Introducing Network Programming

Kolbjørn Austreng

i Introduction

A standalone embedded project is undeniably cool - but it is hard to deny that it is even cooler when small embedded solutions connect together over larger networks. There are many ways this could be achieved by adding more custom hardware and complicated bus protocols. Luckily however, we can simply leverage the abundant access to "run of the mill" TCP/IP networks that we find everywhere today.

Today, microcontrollers that already have peripherals for interfacing with "regular" networks over either Eithernet of WiFi, are commonplace and readily purchased. In this project, we are confined to fairly dated hardware that cannot support these protocols directly, but we can always cheat a little. For instance, since the hardware is already connected to a computer over UART, we can simply translate UART messages to whatever format our hearts desire. That is in essence what this assignment will focus on.

Since most of you will not have had courses where you program networked software, all the software required on the computer side will be provided to you. Communication between the computer and the microcontroller must still happen in a predictable fashion, which is why we have created a custom UART format that you must follow. Your task will then be to rewrite your UART driver to make use of the format understood by the computer.

You are not expected to understand all the moving parts of this assignment, but appendix A gives a brief explanation of what is going on behind the scenes. The most important component is by far Elixir - a language that I can heartily recommend for those of you who are going to take the course TTK4145 about real time programming in the upcoming semester.

ii Initial setup of computer side

You can find and download the required computer code at GitHub[1]. The code is mostly written in Elixir, and is easily run with Elixir's build tool `mix`.

What we are aiming for is quite simple; firstly, we want to use Elixir's amazing event handling properties to run a web server locally. Then, we want to demonstrate our mastery over network programming by having that web server serve up a web page for your project. You could of course also run the server in a dedicated hosting facility, which would allow anyone with an internet connection to access your site - but you have to pay for that yourselves :)

Nevertheless, before we can do anything, we must first ensure that the correct dependencies are installed on our system.

ii.a Installing dependencies

Before we can run any Elixir code, we need Elixir itself. Follow the instructions on elixir-lang[2] to see how that is done.

Once you have installed Elixir and downloaded the code from GitHub, you can install the project specific things we need to kick this off. From the root folder of the project (`uart_to_net`), you may call `mix deps.get` to install all the Elixir dependencies. If you are asked to install a local copy of `hex` or `rebar3`, just answer yes.

ii.b Configure the communication interface

The microcontroller will be sending messages over UART to the web server, which will then update the web page it is serving accordingly. To facilitate the communication, you must tell the server where the microcontroller is mounted to the computer.

From the root directory of the downloaded code, navigate to the `config` folder, and open up `dev.exs`. Here you will see the lines

```
config :uart_to_net, interface: "ttyACM0"
config :uart_to_net, speed: 9600
```

If you have a different baud rate, you simply change the `speed` field. Your microcontroller might also be mounted differently, such as to `/dev/ttyS0`. You would in that case replace `"ttyACM0"` with `"ttyS0"` in the config file.

[1]https://github.com/busserull/uart_to_net
[2]https://elixir-lang.org/install.html

ii.c Start the web server

Now that things are set up, we can activate the web server by calling `sudo mix phx.server` from the root of the downloaded code. The first time you do this, it will take some time to compile all the code, but you should eventually be greeted by a message similar to this:

```
[info] Running UTWeb.Endpoint with Cowboy using
↪ http://0.0.0.0:4000
```

> ⚠ You may get a warning about `inotify-tools` if your computer does not have it installed. This is for hot reloading, and you may safely ignore it.

Your server is now ready to accept requests, so open up your favorite web browser - as long as it is not Internet Explorer - and type in `localhost:4000` in the URL field. You will be supplied with a web page like the one shown in figure 1.

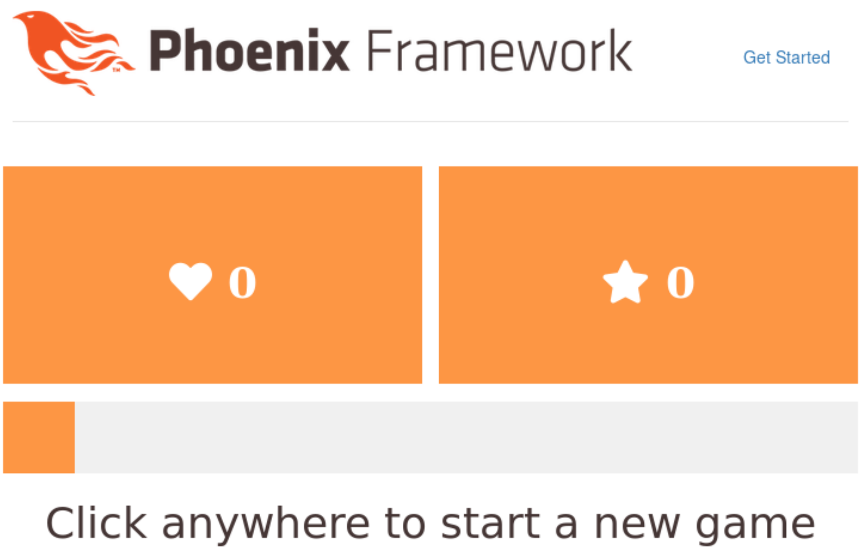

Figure 1: Web page served up by the computer.

Currently, nothing interesting is going on, because the microcontroller is not telling the web server to update its state. Still, if you drag your cursor across the web page, you should see a lot of messages fly across the terminal running the web server - for example:

```
[debug] INCOMING "set_position" on "web:controller" to
↪  UTWeb.ControllerChannel
  Transport:  Phoenix.Transports.WebSocket
  Parameters: %{"payload" => 1000}
```

This is the web server trying to send messages to the microcontroller. The point of the assignment is to make the microcontroller understand these messages, and be able to send them back as well.

As you go on, it should not be necessary to restart the web server. The Erlang virtual machine (and by extension Elixir) even supports hot code reloading - so if there are any daring souls among you who would like to modify the server code, you may do so without the hassle of restarting the server each time.

1 Introducing the STXETX format

The microcontroller and web server need a common language. Instead of using the UART to print ASCII to a terminal, we will be using it to send raw data. The format we will use is fairly simple, and is illustrated in figure 2. The name of the format derives from the start- and end bytes; STX for **S**tart of **T**ex**t**, and ETX for **E**nd of **T**ex**t**. The format works as such:

STX	LEN[0]	LEN[1]	Payload[0..LEN]	ETX

Figure 2: The STXETX format.

1. `STX` is defined int the ASCII standard as the byte `0x02`. This signals the beginning of a packet.
2. `LEN[0]` is the most significant byte of a 16 bit unsigned integer, that tells us how many bytes there are in the `Payload` field.
3. `LEN[1]` is the least significant byte of the same 16 bit unsigned integer.
4. The contents of `Payload` will be described later, but for a packet to be accepted as valid, the number of bytes in `Payload` must match the value given by `LEN[0]` and `LEN[1]`.
5. `ETX` signals the end of a packet. If this byte is not read after `LEN` bytes in the `Payload` field, the packet is flagged invalid, and will be dropped. `ETX` is also defined by ASCII, and has the value `0x03`.

The length- and ETX fields provide out format with a basic mechanism for error detection. If you at some point have to make a similar format for more beefy hardware, you could add a cyclic redundancy check to make your format even more robust in terms of error detection.

In this assignment, we have defined six messages that you may put in the `Payload` field to communicate back and forth with the microcontroller. These message types will be described next.

1.1 0x10 - Score message

A message whose first byte is `0x10` is considered a score update message. This type of message must consist of five bytes; the first being `0x10`, and the remaining four describing an unsigned 32 bit number, where the first byte is the most significant.

For example, the payload

```
<< 0x10, 0x00, 0x00, 0x02, 0x31 >>
```

signifies a score of 561 (0x231). This message should be sent by the microcontroller to update the score indicator on the web page.

1.2 0x12 - Position

A message whose first byte is `0x12` is considered a position update message. The payload must be three bytes; the first being `0x12`, and the remaining two must describe an unsigned 16 bit number in the range 0 to 1000 - including the limits.

An example of a valid payload is

```
<< 0x12, 0x02, 0xfe >>
```

which describes a position of 766 - on a scale from 0 to 1000.

This message should be sent by the microcontroller to indicate the actual position of the encoder in your project. This will update the slider that you see on the web page.

1.3 0x14 - Lives

A message whose first byte is `0x14` is considered a message to update the number of lives that you have left. This payload must consist of five bytes; the first being `0x14`, and the remaining four describing an unsigned 32 bit number, where the first byte is the most significant.

An example of such a message is

```
<< 0x14, 0x00, 0x00, 0x00, 0x03 >>
```

which instructs the web server that you have three lives left. The web page will display the message "*Click anywhere to start a new game*" as long as the lives counter shows 0. This message is sent by the microcontroller to the server.

1.4 0x11 - Request new game

This message consists of a single byte, namely `0x11` and is sent by the server under the following conditions:

1. The game is lost, i.e. the lives indicator shows 0.
2. You click anywhere on the web page.

1.5 0x13 - Request position

This message consists of three bytes; the first is `0x13`, while the remaining two describe an unsigned 16 bit number in the range 0 to 1000. It is sent from the web server to the microcontroller whenever the cursor position on the web page changes - even when the game is currently lost.

An example of such a message is

```
<< 0x13, 0x03, 0xe8 >>
```

which describes a request to set the position to 1000 (0x3e8).

1.6 0x15 - Fire

This message also consists of a single byte; `0x15` and is sent by the server under the following conditions:

1. The game is not lost, i.e. the lives indicator shows anything but 0.
2. You click anywhere on the web page.

1.7 Examples of complete STXETX packets

The following example shows an STXETX packet requesting the microcontroller to set the pong slider to the middle of the board:

```
<< 0x02, 0x00, 0x03, 0x13, 0x01, 0xf4, 0x03 >>
```

This example shows an STXETX packet telling the web server to set the lives counter to 300:

```
<< 0x02, 0x00, 0x05, 0x14, 0x00, 0x00, 0x01, 0x2c, 0x03 >>
```

2 Now what?

The rules of the game is completely up to you. You are free to use the indicators on the web page however you see fit. All you have to do is send the correct packets to the server, and interpret the packets it sends back.

When you have successfully sent some packets back and forth and demonstrated that the communication works, you can pat yourself on the back, confident that you now have one foot in the door of network programming.

Also, since you have technically interfaced a microcontroller with a rather complex stack of networking details, you can now tell your future employer that you have experience with "Internet of Things". But please don't brag about this to the student assistants, because they might be liable to beat you with a chair if you say such nonsense.

A A broad overview of the web server

The web server that you use in this assignment is written in Elixir[3]. Elixir is a prettier and more likable incarnation of Erlang[4]. Erlang, in turn, is a language famed for achieving "the nine nines of uptime"; that is 99.9999999% uptime on critical services, notably in the AXD301 ATM switching system.

Apart from being extremely fault tolerant, Erlang is also exceptionally good at generic event handling. This means that Erlang is the ideal language for a wide array of different server applications. The only caveat is that Erlang has terrible error messages, no (real) strings, no (real) structs, and the syntax looks like someone tried to recount the syntax of Haskell while being rather drunk.

Elixir is the silver bullet to solve most of Erlang's blemishes, and allows you to write equally fault tolerant services at a fraction of the necessary masochism. Still, should you come over anything that Elixir does not directly support, you can simply call Erlang code directly from Elixir. If you inspect the server code, you will come across statements like

```
<< type :: binary-size(1), rest :: binary >> [...]
:binary.decode_unsigned(type) [...]
```

This is Erlang features used directly in Elixir. This ability also allows us to use a battle-hardened web server base written in Erlang, called `Cowboy`. Cowboy, in turn, is neatly wrapped up for us in the Phoenix Framework[5] for web development. This is also the stuff the queue system for the Real time lab is written in.

In essence, this is all there is to our web server itself. It serves up a web page on request, while also hosting another Elixir process that simply translates UART to TCP/IP and back.

On the client side, that is, on the web page once it has been loaded by your web browser, JavaScript takes over. JavaScript maintains communication with our web server via `Phoenix Channels`, that in turn lie on top of the popular `WebSocket` standard (RFC 6455).

This gives us an interesting looking pipeline of `UART (C/Elixir)` -> `TCP/IP (Elixir)` -> `WebSocket (Elixir/JavaScript)` -> `Channel (Elixir/JavaScript)` -> `Web page (HTML/CSS/JavaScript)` and back, all in real time.

[3]https://elixir-lang.org/
[4]http://www.erlang.org/
[5]https://phoenixframework.org/