Zbiorczy opis układu SoC ESP8266

Krzysztof Pierczyk

listopad 2020

Rozdział 1

Wstęp

Po pierwsze, w Internecie jest strasznie informacji mieszających różne kwestie związane ze starym i nowym SDK. W związku z tym warto amiętać, że jedyne źródła informacji na jakich można polegać to:

- oficjalna dokumentacja IDF: kwestie interfejsu progamistycznego oraz organizacji pamieci flash (należy pamiętać, że ESP32 i ESP8266 nie są jeszcze w pełni zgodne w tej materii)
- oficjalna dokumentacja SDK (aktualna): mała ilośc informacji, ale pokazuje te drobne różnice między ESP32 i ESP8266
- oficjalne dokumenty Espressif, które obecnie znajdują się na stronie: tylko kwestie dokumentacji technicznej (nie API)
- fanowska wiki: co się da
- nieliczne artykuły, których autorzy podjęli wysiłek przeanalizowania dostepnego jawnie kodu

Pozostałe źródła należy raczej omijać szerokim łukiem, gdyż nawet jeśli informacje w nich zawarte nie są zupełnie zmyślone, to jednak ich utorzy wykazują się często niezrozumieniem kwestii, o których piszą.

1.1 Nowe IDF-SDK

Espressif podjęło w ostatnim czasie tytaniczny trud przepisania SDK dla ESP8266 w taki sposób, aby upodobnić je do przemyślanego i na bieżąco wspieranego IDF (ang. IoT Development Framework) dla ESP32. Młodszy układ jest zniacznie lepiej udokumentowany, a sama platforma od samego początku dobrze przymyślana (w porównaniu do starego SDK dla ESP8266). Chociaż niektóre kwestie związane z nowym SDK (przede wszystkim związane z mechaniką działania, a nie z API) są mocno zakorzenione w dawnym SDK i korzystają ze starego kodu, to jednak może najczęściej przyjąć, że biblioteki działają w sposób niemal identyczny jak w IDF. Drobne różnice wynikają przede wszystkim z różnic w fizycznej strukturze układów, jak np. różne długości MMU mapującego pamięć Flash do przestrzeni adresowej (24 bity w ESP8266 pozwalają mapować jedynie 1MiB w jednym czasie).

Rozdział 2

Specyfikacja techniczna

2.1 Pamięć

2.1.1 RAM

Większość kluczowych informacji zawarta jest w dokumentach dostępnych na stronie Espressif. Jednak jako, że są tam dostepne źródła odnoszące się do starego SDK warto skorygować kwestie związane z pamięcią dostępną w ESP8266. Układ posiada 160KiB statycznej pamięci RAM. Jest on zaprojektowane w architekturze Harvardzkiej w związku z czym dostęp do danych i instrukcji odbywa się poprzez osobne magistrale. Co za tym idzie wbudowany SRAM został podzielony na dwa bloki:

- 96KiB dRAM przechowujący dane programu (m.in. stos i stertę) [0x3FFE8000-0x3fffffff]

  - ostatnie 16KiB jest pamięcią Read-only wykorzystywaną przez ETS (przerwania wewnętrzne układu, np. zegarowe)

- 64KiB iRAM przy czym:

  48KiB iRAM pozwala przechowywać kod programu [0x40100000-0x4010Bfff]

  16KiB cache służy do buforowania kodu umieszczonego w pamięci Flash [0x4010C000-0x4010ffff]

Konfiguracja Ćomponent config —> ESP8266-specific —> Enable full cache mode"pozwala przeznaczyć dolne 16KiB segmentu iRAM na rozszerzenie cache'u dla pamięci flash. Mapa pamięci stworzona na fanowskiej wiki wskazuje, że obszar cache może być sprzętowo całkowicie wyłączony, co pozwala uzyskać 64KiB pamięci iRAM. Taki scenariusz nie jest jednak wspierany przez środowisko IDF-SDK.

2.1.2 Flash

Układ SoC ESP8266 nie posiada wewnętrznej pamięci Flash. Możliwe jest jednak podłączenie do niego zewnętrznej pamięci SPI Flash. Na chwilę obecną IDF-SDK obsługuje do 8MiB takiej pamięci, chociaż pliki nagłówkowe umieszczają definicje do 64MiB dla kompatybilności z oprogramowaniem NodeMCU. Pamięć ta przechowuje wszystkie elementy potrzebne do uruchomienia programu. Sposób jej formatowania przez SDK został opisany w sekcji "Partycjonowanie pamięci Flash".
Sam moduł ESP8266 obsługujedo 16KiB Flash w trybie QSPI. Możliwe jest także dołączenie kolejnego modułu pamięci obsługiwanego przez standardowy protokuł SPI. Dostęp do tej pamięci odbywać się może programowo przez sterownik SPI zaimplementowane w SDK. Układ zawiera jednak również dedykowaną jednostkę MMU, która pozwala zmapować do 1MiB tej pamięci w przestrzeń adresową [0x40200000-0x402fffff]. Domyślnie jest to początkowy 1MiB jednak możliwe jest przemapowanie w trakcie działania programu. W SDK możliwe jest to tylko na etapie Bootloadera 2-go poziomu.

2.1.2.1 esptool

Narzędziem służacym do komunikacji z ESP8266 z poziomu PC jest esptool.py. Wbudowany na stałe w układ bootlaoader (patrz sekcje "Wbudowana pamięć ROMóraz "Botowanie") ma możliwośc komunikowania się przez port szeregowy celem zaprogramowania pamięci Flash. Narzędzie esptool.py (które jest wewnętrznie wykorzystywane przez idf.py) komunikuje się z nim po uruchomieniu w celu wgrania do pamięci programu flasher stub. Jest to dedykowany bootloader o znacznie szerszych możliwościach niż ten tkwiący w wewnętrznej pamięci ROM. Po jego wgraniu następuje reset systemu, uruchomienie stub'a przez wbudowany bootloader i faktyczne programowanie pamięci Flash przeznaczoną do tego aplikacją odbywa się już poprzez komunikację esptool-flasher stub. Dzięki temu możliwe jest nie tylko wgrywanie właściwego programu ale także pozyskiwanie informacji na temat układu SoC i dołączonej pamięci.

2.1.3 Wbudowana pamięć ROM

ESP8266 posiada wbudowany na stałe Bootloader 1-go poziomu, który rezyduje w wewnętrznej pamięci ROM [0x40000000-0x4000ffff] (64KiB). Pamięć ta jest dostępna jedynie w trybie do odczytu.

2.1.4 Pamięć modułu RTC

Układ ESP8266 posiada wbudowany moduł RTC (ang Real Time Clock), który pozwala odmierzać czas w trybach głębokiego uśpienia. Wśród rejestrów RTC znajdują się dwa banki pamięci po 512B każdy. Pierwszy z nich przeznaczony jest na dane systemowe. Drugi, dostepny jest dla użytkownika [0x60001200-0x600013ff]. Dane te przetrwają stany głębokiego uśpienia i będa dostepne od razu do przebudzeniu. Jest to pamięć o znacznie krótszym czasie dostepu niż pamięć Flash.

2.1.5 Kontrola pracy linkera

Domyślny skrypt linkera składający aplikacje dzieli ją na segmenty. W zależności od rodzaju danych (modyfikowalne/niemodyfikowalne, dane, instrukcje) wyznacza im różne adresy docelowe w nagłówku segmentu (patrz sekcja "Partycjonowanie pamięci Flash"). Kod ten jest później kopiowany w miare potrzeb z pamięci Flash do iRAM/dRAM przez bootloader. Kontrola rozlokowanie fragmentów kodu i danych bez ingerencji w plik linkera możliwa jest dzięki odpowiednim makrodefinicjom zawartym w komponencie esp8266 (nagłówek esp_attr.h).
Domyślnie kod umieszczany jest docelowo w zmapowanym do przestrzeni adresowej obszarze pamięci Flash i cache'owany w trakcie dostępu. Zdefiniowanie funkcji z odpowiednim atrybutem sprawia, że linker umieści ją w segmencie, który bootloader przeniesie do pamięci iRAM w trakcie bootup'u. Dane niemodyfikowalne równiez umieszczane są w pamięci Flash. Tutaj również możliwe jest zadeklarowanie ich z atrybutem, który przeniesie je do segmentu iROM. Nalezy pamiętać, że dane umieszczone w pamięci Flash nie są cachoe'owane przy dostępie. Dane modyfikowalne (stos, sterta, zmienne globalne) są zawsze przenoszone/inicjalizowane w dRAM. Możliwe jest jednak zadeklarowanie zich atrybutem dzięki któremu ~~bootloader~~ startup aplikacji powstrzyma się od ich inicjalizowania (zerowania w przypadku zmiennych globalnych/statycznych).Istnieje także atrybut, który pozwala umieścić dane w pamięci modułu RTC.

`DODATEK:` Komponent spi_ram pozwala wykorzystywać pamięć Flash jako obszar modyfikowalnych danych

2.2 WiFi

ESP8266 posiada układ umożlwiający transmitowanie danych z wykorzystaniem protokołów IEEE 802.11b/g/n. Chociaż fizycznie zawiera on jedynie jedno wyjście, to możliwe jednak wirtualnie możliwe jest korzystanie z dwóch interfejsów (dzięki czemu możliwy jest tryb AP+Station).

Rozdział 3

Bootowanie

Wektor resetu procesora znajduje się pod adresem [0x40000080], który to mieście się w obszarze wewnętrznego ROMu. Dzięki temu, po uruchomieniu układ zaczyna wykonywać kod Bootloadera pierwszego poziomu. Ten oto bada stan na odpowiednich pinach chipu i w zależności od wyniku rozpocząć jeden z dwóch (trzech) możliwych scenariuszy:

- bootowanie z pamięci Flash
- komunikacja z zewnętrznym narzędziem przez interfejs UART
- (bootowanie z pamięci SPI SD)(?)

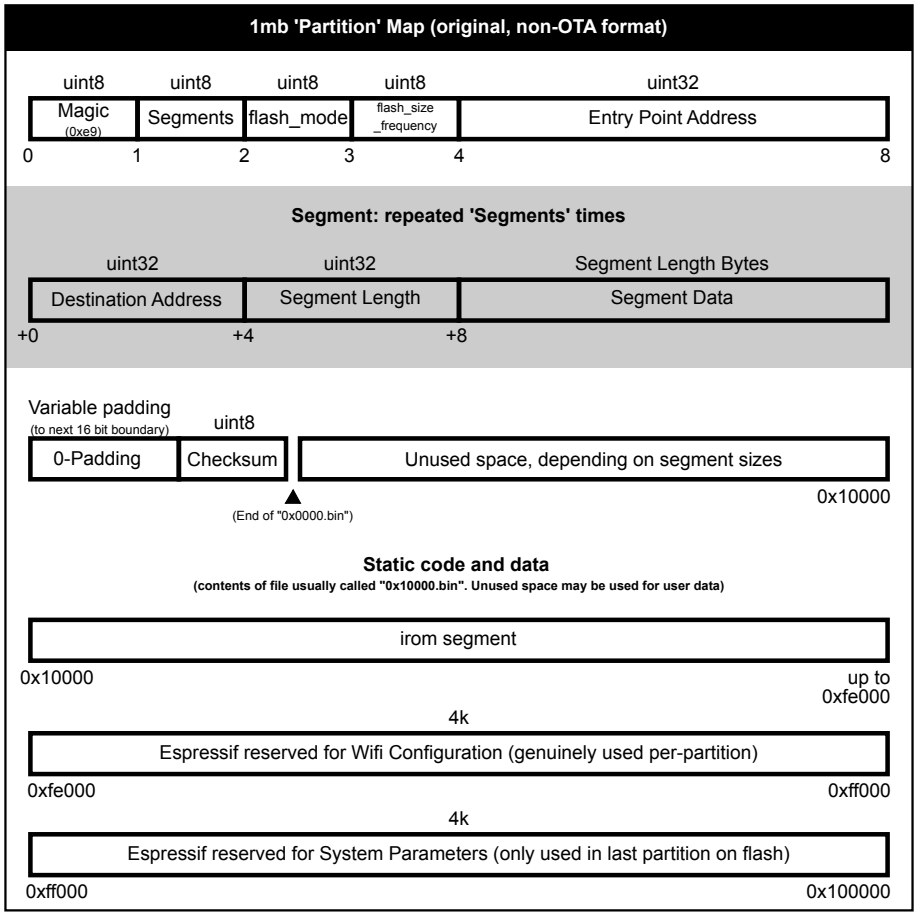
Drugi scenariusz został opisany w częśći ęsptool". Pierwszy polega na odczytaniu zawartoći pamięci Flash, odczytaniu nagłówka obrazu oraz nagłówków segmentów, załadowaniu odpowiednich segmentów do pamięci RAM ~~zainicjalizowaniu segmentu .bss~~ (inicjalizacja odbywa się w startup'ie aplikacji) i przeskoczeniu do punktu wejścieowego (ang. entry point) programu użytkownika. Po bardziej szczegółowe informacje patrz sekcja "Partycjonowanie pamięci Flash".

3.1 Bootloader drugiego poziomu

Środowisko IDF-SDK wraz z każda aplikacją kompiluje także bootloader 2-go poziomu. Bootloader ten jest wgrywany razem z programem jako jedna partycji. Dla ESP32 rozpoczyna on się od adresu [0x00001000] a dla ESP8266 od adresu [0x00000000]. Głównym zadaniem bootloadera jest wgranie aplikacji użytkownika do pamięci z uwzględnieniem segmentów stworzonych przez linkera, ~~zainicjalizowanie danych~~ (inicjalizacja odbywa się w startup'ie aplikacji) oraz wykonanie skoku do punktu wejściowego. Bootloader ten jest jednak dalece bardziej rozbudowany niż ten 1-go poziomu. Przede wszystkim uwzględnia on możliwość istnienia w pamięci wielu aplikacji i uruchomienie tej z nich, która została uwzględniona w odpowiedniej sekcji konfiguracyjnej. Ponadto bootloader ten inicjalizuje wiele modułów sprzętowych wykorzystywanych przez IDF-SDK.

Punkt wejściowy bootloadera 2-go poziomu znajduje się na chwilę obecną w komponencie bootloader. Jest to funkcja call_start_cpu(). Punkt wejściowy aplikacji użytkownika to z kolei również funkcja call_start_cpu() ale z komponentu esp8266 (plik startup.c).

Jeżeli Flash zawiera więcej niż jedną aplikację użytkownika, to w zależności od tego, czy znajduje się ona w tym samym, czy w innym 1MiB-ajtowym segmencie, co bootloader 2-go poziomu, przed załadowaniem programu do RAMu przeprowadzi on remapowanie pamięci Flash w przestrzeń adresową. Należy zwrócić uwagę, że wówczas kod bootloader musi znajdować się całkowicie w RAMie. W związku z tym początkowe zapotrzebowanie aplikacji na RAM musi być odpowiednio małe, aby bootloader nie nadpisał swojego, aktualnie działającego kodu oraz zaalokowanych danych.



3.2 Partycjonowanie pamięci Flash

IDF-SDK wykorzystuje partycjonowanie pamięci Flash. Dzięki temu jest ona korzystać z niej w sposób zorganizowany, co umożliwia jednoczesne przechowywanie a pamięci wielu aplikacji równolegle z systemami plików, czy obszarami danych konfiguracyjncyh.

Powyższy obraz ukazuje podstawy schematu partycjonowania w IDF-SDK (po szczegółwy dotyczące partycji zobacz dokumentację IDF). Na początku pamięci Flash znajduje się zawsze 8-bajtowy nagłówek obrazu bootloadera drugiego poziomu (oczywiście 'zawsze' tak długo, jak mówimy o IDF-SDK). Jest on wykorzystywany przez bootloader 1-go poziomu do załadowania bootloadera 2-go poziomu. Ten ostatni rezyduje za nagłówkiem obrazu i mieście się zawsze w pierwszych 64KiB pamięci Flash.

Za bootloaderem umieszczane są partycje danych związanych z SDK. Są to (w zależności od konfiguracji) partycje NVS (ang Non-Volatile Storage) oraz OTA (ang. Over The Air). Następne w kolejności są partycje aplikacji. Każda taka partycja ma swój nagłówek, który tak jak w przypadku bootloadera 2-go poziomu określa m.in. adres wejściowy aplikacji. Po nagłówku następują kolejne segmenty programu (uproszczone rozwiązanie znane z linuxowych plików .elf). Każdy segment zaczyna się od nagłówka, który określa w jakim obszarze pamięci musi znaleźć się segment przed rozpoczęciem wykonywania aplikacji. Dzięki temu bootloader 2-go poziomu wie, które fragmenty należy przekopiować do RAMu. Segment kończy się sumą kontrolną (na chwilę obecną jest to hash SHA256). Kod i dane statyczne (czyli te, które docelowo rezydują we Flashu) są umieszczone po segmentach i nie wymagają już dedykowanego nagłówka.

Ostatnie 4KiB każdego MiB pamięci Flash zarezerwowane są dla danych konfiguracyjnych WiFi. Dzięki temu, niezależnie od tego który segment 1MiB zostanie zmapowany do przestrzeni adresowej, SDK będzie miało do nich dostęp.

Rozdział 4

Komponenty

4.1 esp8266

Komponent esptool8266 zawiera wszystkie kluczowe funkcje systemu (aplikacji użytkownika). Przechowuje on podstawowe biblioteki dostarczane przez Xtensa'ę, sterowniki peryferiów oraz implementację biblioteki standardowej. Zawiera także implementację biblioteki standardowej języka C oraz domyślne skrypty linkera. To tutaj znajduje się punkt wejściowy aplikacji.