In video #72 we started to make the invisible visible and, as an example, started a small project to enhance an electronic load from China.

At the end of the episode, we still had the following to-do list:

* Build our own microcontroller (using an ESP8266 Wi-Fi chip)
* Build a Nextion display
* Build an I2C slave functionality for the ESP8266
* Understand and emulate a rotary encoder
* Work with fast moving signals and interrupts

So, let’s start: The first point was to build our own microcontroller with an ESP8266. For prototyping, this is not necessary because we can use a readily available NodeMCU board. Later, we can then replace this board with our own design which replaces the display board of our electronic load.

The next thing is to build our Nextion display. If you did not watch videos #56 and #59 this is a good moment, because I will not repeat their content here.

If we work with computer, we always have to design the user interface quite early in the process. The current user interface of the electronic load is quite clumsy. As Voltlog showed in his video, the electronic load I use here has two modes. For today’s video I concentrate on mode 2 which is suitable to test the capacity of battery packs.

How does this mode work with the built-in user-interface? To select a voltage or amperage we have to first select volt or ampere, push the button, then adjust the coarse range, push the button again, and select the fine range. Not very comfortable. I want a simpler interface.

The second flaw in my eyes is the display: During the discharge of the battery, it always displays the current on one display. The other display changes between volt, ampere hours, and watt hours. A display for Watts and also for the lapsed time is not available. And after we stopped the process, we have no possibility to see these numbers again. Not very useful.

So, the new user interface has to overcome all these flaws. It has to replace the rotary encoder, the run button, and display all needed values on one screen.

The screen should look like that: It displays all available values and we can start and stop the load.

If we touch one of the fields which display the “set values”, the screen to set the respective values is shown. And if we touch one of the other fields we get the possibility to reset the counters. I will not discuss the construction of the Nextion file, I just add a link for the download. Just two remarks:

1. If you select the touch press and the touch release event, you might get unexpected reactions
2. To add comfort to the interface I wanted to show the value of the slider in realtime in the display. This is done without the interaction of the ESP, just by programming the Nextion display. Unfortunately, I did not find a possibility to format the output. So, I had to use the trick with the Millivolts and milliamperes. So, I can display whole numbers

After we designed the user interface, we can move on to the next open point: To build an I2C slave for our ESP8266. This mode is currently not supported by the official SDK. So, we have to do a little “bit banging”. This means, we have to replace hardware functionality by software. In this case, we have to do it quite fast because the I2C signals are in the microseconds range.

The second problem is, that these signals can occur any time without any announcement. And our reaction time has to be in microseconds, because otherwise we miss important things. This is why we have to use an interrupt. Interrupts, as their name suggests, interrupt the execution of your sketch and nearly immediately do something different. This is a very interesting concept for many applications, but needs a little thinking:

If you use interrupts, you have to formulate your problem bottom-up. Your normal sketches run in a loop and you have to use while loops to wait for a particular signal. This is straight forward and easy to understand. Interrupt routines, however, do nothing unless triggered by an external signal. This is exactly the contrary of your normal sketch and we have to think different.

Let’s look at our example from the last video: We found two SDA and one SCL signals. If we look at the definition of the I2C signal, we read, that the address and data bits are always defined in relation to the changes of clock signals. If there is not clock, there is no signal transported over this bus. So, it is clear that the SCL signal has to be our interrupt source. And because the data are read at the rising edge of the SCL signal, we attach the interrupt service routine, also called ISR, to the rising edge of the interrupt signal. So, with each rising edge of the SCL signal the ISR is called and we can then read the SDA pin to decide what to do.

The rest is reading and coding. If you are interested, you can look at the subroutine called ISR. It decodes the address and data bits and writes it in an array called value[0..7]. Because these variables are used by the interrupt routine and the normal sketch I declare them as “volatile”.

But wait: In the last video we discovered, that we have two different SDA lines, one for each controller. We could decide to use two different interrupt routines for each channel. I decided to merge these two lines using a digital switch and only use one ISR. I connect the two SDA lines to the respective inputs of one switch. In addition, we have to generate a signal, which switches from one to the other SDA line in the middle of the transmission. This is easier said than done, and I had to heavily use the logic analyzer to get this working. Here, you see the end result: The two separate lines, the switching signal, and the combined signal which is then decoded by the ISR.

I want to use this example to come back to our topic, making the invisible visible. If I for example need to know how long my ISR takes for one go, I have to be very fast. Remember, the ISR interrupts your normal sketch and it has to wait till the ISR is finished. If your ISR would run for a long time, your sketch would not get any time to execute. In the worst case, If the ISR takes longer than the time between interrupts, it would try to interrupt itself and create a crash… So, our ISR has to run faster than the SCL signal. And the time between two rising edges of the SCL is only 7.5 microseconds. This is, why we never, never use Serial.print or delay() in a ISR. Both use milliseconds, which is thousand times slower. But what is the alternative?

We use one pin as our diagnostic pin and switch it low and high at the points where we are interested. To answer the question about the duration of our ISR routine, we switch our diagnosis pin to low at the entry of the ISR and at the exit, we switch it to high again. If we now connect our diagnosis pin to the logic analyzer, we can easily measure the duration, and we see, that the timing is ok. However, I switched my ESP to 160 MHz to get maximum responsiveness.

A very common problem is to know which direction the sketch takes through your different if-then-else statements. I use the example of decoding addresses and data. The ISR has two cases, one for decoding addresses, and one for decoding data.

To find out when the code runs through the address section, we can use the same pin as before and switch it to low and immediately afterwards, to high. Now, we see a short spike if the program passes this particular point.

But if we want to see in which “case statement” our program is? If we just add more spikes, we cannot distinguish between the different statements. Here, I use a trick: I programmed a small subroutine which creates a distinct number of spikes at the diagnosis pin. Like that, we can see immediately, in which area the code is during the decoding of the I2C signal. By the way, here you see the fastest reaction time of the ESP (without going into register programming).

These are a few of my tricks to debug real time systems with simple and inexpensive tools.

So, we have now combined the two SDA signals and we also have a working I2C slave using bit banging. And because the IC used as for switching the two lines is 5 volts resistant, we are also able to shift from the 5-volt logic of the load to the 3.3 volt of the ESP.

If we now connect the Nextion, we already can display the different numbers on our screen.

Now, we can go on to the next hurdle: Emulate a rotary switch. If you search for libraries, you only find libraries to decode the signal of a rotary encoder, not to emulate one. So again, we have to write our own.

In order to emulate something, you have to understand first, how it works. If you google “signals of rotary encoders”, you find these hits: Unfortunately, this is only part of the reality. Again, we have to use the logic analyzer to find the truth. By connecting a real encoder to the oscilloscope, we see the signal created by each turn. It is similar to the ones found with google, but between the movements, both lines are always high. So, we know now what we need and a little later, we have our rotary encoder emulator and we can check first with the oscilloscope and later by connecting it to the electronic load if it works properly. And really, our ESP can now move the numbers in the display.

So, the next open point is done.

These tasks were all easy compared to what comes next: To make all parts work together and together with the (sometimes) strange behavior of the software of the electronic load.

As said earlier, the electronic load has two modes, which only can be changed after reset. If our new interface wants to change the mode, we have to be able to reset the load. Because there is no reset pin on our interface to the display, we have to solder a wire to the reset pin of the load and connect it with a pin of the ESP8266.

If we count now our pins, we discover, that we used them all. 2 for the Nextion display, 3 for the rotary encoder, 4 pins for the two I2C channels, and one each for the run button and the reset line.

So, we have no pins left for the Serial monitor nor for the diagnosis pin. Fortunately, we have an ESP8266 with Wi-Fi capability and I have viewers with good knowledge. Together, I will show you how to build a wireless Serial monitor.

But for today, we already have lots of things to digest. So, lets postpone this part to the next video!

Summarized, we closed the following open items:

* Build our own microcontroller. So far, we only have a prototype using a NodeMCU board. But the proper board is in the mail.
* Build a Nextion display. Here, we have a working prototype.
* Build an I2C slave functionality for the ESP8266. We have one and in addition, we were also able to combine our two SDA signals and shift the levels from 5v to 3.3volt.
* Understand and emulate a rotary encoder. After seeing how it really works this was a piece of cake…
* Work with fast moving signals and interrupts. We successfully did that, and I showed you a few tricks on how to do this. The prove is a working I2C interface

What are the open points for the next episode?

* To get all working together. For debugging, we desperately need a Serial Monitor function which does not need any pins
* We need to be able to read the display values and react accordingly. This will pose us a few challenges, because the programmers of the electronic load did not foresee to be hacked…
* We have to test the whole thing and find also infrequent errors
* We have to refine our skills in setting traps for errors
* We have to put everything into a nice box
* And finally, we have to evaluate if the initial idea was a good one.

I hope, this episode was useful or at least interesting for you. Bye