Overview of the Problem by Joerg Straube

Joerg <joerg.straube@iaeth.ch> wrote on Sept/25: The task of an OS is to provide a rather universal API of the Wifi functionality to the upper client layers. So, the API of the Oberon System to provide WiFi should be carefully crafted. „reduce it to the max to be useful“

The task of the driver SW is to map this general, universal, OS Wifi API to the chip‘s specifics.

You can not assume that the different chips offer the same low layer interface.

As an example: For the TCP/IP functionality in the Unix OS the „sockets“ API seems to be a common ground. Whenever there is a new Ethernet/IP chip, the chip manufacturer often provides his „sockets“ implementation to ease the OS integration.

As the Oberon system is NOT Unix, we have two tasks:
1) invent a Wifi API for the upper layers
2) map this API to the chosen chip’s lower layer

Use Example #1: AX88180 by ASIX www.asix.com.tw

AX88180 is an Non-PCI 32-bit 10/100/1000M Gigabit Ethernet Controller with a 32-bit SRAM-like interface. For the host CPU it is either a piece of addressable RAM, or a FIFO. (It supports both addressing modes.) It is a traditional MAC chip, with all the traditional features like CRC calculation, autonegotiation, etc. It does not provide the on-chip TCP stack. So it is an old style "dumb" interface chip. (Pardon ASIX.) It does not interpret the data buffers. It just passes them both the Tx and Rx ways.

All the SW is implemented on the host. The low-level driver is responsible for setting up and controlling the interface: duplex vs. half duplex, wire speed, enabling the CRC, enabling jumbo packets, etc. The driver is also setting up the PHY connected to the wire side of the MAC.

The SW driver is taking care of the "chip specifics". The upper layers do not deal with the specific chip. They delegate the specifics to the driver. For those layers, all the drivers and all chips look the same. The upper layers deal with the content of the buffers, but not with the specifics of sending or receiving those buffers.

This scheme provides the chip independence because, lets be honest, all MAC chips are similarly dumb. The chips do not even attempt to help in protocol handling. All the protocol handling is implemented in the same upper level SW.

This architecture has a very important benefit: There is only one instance (per operating system) of the protocol SW. It can be thoroughly debugged and optimized. There is no danger that the "helpful driver" will screw the protocol, because the driver is not even trying to help.

There is also a drawback: Protocol handling is very complex. Protocol SW is taking memory and also burning lots of CPU power. Development of this SW stack requires lots of work. In practice, all this work comes from the OS developers. Linux or FreeRTOS are well funded and provide the ready-to-use protocol SW to the application developers.

This comfortable situation breaks down in case of the Oberon OS, or other niche OS's, where the core developement labor is scarce and hardly available.

Use Example #2: Wiznet W5300 & W5500, and some WiFi chips

The niche OS's are helped by shifting all this labor intense and complex SW away from the OS and into the interface chips. Both Wiznet chips are good examples. (See other pages for more such examples in the WiFi domain.) Internally, both chips are running some sort of OS which is not specified. (Some WiFi chips run FreeRTOS, some other run LWIP.)

So now you get the encapsulated TCP/IP stack and you do not need to develop or maintain this SW yourself. You only need to "talk to the chip". This is great as long as you talk to one kind of chip. It would be great if all chips offered the same interface. But they do not.

The interfaces of W5300 and W5500 are not the same. But they are sufficiently similar so they can likely be handled by their drivers. Internally, their TCP/IP engines come from the same vendor, so we expect them being highly similar. The remaining details can be then encapsulated and hidden in the drivers.

There are a few similar chips in the WiFi domain. For example, the TI SimpleLink Wi-Fi CC3135MOD, or MicroChip ATWINC15x0. All these chips use BSD socket interface. It means that the chip specifics can be hidden inside the driver. (Setting up the link parameters, for example.) The buffers exchanged with the chip will not depend on these specifics. So the architecture described in the left column can be realized with these chips.

The modules mentioned above encapsulate a great portion of the SW which previously was running on the host CPU. The trick of using those chips is that their interfaces look similar to each other, so they hopefully can be wrapped into a common host driver.

Use Example #3: A problem with Helpful ASCII Protocols

Some other modules go a step further. They provide an ASCII command interface: (1) the AT commands, (2) the BGScript, or (3) LANCIS. The official blurb is "user friendly" and "easy". We should not be mislead by this. "User friendly" means less effort up front to achieve initial connections. (See [7] in the WRL-13678 section.) However, while simple applications will be easy to achieve, building any sophisticated application with AT commands will likely become a huge mess. "Simple result really fast" is the quintesscence of "hobby quality".

Wrapping the AT commands into a driver is of course possible. We can imagine that the driver will provide BSD sockets to the user, while using the AT commands with the WiFi module. (Or any other ASCII script which the module is providing.) However, one has to doubt the point. The WRL-13678 was built around ESP8266, which can also implement other internal SW, with or without an RTOS. So perhaps a better approach is to invest time up front in ESP8266 using its SDK, and then work with the BSD socket interface.

Reprogramming WRL-13678 is barely possible because the module does not provide enough pins for any serious interface. If one really wants to use the ESP8266, then one should rather take the WROOM module which provides access to more pins. WROOM will not directly plug into RiskZero, but it can be put on an expansion board and plugged into the expansion connector.

Conclusion and recommendations

RiskZero is not really meant for heavy duty WiFi development. The light duty "hobby quality" WiFi can be pursued with the WRL-13678 which plugs into the provided socket. The mid-duty can be pursued with a more serious module on the expansion board. A more powerful motherboard will be needed for heavy duty networking.