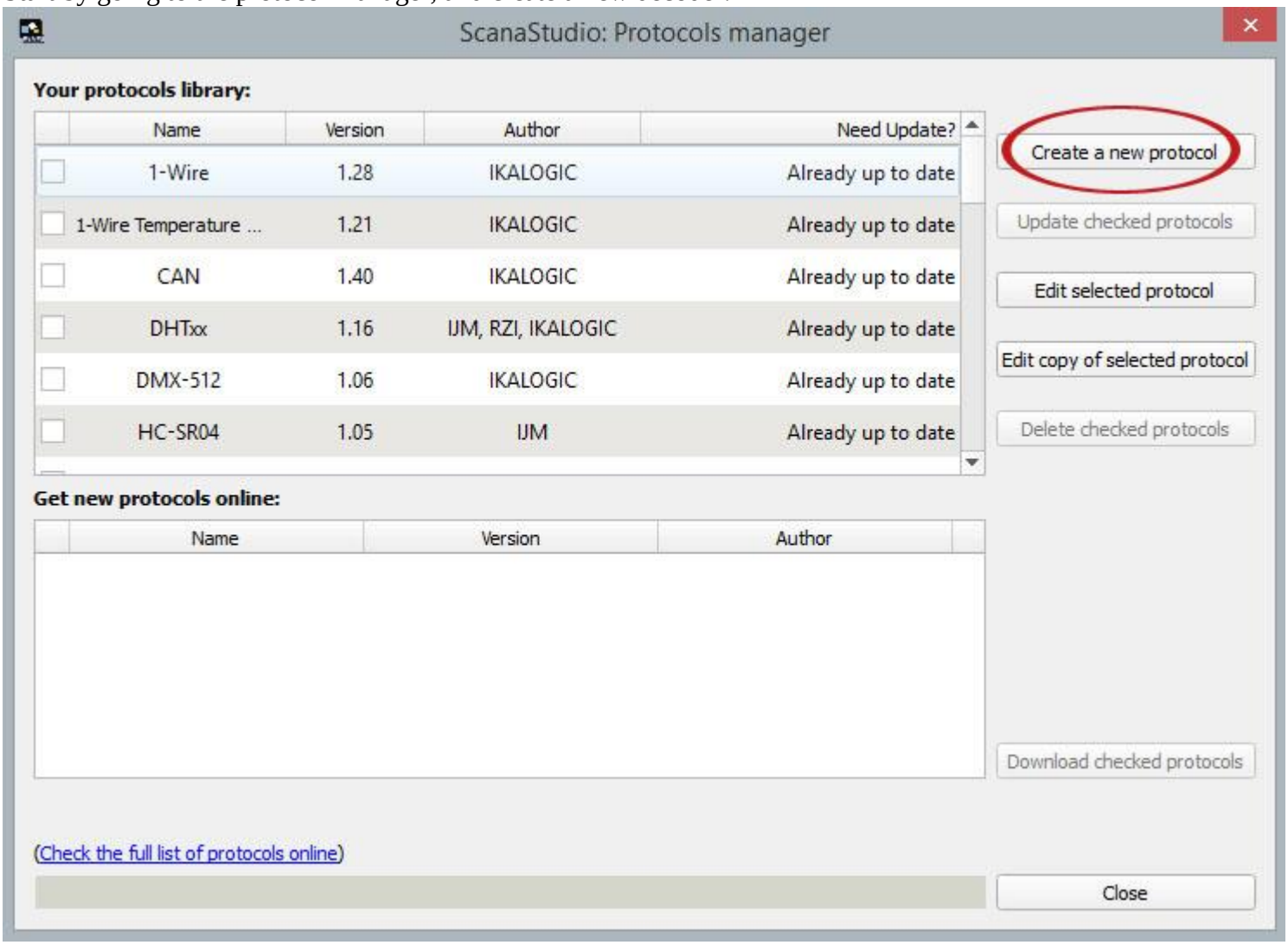
```
                ui_add_item_to_txt_combo( "12" );
                ui_add_item_to_txt_combo( "13" );
                ui_add_item_to_txt_combo( "14" );
                ui_add_item_to_txt_combo( "15" );
                ui_add_item_to_txt_combo( "16" );

        ui_add_txt_combo( "parity", "Parity bit" );
                ui_add_item_to_txt_combo( "No parity bit", true );
                ui_add_item_to_txt_combo( "Odd parity bit" );
                ui_add_item_to_txt_combo( "Even parity bit" );

        ui_add_txt_combo( "stop", "Stop bits bit" );
                ui_add_item_to_txt_combo( "1 stop bit", true );
                ui_add_item_to_txt_combo( "1.5 stop bits" );
                ui_add_item_to_txt_combo( "2 stop bits" );

        ui_add_txt_combo( "order", "Bit order");
                ui_add_item_to_txt_combo( "LSB First", true);
                ui_add_item_to_txt_combo( "MSB First" );

        ui_add_txt_combo( "invert", "Inverted logic" );
                ui_add_item_to_txt_combo( "Non inverted logic (default)", true );
                ui_add_item_to_txt_combo( "Inverted logic: All signals inverted" );
                ui_add_item_to_txt_combo( "Inverted logic: Only data inverted" );
}
```

From this code, we can note that the configuration variables needed by the low level UART decoder are:

- ch: the channel to decode
- baud: the BAUD rate
- nbits: The number of bits per word. NOTE: this is a combo box, where the first element (index “0”) means 5 bits per word. Index “1” means 6 bits per word, and so forth. So if you need to configure the low level uart decoder to 8-bits per word, you need to set this variable to 3.
- parity: 0 = no parity, 1 = odd, 2 = even.
- stop: 0 = 1 stop bits, 1 = 1.5 stop bit, 2) 2 stop bits.
- order: 0 = LSB first, 1 = MSB first.
- invert: 0 = default non inverted.

Now that we know this list of variables we can start writing our high level protocol decoder. You’ll see, it’s not as complicated as it looks!

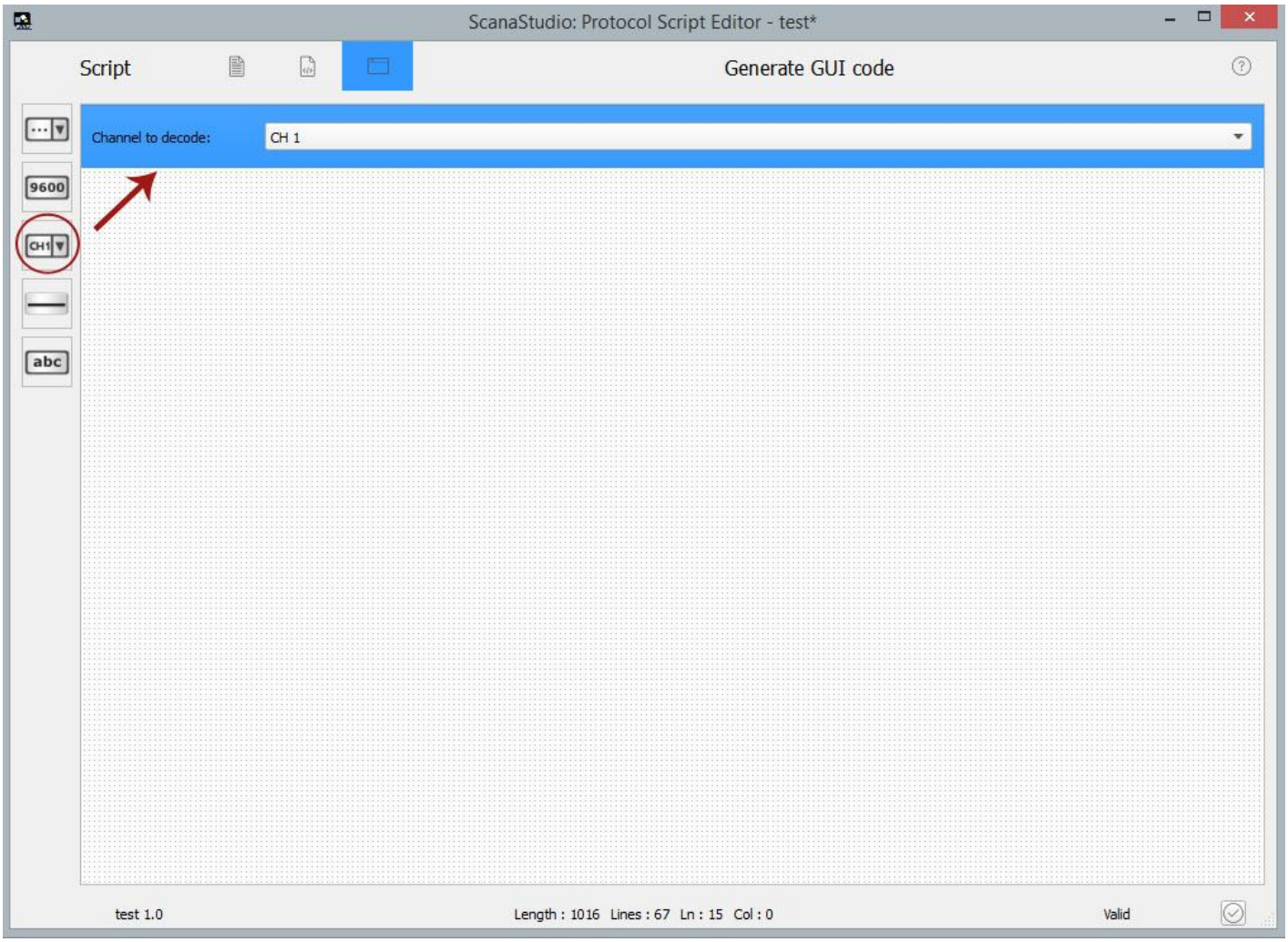
Start by going to the protocol manager, and create a new decoder:



Chose a name like “test”, add a description if you wish, and proudly write your name as the author of this protocol decoder!

Once the new protocol is created, start by saving it (keep the default selected directory unchanged). Then, let’s start by creating our GUI. There’s two ways of doing that, but let’s stick with the easy way, the visual GUI editor. Click on the GUI editor icon as shown below:

You’ll get a tool box to add various types of UI elements, like text boxes and combo boxes that can be used to configure a protocol decoder. For that example, let’s assume that all UART parameters are known (like BAUD rate, Parity and stop bits), and you only want the user to select which channel to decode. To do that, drag-n-drop a “channel selector” from the left tool box, into the GUI area. That’s it!



Move back to the script editor. A way of doing so is by clicking on “Generate GUI code” button. You’ll notice that the gui() function have been populated with a couple lines of code.

Now, to the heart of our subject: Let’s see how to include a low level UART decoder in our high level decoder. It’s done with one function: pre_decode(). You can read more about this function here. This function need to be added in the “decode()” function of your high level decoder. Based on our example, your decode function should look like that at this stage:

```
function decode()
{
        var buffer;
        get_ui_vals();                  // Update the content of user interface variables
        clear_dec_items();              // Clears all the the decoder items and its content
        buffer = pre_decode("uart.js","ch = " + CH_SELECTOR + "; baud = 9600; nbits = 3;
parity = 0; stop = 1; order = 0; invert = 0");
}
```

As you can see, we call “uart.js” decoder (that should already exists on your computer), and pass to it a list of variables that will be interpreted. Among this list, only the channel to decode is really variable in our example. All other parameters are set to constant values.

If you execute this code (you could try), you’ll notice that ScanaStudio takes some time to decode, but nothing gets displayed. That’s normal: the piece of code above decodes the UART content of a channel, and stores all decoded bits and bytes in an array called “buffer”.

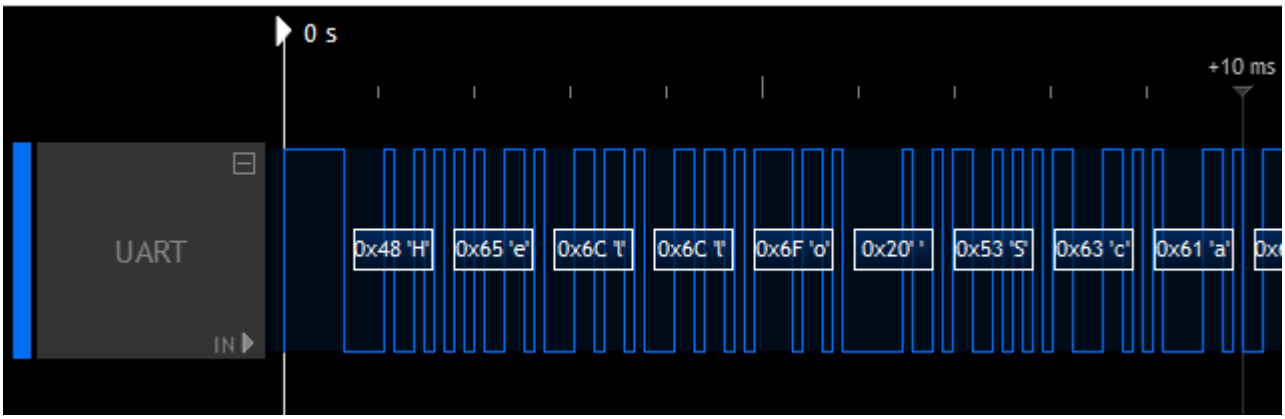
For our simple example, let's assume that the higher level protocol should interpret a "Hello" sequence of character as a start of frame. Here is how the code would look like :

```
function decode()
{
  var i;
  var state = 0;
  var buffer;
  var buffer_temp = new Array();
  get_ui_vals();                // Update the content of user interface variables
  clear_dec_items();            // Clears all the the decoder items and its content

  buffer = pre_decode("uart.js","ch = "+ CH_SELECTOR +"; baud = 9600; nbits = 3; parity = 0;
stop = 1; order = 0; invert = 0");

  // Remove any element that do not contain data, e.g.: Start, Stop
  for (i = 0; i &lt; buffer.length; i++) { if (buffer[i].data.length &gt; 0)
    {
      buffer_temp.push(buffer[i]);
    }
  }

  buffer = buffer_temp;

  for (i = 0; i &lt; buffer.length; i++)
  {
    switch(state)
    {
      case 0: //Search for start of frame sequence (Hello)
        if (i &lt; (buffer.length-5))  //If there's at least 5 bytes
        {

          if (
            (buffer[i].data[0] == "Hello".charCodeAt(0))
            &amp;&amp;(buffer[i+1].data[0] == "Hello".charCodeAt(1))
            &amp;&amp;(buffer[i+2].data[0] == "Hello".charCodeAt(2))
            &amp;&amp;(buffer[i+3].data[0] == "Hello".charCodeAt(3))
            &amp;&amp;(buffer[i+4].data[0] == "Hello".charCodeAt(4))
            )
          {
            dec_item_new(CH_SELECTOR,buffer[i].start_s,buffer[i+4].end_s);
              dec_item_add_pre_text("Start of Frame"); //Maximum zoom
              dec_item_add_pre_text("Start frame");
              dec_item_add_pre_text("SOF");
              dec_item_add_pre_text("S");//Minimum zoom
            state = 1;
          }
        }
      break;
      case 1:
        //To be done...
        state = 0; // Start fetching for start of frame again.
      break;
    }
  }
}
```

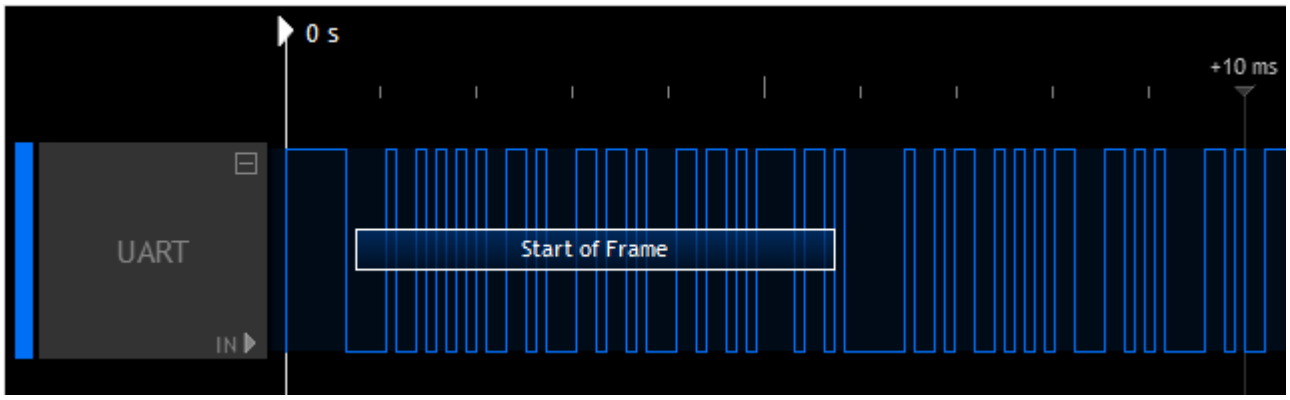
What this decode() function does is the following:

1. Predecode UART signals and store the result into "buffer".
2. Strip the buffer from items that do not contain data (like Start and stop bits)
3. search for "Hello" sequences
4. For each "Hello" found, create a new element named "Start of Frame".

Let's try it out, shall we? Without connecting any device to your computer, create a new ScanaQuad SQ200 workspace, add a UART decoder, and hit the start button. ScanaStudio will run in "demo mode" and generate some UART signals that start with "Hello ScanaStudio tester!…". You can first check the result of the standard (low level) UART decoder. You should see something like that:



Now, add the "test" decoder that you've just created. You should see the "Hello" characters replaced by one "Start of Frame" element:



Conclusion

Obviously, this was just an example, but it shows how easy it is to add a high level decoder based on a lower level decoder. Notice how we didn't have to care about all the details of UART decoding: We don't want you to waste time reinventing the wheel. Instead, we provide the "pre_decode()" function so that you can quickly create a high level decoder that gives meaningful insights into your proprietary decoder.

To get you motivated, here are some things you could do next:

- Create a HEX view of your high level data
- Create an organised hierarchy of packets and sub packets, to be viewed in the packet view