This is a technical update primarily targeted at people who would like to know ESP8266 deeper and/or contribute to MicroPython development. You don't need to know anything like that to just use MicroPython on ESP8266, so feel free to skip this update if it sounds too technical.

ESP8266 is one of the many chips produced by Chinese startup companies. About majority of those chips we haven't even heard, and few getting attention of hardware bloggers and electronics enthusiasts oftentimes end up being a fake silver - with no documentation, no toolchain, no examples, oftentimes with no availability. Why does this happen? Silicon chip market is a crowded place. There're usually established vendors of general-purpose chips with great documentation and support, and a pricepoint to uphold such support. Usually, how a startup (no matter from which country) can get onto the market is by making a special-purpose chip for vertical markets - just for lighting products, or just for remote controls, or just for a particular car block, etc. What matters in these segments is not complete documentation for each detail of a chip, but time-to-market, i.e. how quickly you can make a ready product. So, instead of a complete reference manual, such companies provide "reference designs" of schematics and firmware, and documentation enough to just understand the high-level operation and how to do basic customizations. The toolchain, if provided, is usually licensed from a 3rd party and thus not freely redistributable. All this "reference design" goes directly to a contracted customer (who oftentimes signs an NDA), and to the general public goes just a 1-2 pages product brief. Of course, not working on a complete reference manual and not answering detailed customer questions allows to save costs, thus vertical-market solution may be cheaper and compete with solutions based on general-purpose chips. That's why this model is viable.

ESP8266 without a doubt was conceived as such vertical-market solution. In its original appearance, it had a simple serial "AT" commands interface, to quickly integrate with existing systems. Soon it became known that the chip is actually programmable, but with a proprietary API, which was supposedly designed to make simple connected products (like lights) quickly. Both "AT" and API interfaces were quite limited. Initially, there was no publicly available toolchain. The only documentation was leaked, in Chinese. So far, a usual start for hundreds of other chips. So, how did ESP8266 become what it did?

To answer this question, we need to retrace the steps of initial ESP8266 researchers and hackers, and see what they saw.

First of all, it was discovered that ESP8266 uses a CPU of Xtensa architecture. Some folks heard about it, but very few had experience with it. Turned out, Xtensa is another vertical market business. More specifically, it's an extensible CPU architecture targeted at producers of vertical-market chips. You got the picture - they sell those chips under NDA to their customers, and those sell black-box products to us, that's why very few have heard of Xtensa. But fortunately, the companies behind Xtensa (Tensilica, later acquired by Cadence) were pretty technically open about the architecture, and a lot of high-class reference material is available.

There is also GCC, Linux, etc. support for this architecture. However, it's called Xtensa for a reason - there're wide array of CPU models in the family, and one which can run Linux is quite different in capabilities from the one in ESP8266. Actually, ESP8266 has the LX106 CPU model, and that's one of the smallest and simplest of Xtensa. Which is of course another cost-saving measure. It's pure luck that among early ESP8266 researchers, there was Max Filippov, gcc-xtensa maintainer. At that time, GCC didn't have support for LX106's subarchitecture, and he was quick to implement it. If I was to name a single event which contributed the most to ESP8266's success in open-source community, and single person to enable it, it would be GCC support by Max - without his work, many projects simply wouldn't exist (this one for sure), and the rest would deal with shady business of illegally acquired toolchain.

But it's not only that we quickly got an open-source toolchain for ESP8266. Study of the ESP8266 SDK (which was largely provided as precompiled libraries) showed that its higher levels are largely based on well-known Open Source projects in the area of networking and WiFi:

- As a TCP/IP stack, lwIP is used, initially developed within context of Contiki OS http://www.contiki-os.org/ (the pioneer RTOS of IoT), and now maintained at http://savannah.nongnu.org/ and reused in great number of open-source and vendor projects.
- SSL/TLS is handled by axTLS, one of the smallest TLS implementations out there. http://axtls.sourceforge.net/
- 802.11 WiFi stack is net80211/ieee80211 stack from FreeBSD http://www.unix.com/man-page/
- 802.11 security (WPA) is handled by wpa_supplicant. http://w1.fi/wpa_supplicant

The picture was clear now - ESP8266 wasn't just a random WiFi chip like dozen(s) of others. It was a chip largely (perhaps around 75%) based on Open Source software. A triumph of Open Source, if you let. Instead of developing complete firmware of unknown quality in-house, instead of licensing proprietary products from other vendors (still oftentimes of questionable quality), Espressif Systems decided to reuse well-known open projects. Results? That immediately established some quality baseline, as even older (by now) versions used on ESP8266 underwent good deal of community testing. It also allowed to largely save development costs, and output a highly competitive product.

Not everything was so bright though. The Open Source projects listed above are not a random selection. They are actually carefully selected to have one common trait - they are liberally licensed projects (MIT or BSD license). This license (unlike GPL) allows to take Open Source code, modify it (e.g. for particular product) and NOT release modified source back to the community. That's how we get binary blobs in ESP8266 SDK - fully legal on side of Espressif Systems. Still, this design is miles ahead of a typical WiFi product design. Early hackers and adopters of ESP8266 fell in love with it because of this design. They knew that if they dig into ESP8266, it wouldn't be in vain. If they dig deep enough, they'll figure out how originally Open Source components fit together on ESP8266, how to enable in them features not originally enabled (IPv6 anyone?), how to upgrade to newer versions, and how to replace some components with more featureful and secure versions.

That's how ESP8266 became what it did.

Was everything only sunshine and no clouds? Not really. Let's continue our expedition down the ESP8266 core, as here we're approaching the topic of how MicroPython is going to advance the state of the ESP8266 art.

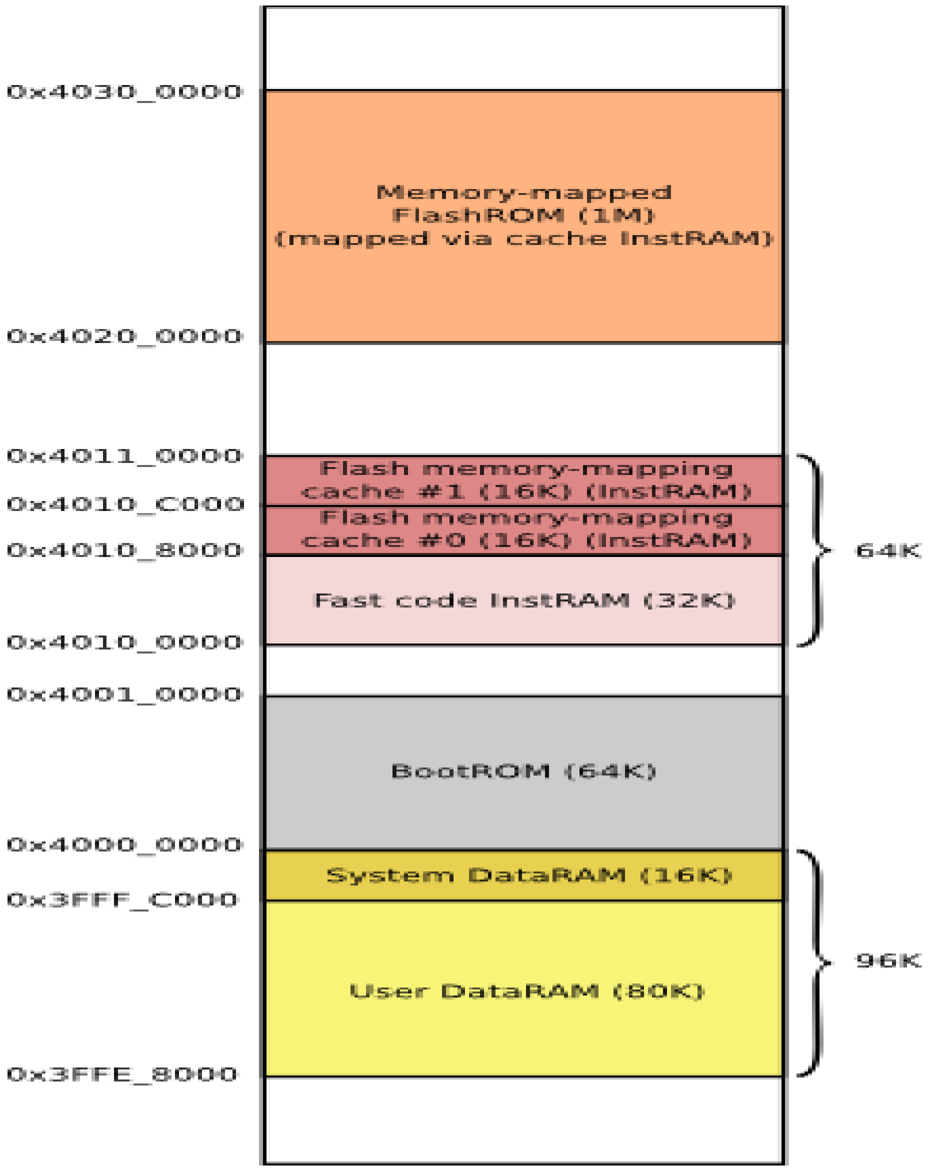
So, what else did early adopters see? The chip didn't contain any internal user-programmer FlashROM. That's certainly yet another smart cost-saving step: putting a lot of internal Flash would make the chip expensive, and why put little, if it still would need to be extended externally? There're low-cost, low pin-count (i.e. can make a low-cost PCB too) FlashROM solutions, and that's what is used with ESP8266. Low pin-count means serial bus though, and that means such external Flash is much slower than internal Flash. We'll see below how that is addressed in ESP8266. There's another drawback of external Flash - for commercial customers it lacks "IP protection" schemes typically found in MCUs with internal flash. But what a bliss an external flash is for open-source users! It means we can remove proprietary blob firmware which is usually shipped with ESP8266 modules and replace it with our, open firmware! And external flash makes ESP8266 modules virtually unbrickable - you can always reflash it with new firmware, and the process is the same all the time. (Of a related note, the upcoming ESP32 retains external flash architecture, but, based on the initial docs, provides some "lock down" features. It's our hope that these features won't be abused, but it's a fact that ESP32 may be less open than ESP8266.)

With no internal FlashROM, early hackers discovered that there's however 64K non-reprogrammable ROM. It contains a bootloader to load firmware from external FlashROM, and reprogram it. It also contains a simple RTOS (real-time OS) which ESP8266 runs. Finally, there's a library of various support routines. We'll cover it later.

Most of modern architectures nowadays provide von-Neuman architecture, where various types of RAMs and ROMs are mapped into common address space, convenient for the programmer (and also various tools like compilers). This is made easily possible on 32-bit architectures due to the vast address space. However, internally, most systems are Harvard architectures, with clearly separate "data" and "instruction" storage, for maximum performance. For Xtensa LX106, a very simple, low-cost CPU, distinction between data RAM/ROM and instruction RAM/ROM are important to account for, as there're noticeable differences on how they can be used, even though they are mapped into a common address space.

With this prelude, early ESP8266 hackers, comparing the published specs and what they actually saw with their debugging tools, figured out that there was 96KB (kilobytes) of DataRAM. That's not a power of 2, but otherwise is a sum of: 64+32. However, 16KB of that are used as "sytems RAM block" - to host various RTOS data structures and stack. That's where the usual figure of "80KB user-usable RAM" comes from.

With instruction RAM, the situation was less clear. Per the linker map, there was 32KB of it. But immediately after it, people saw another 32KB block, which looked like some kind of RAM too. And additionally, there was an open question of how code is executed out of FlashROM. More specifically, it was found out that one small segment of code is loaded from FlashROM to 32KB InstRAM by the bootloader, but largest code segment still resides in the FlashROM directly. Given that code there is given addresses in standard 32-bit address space, it was obvious that FlashROM contents are somehow memory-mapped. There's no documentation on this part of ESP8266 workings, so only the high-level picture is understood, but after extensive investigation and experimentation it became clear that ESP8266 contains single 64KB block of InstRAM. However, upper half of it can be used as 2 independent 16KB blocks of cache for Flash memory-mapping. Usually, both blocks are enabled to maximize Flash code execution performance, but there're means to enable only 1, or disable it completely. It was also found out that a maximum 1MB of contiguous FlashROM space can be mapped, though location of this 1MB window into Flash is adjustable.



We have now finished our enumeration of the ROM and RAM resources that an ESP8266 has.

Let's make a summary table with sizes and data/instruction types:

•BootROM -- 64KB -- InstROM

•Data RAM -- 96KB -- DataRAM

•Instruction RAM -- 32KB -- InstRAM

•Flash cache -- 32KB -- InstRAM

•Flash code -- 1MB -- Mapped via InstRAM cache

Why is instruction/data memory type important? Let me start by saying that many RISC processors due to their simple architecture and low cost have memory access alignment restriction. It works like this: you usually may access an N-byte unit (where N is a small power of 2, usually up to bitness of CPU), only by the address divisible by N. E.g., if you want to access 32-bit (4 bytes) value, it must be done by address divisible by 4. That's how MIPS works. Of popular ARM Cortex CPUs, simple Cotrex-M0 has such restriction, but more complex Cortex-M3 and Cortex-M4 has configurable support for unaligned access. But unaligned access leads to reduced performance, so it's better to avoid it nonetheless. Anyway, the point is that such alignment restriction is pretty standard and well-known, and e.g. compilers for RISC layout data with suitable alignment automatically. Xtensa also has this restriction, for DataRAM. However, for instruction memory on LX106, there's an even stricter restriction - it can be accessed *only* as 32-bit, aligned words. This actually may come as surprise, because Xtensa instructions are usually either 2 or 3 bytes in size (other RISC CPUs usually have 4 or 2 byte instructions), but the point is that there's a dedicated instruction block, which takes 32-bit words and decodes instructions out of it. This block is already more complex than other RISC CPUs, and to compensate complexity and keep costs down, instruction RAM subsystems is simplified - it just provides a fixed-size bus with minimal control overhead.

How does this InstRAM restriction affect us, on ESP8266? In a direct, and not very pleasant, way. The matter: any program consists in large part of constant data. If you have in your application a statement like

print("Hello, world!")

The string "Hello, world!" is such a constant, an array of bytes. On most other architectures, it would go directly to (Flash)ROM, which is usually in plenitude. If we do the same on ESP8266 and put it in FlashROM, then the data will be available memory-mapped using the caching mechanism. But remember that the cache is of InstRAM type, and must be access only as 32-bit words. If you will try to access your string byte by byte, as is usually done, you will get an immediate crash. A big showstopper! Trying to solve this problem specifically for ESP8266 on the application level is quite problematic. Most applications are not written for ESP8266, but exist and work on other architectures, and are just ported to ESP8266. They may have gazillion of places where ESP8266-specific assumption are violated, tracking them all will be quite an effort, and then how to fix them is still a problem. Such fixes are not needed for other architectures, and, if applied, lead to performance or size degradation, and making them conditional for ESP8266 would lead to rather dirty and hard-to-maintain code.

Understanding all this, Espressif systems, creators of ESP8266, went for a solution of least resistance: to ensure compatibility with existing code, all constant data are kept not in ROM, but in ... RAM! That's right, constant data which is read-only and belongs to read-only memory is kept in precious RAM, where rather modifiable user data should be kept. Again, it was not a big deal for the original usage ESP8266 was destined for - there was still enough RAM to blink a light with a smartphone. But as we saw above, ESP8266 actually has quite pumpy resources which are capable of much more. But as people start to try doing more, they inevitably hit memory issues due to the layout described above. Let's count together: out of 96K, 16K are split immediately for the "system data block", WiFi and networking require about 20KB of statically allocated (i.e. not reusable for anything else) buffers. These statically allocated buffers aren't all that's required for networking and RTOS, because there are dynamically allocated buffers too. Say, another 20KB (with 10KB you actually risk to get out of system memory). 10KB of constant data isn't too much - a simple single HTML page and detailed user message will get you there easily, more for a couple more pages. There're some other system and application support overheads, and in the end you get just 15-25KB of RAM for your application processing.

The situation like this is part of the myth that you can't do anything useful with ESP8266 unless you program in C. This myth in especially popular in Arduino circles, where it was inherited from the original Arduino which had only 2KB or RAM and 32KB of FlashROM. Certainly there's another way to deal with the situation - overcome it, and that's what our MicroPython on ESP8266 project is about!

Data RAM pressure is by far the biggest problem of ESP8266, but InstRAM is also a limited resource. Let's discuss what it is useful for and whether it requires usage optimization. As was mentioned above, getting information out of serial FlashROM is a slow operation, and thus code execution speed out of it will be affected. For tight loops which were cached, sustained execution speed should be the same as InstRAM, but with real code which jumps around all the time, there may be a noticeable slowdown. Unfortunately, there're no objective, reproducible benchmarks to measure this slowdown, mostly speculative testimonies which suggest there may be a 10x slowdown comparing to InstRAM. So, that's one and the biggest use for InstRAM - when maximum, and predictable, execution speed is required. Another usage is due to the effect of FlashROM caching and interrupts: memory-mapped cache may be disabled at times (e.g. to access FlashROM as data storage), and if interrupt arrives at this time and its handler is in FlashROM, it will crash. So, generally all interrupt handlers should be in InstRAM.

32K of InstRAM seems like not too little, but a large part of it is already filled by code from SDK binary blobs, responsible for highly realtime operations like handling WiFi. Another part is filled with frequently used (and thus performance-critical) routines, like memory copying and arithmetics. Thus, only breadcrumbs are left for user applications. You may remember that at the beginning of our inside journey we discussed the BootROM and that it contains a routine library. Many of the support routines ending up in InstRAM are actually available in BootROM. And there's actually a provision by vendor ESP8266 SDK to use BootROM versions instead of putting a second copy in InstRAM. But due to the way it's coded, it's actually not active by default, so most applications end up with InstRAM copy. We're not sure why it's like that - if it's a mistake or overlooking of fine details, or just deliberate choice to stay on the safe side - after all, there may be bugs in BootROM routines, which were later fixed in a newer SDK, or a user toolchain may be not compatible with BootROM's versions. But by carefully examining functions and verifying the end result (and MicroPython shines here - we have 92% test coverage, growing), there's a way to free up InstRAM to put there performance-critical parts of application code.

High-level DataRAM and InstRAM optimizations are the biggest items in our work list, but there are more specific tasks too, like to see if we can somehow reuse at least some chunks of the system data block, what's located in static data buffers of SDK and whether they can be slimmed down, how to better control dynamic buffers, etc.

This update went long and technical, but we hope you enjoyed it, as it gives good insight into what happens inside your ESP8266 chip, what challenges we face with our project, and our approach of addressing them. There was already a lot of inquisitive research behind this, and more investigation and experimentation will be required to achieve the best, stable results. We are happy that MicroPython users will be the first to enjoy the results of it, but also hope to advance the technical state of ESP8266 community in general, as indeed ESP8266 has become a truly open-source project, growing well beyond a small vertical-market niche it might have been instead.

References:

- https://github.com/jcmvbkbc/
- https://github.com/pfalcon/
- https://github.com/esp8266/
- http://richard.burtons.org/
- https://github.com/pfalcon/